PySCF Documentation

Release 1.7.0

Qiming Sun <osirpt.sun@gmail.com>

Oct 12, 2019
1 Contents
1.1 An overview of PySCF ................................. 3
1.2 Version history ........................................... 7
1.3 Installation .............................................. 8
1.4 Code standard ............................................ 13
1.5 Benchmark .............................................. 15
1.6 Tutorial .................................................. 15
1.7 gto — Molecular structure and GTO basis ........ 25
1.8 ao2mo — Integral transformations ................. 89
1.9 df — Density fitting .................................... 100
1.10 symm — Point group symmetry and spin symmetry. 109
1.11 scf — Mean-field methods ............................ 115
1.12 dft — Density functional theory .................. 213
1.13 dfd3 — DFT plus Dispersion Correction .......... 240
1.14 grad — Analytical nuclear gradients .......... 240
1.15 hessian — Analytical nuclear Hessian .......... 248
1.16 soscf — Second order Hartree-Fock solver ...... 250
1.17 sgx — Pseudo-spectral methods (COSX, PS, SN-K) . 251
1.18 tddft — Time dependent density functional theory 252
1.19 tdsf — Time dependent SCF .......................... 260
1.20 mp — MP2 .............................................. 260
1.21 ci — Configuration interaction ..................... 263
1.22 doci — Doubly occupied configuration interaction 267
1.23 cc — Coupled cluster .................................. 267
1.24 gw — Molecular G0W0 .................................. 275
1.25 fci — Full configuration interaction ............. 276
1.26 hci — Interface to Heat-Bath selective CI .......... 289
1.27 cornell_shci — Interface to Fast Semistochastic Heat Bath Configuration Interaction (SHCI) .... 290
1.28 mcsf — Multi-configurational self-consistent field 290
1.29 dmrsgscf .............................................. 313
1.30 fciqmcscf .............................................. 314
1.31 mrpt — Multi-reference perturbation theory .... 315
1.32 icmpspt — Internal-contracted MPS perturbation method 315
1.33 x2c — exact-two-component approach .......... 316
1.34 pbc — Periodic boundary conditions ............ 317
1.35 qmmm — QM/MM interface ........................... 354
1.36 solvent — Solvent methods .......................... 356
1.37 semiempirical — Semiempirical methods .......... 359
1.38 geomopt — Geometry optimization ............... 361
1.39 lo — Orbital localization and analysis tools ..... 362
1.40 tools ................................................................. 366
1.41 Miscellaneous ..................................................... 374
1.42 ccn — Auto-Generated Coupled-Cluster Equations of Arbitrary Order ................................. 374
1.43 nao — Numerical Atomic Orbitals .............................. 377
1.44 lib — Helper functions, parameters, and C extensions ......................................................... 386
1.45 prop — Molecular properties .................................... 399
1.46 data — Data for constants ....................................... 413
1.47 shcisCF — Semistochastic Heat Bath Configuration Interaction for SCF ......................... 414

Python Module Index .................................................................................................................. 415
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).

In addition to the core libraries, PySCF supports a rich ecosystem of plugins and external modules that, for example, provide MPI versions of some routines, additional quantum chemistry methods and analysis, interface with quantum computing toolkits etc. See :ref:`.. _installing_plugin:`.
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 How to cite

Bibtex entry:

```latex
@Misc{PYSCF,
  title = {PySCF: the Python-based simulations of chemistry framework},
  author = {Qiming Sun and Timothy C. Berkelbach and Nick S. Blunt and George H. \rightarrow
  Booth and Sheng Guo and Zhendong Li and Junzi Liu and James D. McClain and Elvira R. \rightarrow
  Sayfutyarova and Sandeep Sharma and Sebastian Wouters and Garnet KinLic Chan},
  year = {2017},
  journal = {Wiley Interdisciplinary Reviews: Computational Molecular Science},
  volume = {8},
  number = {1},
  pages = {e1340},
  doi = {10.1002/wcms.1340},
}
```

In addition, if you use Libcint to compute integrals, please cite the following paper:


1.1.2 Features

- 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
• Interface to tensor contraction library TBLIS
• Interface to Heat-bath Selected CI program Dice
• Interface to geometry optimizer Pyberny

1.1.3 Designs

Kernel and Stream functions

Every class has the kernel method which serves as the entry or the driver of the method. Once an object of one method was created, you can always call .kernel() to start or restart a calculation.

The return value of kernel method is different for different class. To unify the return value, the package introduces the stream methods to pipe the computing stream. A stream method of an object only return the object itself. There are three general stream methods available for most method classes. They are:

1. .set method to update object attributes, for example:

```python
mf = scf.RHF(mol).set(conv_tol=1e-5)
```

is identical to two lines of statements:

```python
mf = scf.RHF(mol)
mf.conv_tol = 1e-5
```

2. .run method to pass the call to the .kernel method. If arguments are presented in .run method, the arguments will be passed to the kernel function. If keyword arguments are given, .run method will first call .set method to update the attributes then execute the .kernel method. For example:

```python
mf = scf.RHF(mol).run(dm_init, conv_tol=1e-5)
```

is identical to three lines of statements:

```python
mf = scf.RHF(mol)
mf.conv_tol = 1e-5
mf.kernel(dm_init)
```

3. .apply method to pass the current object (as the first argument) to the given function/class and return a new object. If arguments and keyword arguments are presented, they will all be passed to the function/class. For example:

```python
mc = mol.apply(scf.RHF).run().apply(mcscf.CASSCF, 6, 4, frozen=4)
```

is identical to:

```python
mf = scf.RHF(mol)
mf.kernel()
mc = mcscf.CASSCF(mf, 6, 4, frozen=4)
```

Aside from the three general stream methods, the regular class methods may return the objects as well when the methods do not have particular value to return. Using the stream methods, you can evaluate certain quantities with one line of code:
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. Intermediates are NOT saved in the class. After calling the .kernel() or .run() method, results will be generated and saved in the object. For example:

```python
from pyscf import gto, scf, ccsd
mol = gto.M(atom='H 0 0 0; H 0 0 1.1', basis='ccpvtz')
mf = scf.RHF(mol).run()
mycc = ccsd.CCSD(mf).run()
print(mycc.e_tot)
print(mycc.e_corr)
print(mycc.t1.shape)
print(mycc.t2.shape)
```

Many useful functions are defined at both the module level and class level. They can be accessed from either the module functions or the class methods and the return values should be the same:

```python
vj, vk = scf.hf.get_jk(mol, dm)
vj, vk = SCF(mol).get_jk(mol, dm)
```

Note some module functions may require the class as the first argument.

Most functions and classes are pure, i.e. no intermediate status are held within the classes, and the argument of the methods and functions are immutable during calculations. These functions can be called arbitrary times in arbitrary order and their returns should be always the same.

Exceptions are often suffixed with underscore in the function name, e.g. mcscf.state_average_(mc) where the attributes of mc object may be changed or overwritten by the state_average_ method. Cautious should be taken when you see the functions or methods with ugly suffices.

Global configurations

Global configuration file is a Python script that contains PySCF configurations. When importing pyscf module in a Python program (or Python interpreter), the package will preload the global configuration file and take the configurations as the default values of the parameters of functions or attributes of classes during initialization. For example, the configuration file below detects the available memory in the operate system at the runtime and set the maximum memory for PySCF:

```python
$ cat ~/.pyscf_conf.py
import psutil
total, available, percent, used, free, active, inactive, buffers, cached, shared = psutil.virtual_memory()
MAX_MEMORY = available
```

By setting MAX_MEMORY in the global configuration file, you don’t need the statement to set the max_memory attribute in every script. The dynamically determined max_memory will be loaded during the program initialization step automatically.

1.1. An overview of PySCF
There are two methods to let the PySCF package load the global configurations. One is to create a configuration file .pyscf_conf.py in home directory or in work directory. Another is to set the environment variable PYSCF_CONFIG_FILE which points to the configuration file (with the absolute path). The environment variable PYSCF_CONFIG_FILE has high priority than the configuration file in default locations (home directory or work directory). If environment variable PYSCF_CONFIG_FILE is available, the program will read the configurations from the $PYSCF_CONFIG_FILE. If PYSCF_CONFIG_FILE is not set or the file it points to does not exist, the program will turn to the default location for the file .pyscf_conf.py. If none of the configuration file exists, the program will use the built-in configurations which are generally conservative settings.

In the source code, global configurations are loaded by importing pyscf.__config__ module:

```python
from pyscf import __config__
MAX_MEMORY = getattr(__config__, 'MAX_MEMORY')
```

Please refer to the source code for the available configurations.

### Scanner

Scanner is a function that takes an Mole (or Cell) object as input and return the energy or nuclear gradients of the given Mole (or Cell) object. Scanner can be considered as a shortcut function for a sequence of statements which includes the initialization of a required calculation model with necessary precomputing, next updating the attributes based on the settings of the referred object, then calling kernel function and finally returning results. For example:

```python
cc_scanner = gto.M().apply(scf.RHF).apply(cc.CCSD).as_scanner()
for r in (1.0, 1.1, 1.2):
    print(cc_scanner(gto.M(atom='H 0 0 0; H 0 0 %g' %r)))
```

An equivalent but slightly complicated code is:

```python
for r in (1.0, 1.1, 1.2):
    mol = gto.M(atom='H 0 0 0; H 0 0 %g' %r)
    mf = scf.RHF(mol).run()
    mycc = cc.CCSD(mf).run()
    print(mycc.e_tot)
```

There are two types of scanner available in the package. They are energy scanner and nuclear gradients scanner. The example above is the energy scanner. Energy scanner only returns the energy of the given molecular structure while the nuclear gradients scanner returns the nuclear gradients in addition.

Scanner is a special derived object of the caller. Most methods which are defined in the caller class can be used with the scanner object. For example:

```python
mf_scanner = gto.M().apply(scf.RHF).as_scanner()
mf_scanner(gto.M(atom='H 0 0 0; H 0 0 1.2'))
mf_scanner.analyze()
dm1 = mf_scanner.make_rdm1()

mf_grad_scanner = mf_scanner.nuc_grad_method().as_scanner()
mf_grad_scanner(gto.M(atom='H 0 0 0; H 0 0 1.2'))
```

As shown in the example above, the scanner works pretty close to the relevant class object except that the scanner does not need the kernel or run methods to run a calculation. Given molecule structure, the scanner automatically checks and updates the necessary object dependence and passes the work flow to the kernel method. The computational results are held in the scanner object as the regular class object does.

To make structure of scanner object uniform for all methods, two attributes (.e_tot and .converged) are defined for all energy scanner and three attributes (.e_tot, .de and .converged) are defined for all nuclear gradients
### 1.2 Version history

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.6.4</td>
<td>2019-09-14</td>
</tr>
<tr>
<td>1.6.3</td>
<td>2019-07-28</td>
</tr>
<tr>
<td>1.6.2</td>
<td>2019-06-17</td>
</tr>
<tr>
<td>1.6.1</td>
<td>2019-03-15</td>
</tr>
<tr>
<td>1.6</td>
<td>2018-12-31</td>
</tr>
<tr>
<td>1.6 beta</td>
<td>2018-11-26 (feature frozen)</td>
</tr>
<tr>
<td>1.6 alpha</td>
<td>2018-08-15</td>
</tr>
<tr>
<td>1.5.5</td>
<td>2018-12-31</td>
</tr>
<tr>
<td>1.5.4</td>
<td>2018-11-16</td>
</tr>
<tr>
<td>1.5.3</td>
<td>2018-09-06</td>
</tr>
<tr>
<td>1.5.2</td>
<td>2018-08-15</td>
</tr>
<tr>
<td>1.5.1</td>
<td>2018-07-01</td>
</tr>
<tr>
<td>1.5</td>
<td>2018-06-08</td>
</tr>
<tr>
<td>1.5 beta</td>
<td>2018-04-15 (feature frozen)</td>
</tr>
<tr>
<td>1.5 alpha</td>
<td>2018-03-21</td>
</tr>
<tr>
<td>1.4.7</td>
<td>2018-04-15</td>
</tr>
<tr>
<td>1.4.4</td>
<td>2018-03-20</td>
</tr>
<tr>
<td>1.4</td>
<td>2017-10-05</td>
</tr>
<tr>
<td>1.4 beta</td>
<td>2017-08-22 (feature frozen)</td>
</tr>
<tr>
<td>1.4 alpha</td>
<td>2017-06-24</td>
</tr>
<tr>
<td>1.3.5</td>
<td>2017-08-12</td>
</tr>
<tr>
<td>1.3.4</td>
<td>2017-08-09</td>
</tr>
<tr>
<td>1.3.3</td>
<td>2017-07-05</td>
</tr>
<tr>
<td>1.3.2</td>
<td>2017-06-05</td>
</tr>
<tr>
<td>1.3.1</td>
<td>2017-05-14</td>
</tr>
<tr>
<td>1.3</td>
<td>2017-04-25</td>
</tr>
<tr>
<td>1.3 beta</td>
<td>2017-02-15 (feature frozen)</td>
</tr>
<tr>
<td>1.3 alpha 2</td>
<td>2017-01-04</td>
</tr>
<tr>
<td>1.3 alpha 1</td>
<td>2016-12-04</td>
</tr>
<tr>
<td>1.2.3</td>
<td>2017-04-24</td>
</tr>
<tr>
<td>1.2.2</td>
<td>2017-02-15</td>
</tr>
<tr>
<td>1.2.1</td>
<td>2017-01-26</td>
</tr>
<tr>
<td>1.2</td>
<td>2016-11-07</td>
</tr>
<tr>
<td>1.2 beta</td>
<td>2016-09-13 (feature frozen)</td>
</tr>
<tr>
<td>1.2 alpha</td>
<td>2016-08-05</td>
</tr>
<tr>
<td>1.1</td>
<td>2016-06-04</td>
</tr>
<tr>
<td>1.1 beta</td>
<td>2016-04-11 (feature frozen)</td>
</tr>
<tr>
<td>1.1 alpha 2</td>
<td>2016-03-08</td>
</tr>
<tr>
<td>1.1 alpha 1</td>
<td>2016-02-08</td>
</tr>
<tr>
<td>1.0</td>
<td>2015-10-07</td>
</tr>
<tr>
<td>1.0 rcl</td>
<td>2015-09-07</td>
</tr>
<tr>
<td>1.0 beta 1</td>
<td>2015-08-02 (feature frozen)</td>
</tr>
<tr>
<td>1.0 alpha 2</td>
<td>2015-07-03</td>
</tr>
<tr>
<td>1.0 alpha 1</td>
<td>2015-04-07</td>
</tr>
</tbody>
</table>
1.3 Installation

1.3.1 Installation with pip

This is the recommended way to install PySCF:

```bash
$ pip install pyscf
```

Pypi provides a precompiled PySCF code (python wheel) which works on almost all Linux systems, and most of Mac OS X systems, and the ubuntu subsystems on Windows 10. If you already have pyscf installed, you can upgrade it to the new version:

```bash
$ pip install --upgrade pyscf
```

Note we observed that our precompiled python wheels sometimes does not work with certain version of Python (python-3.4 and python-3.5). If you’re using mac OS X with python-3.4 or python-3.5, pip may execute the setup.py file in the source code and it may be terminated due to an error of library path of BLAS library. BLAS library is required to install PySCF library. The installation script can detect the installed BLAS libraries in the system and choose one for the program. If BLAS library is existed but wasn’t found by the installation script, you can specify the BLAS library either through the environment LDFLAGS, e.g. LDFLAGS="-L/path/to/blas -lblas" pip install pyscf or the environment variable PYSCF_INC_DIR, e.g. PYSCF_INC_DIR=/path/to/blas:/path/to/other/lib pip install to tell the installation script which BLAS libraries to link against. Another issue of the installation script you may get is that pyscf.dft module is not working. pyscf.dft module requires the exchange-correlation functional library libxc which was not yet available in the PyPI repository. To enable pyscf.dft module, you can download and manually compile libxc library and set the environment variable PYSCF_INC_DIR, e.g. export PYSCF_INC_DIR=/path/to/libxc before calling pip install command. Libxc library can be found in http://octopus-code.org/wiki/Libxc:download. When compiling the libxc library, you need to add --enable-shared flag.

1.3.2 Installation with conda

If you have Conda (or Anaconda) environment, PySCF package can be installed with Conda cloud:

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

1.3.3 PySCF docker image

The following command starts a container with the jupyter notebook server listening for HTTP connections on port 8888:

```bash
$ docker run -it -p 8888:8888 pyscf/pyscf-1.5.0
```

Then visit https://localhost:8888 with your browser to use notebook and pyscf.

Another way to use PySCF in docker container is to start an Ipython shell:

```bash
$ docker run -it pyscf/pyscf-1.5.0 start.sh ipython
```

1.3.4 Manual installation from github repo

Manual installation requires cmake, numpy, scipy and h5py libraries. You can download the latest PySCF (or the development branch) from github:
Build the C extensions in pyscf/lib:

```bash
$ 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 xfun. Finally, to make Python 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:

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

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

```bash
>>> import pyscf
```

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

```bash
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
```

This is caused by the incorrect RPATH. Script pyscf/lib/_runme_to_fix_dylib_osx10.11.sh in pyscf/lib directory can be used to fix this problem:

```bash
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.

Last, it’s recommended to set a scratch directory for PySCF. The default scratch directory is controlled by environment variable PYSCF_TMPDIR. If it’s not specified, the system temporary directory TMPDIR will be used as the scratch directory.

### 1.3.5 Installation without network

If you have problems to download 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 xfun libraries. libcint cint3 branch and xfun stable-1.x branch are required by PySCF. They can be downloaded from github:
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:

```bash
$ 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/dftlibs/xcfun.git
$ cd xcfun
$ git checkout 355f42497a9cd17d16ae91d5af1aaaf93756ae8b
$ cd .. && tar czf xcfun.tar.gz xcfun
```

Next, compile PySCF:

```bash
$ 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.

### 1.3.6 Using optimized BLAS

The default installation tries to find BLAS libraries automatically. This automated setup script may link the code to slow BLAS libraries. You can compile the package with other BLAS vendors to improve performance, for example the Intel Math Kernel Library (MKL), which can provide 10x speedup in many modules:

```bash
$ cd pyscf/lib/build
$ cmake -DBLA_VENDOR=Intel10_64lp_seq ..
$ make
```
When linking the program to MKL library, for some MKL versions, cmake may have problems to find the correct MKL libraries. Setting `LD_LIBRARY_PATH` to include the MKL dynamic libraries sometimes can help cmake to find the MKL libraries, e.g.:

```
export LD_LIBRARY_PATH=/opt/intel/compilers_and_libraries_2018/linux/mkl/lib/intel64:
  $LD_LIBRARY_PATH
```

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`, e.g. `BLA_VENDOR=ATLAS`, `BLA_VENDOR=IBMESSL`, `BLA_VENDOR=OpenBLAS` (requiring cmake-3.6). Please refer to cmake manual 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")
```

**Note:** MKL library may lead to an OSError at runtime: `OSError: ... mkl/lib/intel64/libmkl_avx.so: undefined symbol: ownLastTriangle_64fc or MKL FATAL ERROR: Cannot load libmkl_avx.so or libmkl_def.so.. It can be solved by preloading MKL core library with: `export LD_PRELOAD=$MKLROOT/lib/intel64/libmkl_avx.so:$MKLROOT/lib/intel64/libmkl_def.so`.

### 1.3.7 Using optimized integral library

The default integral library used by PySCF is libcint (https://github.com/sunqm/libcint). This integral library was implemented in the model that ensures the compatibility on various high performance computer systems. For X86-64 platforms, libcint library has an efficient counterpart Qcint (https://github.com/sunqm/qcint) which is heavily optimized against X86 SIMD instructions (AVX-512/AVX2/AVX/SSE3). 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
  ...
```

### 1.3.8 Cmake config file

Cmake options can be saved in a config file `pyscf/lib/cmake.arch.inc`. Settings in this config file will be automatically loaded and overwritten the default cmake options during compilation. For example, you can put `CMAKE_C_FLAGS` in this config file to include advanced compiler optimization flags:

### 1.3. Installation

11
Other settings and variables and flags can all be put in this config file:

```cpp
set(ENABLE_XCFUN Off)
set(WITH_F12 Off)
```

Some examples of platform specific configurations can be found in directory `pyscf/lib/cmake_arch_config`.

### 1.3.9 Plugins

**nao**

The `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:

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

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

**DMRG solver**

Density matrix renormalization group (DMRG) implementations Block (https://sanshar.github.io/Block) 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 https://sanshar.github.io/Block/build.html. Before using the Block or CheMPS2, you need create a configuration file `pyscf/dmrgscf/settings.py` (as shown by settings.py.example) to store the path where the DMRG solver was installed.

**Heat-bath Selected CI**

Dice is an efficient implementation for heat-bath selected CI (SHCI) algorithm. It can be used with the CASCI and CASSCF module to solve large active space problems. The method to use SHCI is very much like the use of DMRG program. The path of Dice program and other configurations should be initialized in the configuration file `pyscf/shci/settings.py` before using the SHCI method.

**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 3.0.0. `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.

TBLIS

TBLIS provides a native algorithm to perform tensor contraction for arbitrary high dimensional tensors. The native algorithm does not need to transform tensors into matrices then call the BLAS libraries for the matrix contraction. Tensor transposing and data moving are largely avoided in TBLIS tensor library. The interface to TBLIS offers an efficient implementation for `numpy.einsum()` style tensor contraction. To enable the tlibs-einsum plugin, you can set the cmake flags `-DENABLE_TBLIS=ON` when compiling the C extensions:

```
$ cmake -DENABLE_TBLIS=ON ..
```

Note TBLIS library was implemented with C++11 standard. You need at least GCC 5.2 to compile this plugin.

Pyberny

The geometry optimizer Pyberny provides an independent implementation that supports various geometry optimization techniques (comprising redundant internal coordinates, iterative Hessian estimate, trust region, line search, and coordinate weighing etc.). It can take the output of PySCF Gradients Scanner and generate new geometry to feed back to PySCF program. The geometry optimization `geomopt` exposes a wrapper function to simplify the geometry optimization setup:

```
from pyscf import gto, scf, geomopt
mf = gto.M(atom='H 0 0 0; H 0 0 1.').apply(scf.RHF)
mol_eq = geomopt.optimize(mf)
```

XianCI

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

1.4 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
  - 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 - Compatible 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

1.4.1 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.

1.4.2 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 mutes 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 environments (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 environments. They are not supposed to be changed by other functions.

  – Kernel functions Although the method classes have various entrance/main function, 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 class methods’ arguments. Many class methods can take the results of the calculations which were held in the 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.

## 1.5 Benchmark

<table>
<thead>
<tr>
<th>Platform</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>4 Intel E5-2670 @ 2.6 GB</td>
</tr>
<tr>
<td>Memory</td>
<td>64 GB DDR3</td>
</tr>
<tr>
<td>OS</td>
<td>Custom Redhat 6.6</td>
</tr>
<tr>
<td>BLAS</td>
<td>MKL 11.0</td>
</tr>
<tr>
<td>Compiler</td>
<td>Intel 13.0</td>
</tr>
</tbody>
</table>

**Benzene**, on 16 CPU cores

<table>
<thead>
<tr>
<th>Basis</th>
<th>6-31G**</th>
<th>cc-pVTZ</th>
<th>ANO-Roos-TZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>HF</td>
<td>0.55 s</td>
<td>5.76 s</td>
<td>389.1 s</td>
</tr>
<tr>
<td>density fit HF</td>
<td>3.56 s</td>
<td>7.61 s</td>
<td>13.8 s</td>
</tr>
<tr>
<td>B3LYP</td>
<td>3.84 s</td>
<td>11.44 s</td>
<td>360.2 s</td>
</tr>
<tr>
<td>MP2</td>
<td>0.21 s</td>
<td>4.66 s</td>
<td>115.9 s</td>
</tr>
<tr>
<td>CASSCF(6,6)</td>
<td>2.88 s</td>
<td>34.73 s</td>
<td>639.7 s</td>
</tr>
<tr>
<td>CCSD</td>
<td>18.24 s</td>
<td>477.0 s</td>
<td>6721 s</td>
</tr>
</tbody>
</table>

**C60**, on 16 CPU cores

<table>
<thead>
<tr>
<th>Basis</th>
<th>6-31G**</th>
<th>cc-pVTZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>HF</td>
<td>1291 s</td>
<td>189 m</td>
</tr>
<tr>
<td>SOSCF (newton)</td>
<td>77 m</td>
<td></td>
</tr>
<tr>
<td>density fit HF</td>
<td>316.7 s</td>
<td>43.3 m</td>
</tr>
</tbody>
</table>

**Fe(II)-porphyrin** (FeC20H12N4), on 16 CPU cores

<table>
<thead>
<tr>
<th>Basis</th>
<th>cc-pVDZ</th>
<th>cc-pVTZ</th>
<th>cc-pVQZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOSCF (newton)</td>
<td>193.4 s</td>
<td>20.1 m</td>
<td>127.1 m</td>
</tr>
<tr>
<td>CASSCF(10,10)</td>
<td>1808 s</td>
<td>241 m</td>
<td></td>
</tr>
<tr>
<td>CASSCF(11,8)</td>
<td>763.8 s</td>
<td>150.3 m</td>
<td>1280 m</td>
</tr>
</tbody>
</table>

## 1.6 Tutorial

This tutorial shows how to use PySCF package in the perspective of method development. It involves some knowledge of coding in Python. An IPython notebook of user-guide can be found in https://github.com/nmardirossian/PySCF_Tutorial.

### 1.6.1 Quick setup

You can install PySCF from github repo:
You may need to update the Python runtime searching path PYTHONPATH (assuming the pyscf source code is put in /home/abc, replacing it with your own path):

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

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

### 1.6.2 A simple example

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

```python
>>> 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
```

### 1.6.3 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:

```python
>>> 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:

```python
>>> 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():
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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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`:

```python
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:

```python
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 unpaired electrons = the difference between the numbers of alpha and beta electrons.

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

```python
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:

```python
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:

```python
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
```

### 1.6.4 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:

```python
>>> 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:
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.

### 1.6.5 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:

```python
>>> 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:

```python
>>> 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):

```python
>>> 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:

```python
>>> 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:
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:

```python
>>> 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)))
```

```bash
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:

```python
>>> 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`:

```python
>>> from pyscf import mp
>>> mp2 = mp.MP2(m)
```
This is the correlation energy of singlet ground state. For the triplet state, we can write a function to compute the correlation energy

\[ E_{\text{corr}} = \frac{1}{4} \sum_{ijab} \langle ij | ab \rangle \langle ab | ij \rangle \left( \epsilon_i + \epsilon_j - \epsilon_a - \epsilon_b \right) \]

```python
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
```

In this example, we concatenate \( \alpha \) and \( \beta \) 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:

```python
>>> 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

\[
\langle \left( \frac{\partial}{\partial R} \phi_i \right) | \phi_k \rangle \phi_j | \phi_l \rangle = \int \frac{\partial^2 \langle r_1 | r_2 \rangle}{\partial R} \phi_j (r_1) \phi_k (r_2) \phi_i (r_2) \phi_l (r_2) \frac{d r_1 d r_2}{|r_1 - r_2|}
\]

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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.

### 1.6.6 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 checkpoint 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:

### 1.6.7 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()
aux = auxmol.nao_nr()
eri3c = numpy.empty((nao,nao,aux))
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
```

```
```

### 1.6. Tutorial 23

---

**PySCF**

PySCF is an open-source Python library for quantum chemistry. It provides a comprehensive set of tools for performing electronic structure calculations, including Hartree-Fock, density functional theory (DFT), and configuration interaction (CI) methods.

**Documentation**

PySCF's documentation is available at the official website: [PySCF Documentation](https://www.pyscf.org). The documentation includes detailed descriptions of all functions, classes, and modules, along with examples and tutorials.

**Installation**

PySCF can be installed via `pip` or `conda`. The latest version is available on PyPI: [PySCF on PyPI](https://pypi.org/project/pyscf/).

**Contributing**

If you would like to contribute to PySCF, you can find the contribution guidelines and issue tracker on the official repository: [PySCF on GitHub](https://github.com/pyscf/pyscf).

---

**Acknowledgments**

The PySCF project is supported by contributions from a global community of researchers and developers. Special thanks to: [Sponsorship List](https://www.pyscf.org/sponsors/).

---

**Contact**

If you have questions or feedback, please feel free to contact us: [Contact Information](https://www.pyscf.org/contact/).

---

**License**

PySCF is licensed under the [Apache License 2.0](https://www.apache.org/licenses/LICENSE-2.0). Contributions are expected to follow the [Contribution Guidelines](https://www.pyscf.org/contributing/).

---

**Contributors**

This project is the result of collective effort. Special thanks to all contributors: [Contribution List](https://www.pyscf.org/contributors/).

---

**Acknowledgements**

This work was supported by: [Grant List](https://www.pyscf.org/acknowledgements/).

---

**Attributions**

PySCF uses code from the following libraries: [Attributions](https://www.pyscf.org/attributions/).

---

**Legal**

For legal information, please refer to [Legal Notice](https://www.pyscf.org/legal/).
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:

```python
eri2c = numpy.empty((naux,naux))
for k in range(mol.nbas, mol.nbas+auxmol.nbas):
    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.

```python
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
```

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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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.

1.6.8 Other features

Density fitting

Customizing Hamiltonian

Symmetry in CASSCF

1.7 gto — Molecular structure and GTO basis

The gto 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.

1.7.1 Examples
1.7.2 Input

Geometry

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

```python
cartesian_list = [[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:

```python
>>> 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:

```python
>>> 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:

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

>>> mol.natm
2
```

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

```python
>>> 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:

```python
>>> 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:

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

```python
>>> 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]]]
```

In the program, the molecular geometry is accessed with Mole.atom_coords() function. This function returns a (N,3) array for the coordinates (in Bohr) of each atom:

```python
>>> print(mol.atom_coords())
[[ 0. 0. 0.]
 [ 0. 1.88972612 0.]
 [ 0. 0. 1.88972612]]
```

**Input Basis**

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

```python
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):

```python
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.

```python
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`).

Basis parser supports “Ghost” atom:

```python
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:

```python
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:

```bash
$ 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:

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

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

**Spin and charge**

Charge and spin multiplicity can be assigned to `Mole` object:

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

Note `Mole.spin` is the number of unpaired electrons (difference between the numbers of alpha electrons and beta electrons). These two attributes do not affect any other parameters in the `Mole.build` initialization function. They can be set or modified (although not recommended to do so) after the molecular object is initialized:

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

`Mole.charge` is the parameter to define the total number electrons of the system. In a custom system such as the Hubbard lattice model, the total number of electrons needs to be defined directly:
>>> mol = gto.Mole()
>>> mol.nelectron = 10

Symmetry

Point group symmetry information is held in Mole object. The symmetry module (symm – Point group symmetry and spin symmetry) of PySCF program can detect arbitrary point groups. In the SCF calculations, PySCF program supports linear molecular symmetry $D_{\infty h}$ (labelled as Dooh in the program), $C_{\infty v}$ (labelled as Coov) plus $D_{2h}$ and its subgroups.

If the attribute Mole.symmetry is set, Mole.build() function will detect the top point group and the supported subgroups. The detected point groups are saved in Mole.topgroup and Mole.groupname:

```python
>>> mol = gto.Mole()
>>> mol.atom = 'B 0 0 0; H 0 1 1; H 1 0 1; H 1 1 0'
>>> mol.symmetry = True
>>> mol.build()
>>> print(mol.topgroup)
C3v
>>> print(mol.groupname)
Cs
```

Also, during the Mole object initialization, the program will move the charge center of the system to the origin (0,0,0) and place the main rotation axis on z-axis (if available, see symm – Point group symmetry and spin symmetry for more details about how the main axis is determined). In the last example, the molecule is rotated to:

```python
>>> print(mol.atom_charges())
[[ 0.00000000e+00 -8.18275415e-01 0.00000000e+00]
 [ -7.71477460e-01 1.36379236e+00 -1.33623816e+00]
 [ -7.71477460e-01 1.36379236e+00 1.33623816e+00]
 [ 1.54295492e+00 1.36379236e+00 -2.22044605e-16]]
```

Sometimes it is necessary to use a lower symmetry instead of the detected symmetry group. The subgroup symmetry can be specified in Mole.symmetry_subgroup and the program will first detect the highest possible symmetry group and lower the point group symmetry to the specified subgroup:

```python
>>> mol = gto.Mole()
>>> mol.atom = 'N 0 0 0; N 0 0 1'
>>> mol.symmetry = 'coov'
>>> mol.build()
>>> print(mol.topgroup)
coov
>>> print(mol.groupname)
1
```

In many situations, you may require the program to use the point group symmetry in different orientation. This can be achieved by explicit specification of the symmetry elements:
When a particular symmetry was assigned to `Mole.symmetry`. The program will use the given symmetry group for the system and use the input orientation. The initialization function `Mole.build()` will test whether the input symmetry matches the input orientation. If the given symmetry group does not agree to the actual input symmetry, the initialization will stop and issue an error message:

```python
>>> mol = gto.Mole()
>>> mol.atom = 'N 0 0 0; N 0 0 1'
>>> mol.symmetry = 'Dooh'
>>> mol.build()
RuntimeWarning: Unable to identify input symmetry Dooh. Try symmetry="Dooh" with geometry (unit="Bohr")
(('N', [0.0, 0.0, -0.9448630622825309])
(('N', [0.0, 0.0, 0.9448630622825309])
```

Note if Z-matrix is given in the input, the molecule may be placed in an arbitrary orientation. Although still works, specifying `Mole.symmetry` often leads to the above error message.

Note: `Mole.symmetry_subgroup` does not have effects when specific symmetry group is assigned to `Mole.symmetry`.

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

```python
>>> mol = gto.Mole()
>>> mol.atom = 'N 0 0 0; N 0 0 1'
>>> mol.symmetry = True
>>> mol.build()
>>> for s,i,c in zip(mol.irrep_name, mol.irrep_id, mol.symm_orb):
...     print(s, i, c.shape)
Alg 0 (10, 3)
Elgx 2 (10, 1)
Elgy 3 (10, 1)
Alu 5 (10, 3)
Eluy 6 (10, 1)
Elux 7 (10, 1)
```

## 1.7.3 Program reference

mole

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

<table>
<thead>
<tr>
<th>Input</th>
<th>Python</th>
<th>Libcint</th>
</tr>
</thead>
<tbody>
<tr>
<td>atom</td>
<td>.atom</td>
<td>.atm</td>
</tr>
<tr>
<td>basis</td>
<td>.basis</td>
<td>.bas</td>
</tr>
</tbody>
</table>
input layer does not talk to libcint directly. Data are held in python internal format 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.

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

This is a shortcut to build up Mole object.

Args: Same to Mole.build()

Examples:

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

```python
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 or None] 2S, num. alpha electrons - num. beta electrons to control multiplicity. If spin = None is set, multiplicity will be guessed based on the neutral molecule.
- **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 molecular structure. The internal format is

  ```python
  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 or [function(nuc_charge, nucprop) => zeta]] Nuclear model. 0 or None means point nuclear model. Other values will enable Gaussian nuclear model. If a function is assigned to this attribute, the function will be called to generate the nuclear charge distribution value “zeta” and the relevant nuclear model will be set to Gaussian model. Default is point nuclear model.
- **nucprop** [dict] Nuclear properties (like g-factor ‘g’, quadrupole moments ‘Q’). It is needed by pyscf.prop module and submodules.
- **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
topgroup  [str] Point group of the system.

groupname  [str] The supported subgroup of the point group. It can be one of Dooh, Coov, 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 is to ensure certain functions being initialized only 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 represents one atom

natm : number of atoms

_bas : [[atom-id,angular-momentum,num-primitive-GTO,num-contracted-GTO,0,ptr-of-exps,ptr-of-contract-coeff,0],...] each element represents one shell

_nbas : number of shells

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

Examples:

```python
>>> 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
```

ao2mo  (mo_coeffs, erifile=None, dataname='eri_mo', intor='int2e', **kwargs)

Integral transformation for arbitrary orbitals and arbitrary integrals. See more detailed documentation in func:ao2mo.kernel.

Args:

mo_coeffs (an np array or a list of arrays) A matrix of orbital coefficients if it is a numpy ndarray, or four sets of orbital coefficients, corresponding to the four indices of (ijkl).
Kwargs:

- `erifile` *(str or h5py File or h5py Group object)* [The file/object] to store the transformed integrals. If not given, the return value is an array (in memory) of the transformed integrals.

- `dataname` *(str)* Note this argument is effective if erifile is given. The dataset name in the erifile (ref the hierarchy of HDF5 format [http://www.hdfgroup.org/HDF5/doc1.6/UG/09_Groups.html](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 specified dataname, the old integrals will be replaced by the new one under the key dataname.

- `intor` *(str)* [integral name Name of the 2-electron integral. Ref to getints_by_shell() for the complete list of available 2-electron integral names]

Returns: An array of transformed integrals if erifile is not given. Otherwise, return the file/fileobject if erifile is assigned.

Examples:

```python
>>> import pyscf
>>> mol = pyscf.M(atom='O 0 0 0; H 0 1 0; H 0 0 1', basis='sto3g')
>>> mol = numpy.random.random((mol.nao_nr(), 10))
>>> mo2 = numpy.random.random((mol.nao_nr(), 8))
>>> eri1 = mol.ao2mo(mo1)
>>> print(eri1.shape)
(55, 55)
>>> eri1 = mol.ao2mo(mo1, compact=False)
>>> print(eri1.shape)
(100, 100)
>>> eri1 = mol.ao2mo(eri, (mo1,mo2,mo2,mo2))
>>> print(eri1.shape)
(80, 36)
>>> eri1 = mol.ao2mo(eri, (mo1,mo2,mo2,mo2), erifile='water.h5')
```

- `ao_labels` *(mol, fmt=True, base=0)* Labels of 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`
  Offset of every shell in the spherical basis function spectrum

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

Examples:

```python
>>> 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]
```
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:

```python
>>> 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=None)
Offset of every shell in the spherical basis function spectrum

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

Examples:

```python
>>> 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:

```python
>>> 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, unit='Bohr')
Coordinates (ndarray) of the given atom id

**Args:**

- **atm_id** [int] 0-based

Examples:

```python
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1')
>>> mol.atom_coord(1)
[ 0.  0.  2.07869874]
```
atom_coords (unit='Bohr')
np.asarray([mol.atom_coords(i) for i in range(mol.natm)])

atom_mass_list (mol, isotope_avg=False)
A list of mass for all atoms in the molecule

Kwargs:
    isotope_avg [boolean] Whether to use the isotope average mass as the atomic mass

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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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]
 [ 4.11887044]
 [ 1.53971186]]
```

`bas_exp(bas_id)`
Exponents (ndarray) of the given shell

**Args:**
PySCF Documentation, Release 1.7.0

bas_id [int] 0-based

Examples:

```python
>>> 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:

```python
>>> 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_spnior(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:

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

bas_nprim(bas_id)

The number of primitive GTOs for the given shell

Args:

bas_id [int] 0-based

Examples:

```python
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1', basis='cc-pvdz')
>>> mol.bas_nprim(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=0,
symmetry=None, symmetry_subgroup=None, cart=None)

Setup molecule 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 molecular 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 to control multiplicity. If setting spin = None, multiplicity will be guessed based on the neutral molecule. 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 that transforms Cartesian GTOs to spherical GTOs for all basis functions

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:

```python
>>> 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 (mol, fmt=True, base=0)
Labels of 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 %-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.

copy (mol)
Deepcopy of the given Mole object

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

elements  A list of elements in the molecule
**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:**

```python
>>> 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_ao** *(mol, eval_name, coords, comp=None, shls_slice=None, non0tab=None, ao_loc=None, out=None)*

Evaluate AO function value on the given grids,

**Args:** eval_name : str

<table>
<thead>
<tr>
<th>Function</th>
<th>comp</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>“GTOval_sph”</td>
<td>1</td>
<td>IAO&gt;</td>
</tr>
<tr>
<td>“GTOval_ip_sph”</td>
<td>3</td>
<td>nabla</td>
</tr>
<tr>
<td>“GTOval_ig_sph”</td>
<td>3</td>
<td>(#C(0 1) g)</td>
</tr>
<tr>
<td>“GTOval_ipig_sph”</td>
<td>3</td>
<td>(#C(0 1) nabla g)</td>
</tr>
<tr>
<td>“GTOval_cart”</td>
<td>1</td>
<td>IAO&gt;</td>
</tr>
<tr>
<td>“GTOval_ip_cart”</td>
<td>3</td>
<td>nabla</td>
</tr>
<tr>
<td>“GTOval_ig_cart”</td>
<td>3</td>
<td>(#C(0 1) g)</td>
</tr>
</tbody>
</table>

**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:**

- **comp** [int] Number of the components of the operator
- **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 dft.gen_grid.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) to store AO values on grids.

**Examples:**

```python
>>> 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)
```
>>> print(ao_value.shape)
(3, 100, 24)

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

**Args:** `eval_name : str`

<table>
<thead>
<tr>
<th>Function</th>
<th>comp</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>“GTOval_sph”</td>
<td>1</td>
<td>IAO&gt;</td>
</tr>
<tr>
<td>“GTOval_ip_sph”</td>
<td>3</td>
<td>nabla IAO&gt;</td>
</tr>
<tr>
<td>“GTOval_sph”</td>
<td>3</td>
<td>(#C(0 1) g) IAO&gt;</td>
</tr>
<tr>
<td>“GTOval_ipig_sph”</td>
<td>3</td>
<td>(#C(0 1) nabla g) IAO&gt;</td>
</tr>
<tr>
<td>“GTOval_cart”</td>
<td>1</td>
<td>IAO&gt;</td>
</tr>
<tr>
<td>“GTOval_ip_cart”</td>
<td>3</td>
<td>nabla IAO&gt;</td>
</tr>
<tr>
<td>“GTOval_ig_cart”</td>
<td>3</td>
<td>(#C(0 1) g) IAO&gt;</td>
</tr>
</tbody>
</table>

**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:**

- **comp** [int] Number of the components of the operator
- **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 `dft.gen_grid.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) to store AO values on grids.

**Examples:**

```python
>>> 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)
>>> print(ao_value.shape)
(3, 100, 24)
```

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

\[ e = e^{-\alpha \cdot \beta^{i-1}} \text{ for } i = 1 \ldots n \]

**Args:**

- **l** [int] Angular momentum
n  [int]  Number of GTOs

Returns: Formated basis

Examples:

```python
>>> 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:

```python
>>> 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; ""t"" and arbitrary length of blank space to separate 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), new coordinates
- **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. The internal format is

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

Examples:

```python
>>> 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, (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:

```python
>>> gto.format_basis({'H':'sto-3g', 'H^2': '3-21g'})
{'H': [[0, [3.4252509099999999, 0.15432897000000001],
           [0.6239137300000006, 0.5353281399999995],
           [0.1688553999999999, 0.44463454000000002]],
    'H^2': [[0, [5.4471780000000001, 0.15628500000000001],
             [0.82454700000000003, 0.90469100000000002]],
             [0, [0, 0.18319199999999999, 1.0]]])
```

format_ecp(ecp_tab)

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

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

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

   (((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 ...)))},

   ...

fromfile(filename, format=None)

Update the Mole object based on the input geometry file

fromstring(string, format='xyz')

Update the Mole object based on the input geometry string

gto_norm(l, expnt)

Normalized factor for GTO radial part \( g = r^l e^{-a r^2} \)

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


Args:

l (int): angular momentum

expnt : exponent \( \alpha \)

Returns: normalization factor
Examples:

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

`has_ecp()`
Whether pseudo potential is used in the system.

`intor (intor, comp=None, 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:

```python
>>> 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. ]]]
[[ 0.10289944 0.48176097]
 [ -0.48176097 -0.10289944]]
```

`intor_asymmetric (intor, comp=None)`
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('intle_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=None)*

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

**Args:**

- **intor** [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=None)*

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=0, symmetry=None, symmetry_subgroup=None, cart=None)*

Setup molecule 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 molecular 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 to control multiplicity. If setting spin = None, multiplicity will be guessed based on the neutral molecule. 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.

**make_atm_env** *(atom, ptr=0, nucmod=1, nucprop=None)*
Convert the internal format Mole._atom to the format required by libcint integrals

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

**make_ecp_env** *(_atm, _ecp, pre_env=[])*
Generate the input arguments _ecpbas for ECP integrals

**make_env** *(atoms, basis, pre_env=[], nucmod={}, nucprop=None)*
Generate the input arguments for libcint library based on internal format Mole._atom and Mole._basis

**ms**
Spin quantum number. multiplicity = ms*2+1

**nao**
Total number of contracted GTOs for the given 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:**

```python
>>> 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=None)*

Total number of contracted 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:**

```python
>>> 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=None)*

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:

```python
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='ccpvtz')
>>> mol.search_ao_label('Cl.*p')
[19 20 21 22 23 24 25 26 27 28 29 30]
>>> mol.search_ao_label('Cl 2p')
[19 20 21]
>>> mol.search_ao_label(['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:

```python
>>> 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_origin`\((coord)\)

Update common origin for integrals of dipole, rxp etc. Note the unit of the coordinates needs to be Bohr

Examples:

```python
>>> mol.set_common_origin(0)
>>> mol.set_common_origin((1,0,0))
```

`set_common_origin_`\((coord)\)

Update common origin for integrals of dipole, rxp etc. Note the unit of the coordinates needs to be Bohr

Examples:

```python
>>> mol.set_common_origin(0)
>>> mol.set_common_origin((1,0,0))
```
```python
>>> mol.set_common_origin(0)
>>> mol.set_common_origin((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_or_coords, unit=None, symmetry=None, inplace=True)*  
Update geometry

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

Examples:

```python
>>> 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: \( \rho(r) = nuc\_charge \times \text{Norm} \times \exp(-zeta \times r^2) \). This function can **only** be called after .build() method is executed.

Examples:

```python
>>> 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 \( \text{erf}(\omega r_{12})/r_{12} \) set omega to 0 to switch off the range-separated Coulomb

**set_range_coulomb_** *(omega)*  
Apply the long range part of range-separated Coulomb operator for all 2e integrals \( \text{erf}(\omega r_{12})/r_{12} \) set omega to 0 to switch off the range-separated Coulomb

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

Examples:

```python
>>> mol.set_rinv_origin(0)
>>> mol.set_rinv_origin((0,1,0))
```

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

Examples:

```python
>>> mol.set_rinv_origin(0)
>>> mol.set_rinv_origin((0,1,0))
```

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

Examples:

```python
>>> mol.set_rinv_origin(0)
>>> mol.set_rinv_origin((0,1,0))
```
set_rinv_origin_(coord)
Update origin for operator \( \frac{1}{r-\mathcal{O}} \). **Note** the unit is Bohr

Examples:

```python
>>> mol.set_rinv_origin(0)
>>> mol.set_rinv_origin((0,1,0))
```

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

sph2spinor_coeff()
Transformation matrix that transforms real-spherical GTOs to spinor GTOs for all basis functions

Examples:

```python
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1', basis='ccpvtz')
>>> ca, cb = mol.sph2spinor_coeff()
>>> s0 = mol.intor('int1e_ovlp_spinor')
>>> s1 = ca.conj().T.dot(mol.intor('int1e_ovlp_sph')).dot(ca)
>>> s1+= cb.conj().T.dot(mol.intor('int1e_ovlp_sph')).dot(cb)
>>> print(abs(s1-s0).max())
```

sph_labels (mol, fmt=True, base=0)
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:

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

spheric_labels (mol, fmt=True, base=0)
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.sph_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')]}

spinor_labels (mol, fmt=True, base=0)
Labels of spinor GTO functions

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 $T|\psi_i\rangle = -|\psi_j\rangle$,
otherwise $T|\psi_i\rangle = |\psi_j\rangle$

tmap (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 $T|\psi_i\rangle = -|\psi_j\rangle$,
otherwise $T|\psi_i\rangle = |\psi_j\rangle$

to_uncontracted_cartesian_basis (mol)
Decontract the basis of a Mole or a Cell. Returns a Mole (Cell) object with the uncontracted basis en-
vironment and a list of coefficients that transform the uncontracted cartesian basis to the original basis.
Each element in the list corresponds to one shell of the original Mole (Cell).

Examples:

```python
>>> mol = gto.M(atom='Ne', basis='ccpvdz')
>>> pmol, ctr_coeff = mol.to_uncontracted_cartesian_basis()
>>> c = scipy.linalg.block_diag(*ctr_coeff)
>>> s = reduce(numpy.dot, (c.T, pmol.intor('int1e_ovlp'), c))
>>> abs(s-mol.intor('int1e_ovlp')).max()
0.0
```

tofile (mol, filename, format=None)
Write molecular geometry to a file of the required format.

Supported output formats:

- raw: Each line is <symobl> <x> <y> <z>
- xyz: XYZ cartesian coordinates format
- zmat: Z-matrix format

tostring (mol, format='raw')
Convert molecular geometry to a string of the required format.

Supported output formats:

- raw: Each line is <symobl> <x> <y> <z>
- xyz: XYZ cartesian coordinates format
- zmat: Z-matrix format

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

Returns: electron number in integer

Examples:
```python
>>> 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.

**with_common_origin** *(coord)*

Return a temporary mol context which has the required common origin. The required common origin has no effects out of the temporary context. See also `mol.set_common_origin()`

Examples:

```python
>>> with mol.with_common_origin((1,0,0)):
...   mol.intor('int1e_r', comp=3)
```

**with_common_origin** *(coord)*

Return a temporary mol context which has the required common origin. The required common origin has no effects out of the temporary context. See also `mol.set_common_origin()`

Examples:

```python
>>> with mol.with_common_origin((1,0,0)):
...   mol.intor('int1e_r', comp=3)
```

**with_range_coulomb** *(omega)*

Return a temporary mol context which has the required parameter omega for long range part of range-separated Coulomb operator. If omega = None, it will be treated as the regular Coulomb operator. See also `mol.set_range_coulomb()`

Examples:

```python
>>> with mol.with_range_coulomb(omega=1.5):
...   mol.intor('int2e')
```

**with_rinv_as_nucleus** *(atm_id)*

Return a temporary mol context in which the rinv operator (1/r) is treated like the Coulomb potential of a Gaussian charge distribution \( \rho(r) = \text{Norm} \times \exp(-\zeta \times r^2) \) at the place of the input atm_id.

Examples:

```python
>>> with mol.with_rinv_as_nucleus(3):
...   mol.intor('int1e_rinv')
```

**with_rinv_origin** *(coord)*

Return a temporary mol context which has the required origin of 1/r operator. The required origin has no effects out of the temporary context. See also `mol.set_rinv_origin()`

Examples:

```python
>>> with mol.with_rinv_origin((1,0,0)):
...   mol.intor('int1e_rinv')
```

**with_rinv_origin** *(coord)*

Return a temporary mol context which has the required origin of 1/r operator. The required origin has no effects out of the temporary context. See also `mol.set_rinv_origin()`

Examples:
with_rinv_zeta(zeta)

Return a temporary mol context which has the required Gaussian charge distribution placed at "rinv_origin": \( \rho(r) = \text{Norm} \times \exp(-zeta \times r^2) \). See also mol.set_rinv_zeta()

Examples:

```python
>>> with mol.with_rinv_zeta(zeta=1.5), mol.with_rinv_origin((1.,0,0)):
...   mol.intor('int1e_rinv')
```

pyscf.gto.mole.ao_labels(mol, fmt=True, base=0)

Labels of 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:

```python
>>> 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=None)

Offset of every shell in the spherical basis function spectrum

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

Examples:

```python
>>> 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_mass_list(mol, isotope_avg=False)

A list of mass for all atoms in the molecule

Kwargs:

isotope_avg [boolean] Whether to use the isotope average mass as the atomic mass

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

symmetry inequivalent atoms

pyscf.gto.mole.cart2j_kappa(kappa, l=None, normalized=None)

Cartesian to spinor transformation matrix for kappa
Kwargs:

- `normalized`: How the Cartesian GTOs are normalized. ‘sp’ means the s and p functions are normalized.

```python
pyscf.gto.mole.cart2j_l(l, normalized=None)
Cartesian to spinor transformation matrix for angular moment l
```

```python
pyscf.gto.mole.cart2sph(l, c_tensor=None, normalized=None)
Cartesian to real spherical transformation matrix
```

Kwargs:

- `normalized`: How the Cartesian GTOs are normalized. ‘sp’ means the s and p functions are normalized.

```python
pyscf.gto.mole.cart2spinor_kappa(kappa, l=None, normalized=None)
Cartesian to spinor transformation matrix for kappa
```

Kwargs:

- `normalized`: How the Cartesian GTOs are normalized. ‘sp’ means the s and p functions are normalized.

```python
pyscf.gto.mole.cart2spinor_l(l, normalized=None)
Cartesian to spinor transformation matrix for angular moment l
```

```python
pyscf.gto.mole.cart2zmat(coord)
```

```python
>> 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 4.2255607338457 2 50.7684795164077
1 2.90305235726773 2 79.3904651036893 3 6.20854462618583
```

```python
pyscf.gto.mole.cart_labels(mol, fmt=True, base=0)
Labels of 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”

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

```python
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

\[
(\mu|\nu), \mu \in mol1, \nu \in mol2
\]

Returns: Concatenated atm, bas, env

Examples: Compute the overlap between H2 molecule and O atom

```python
>>> 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,
```
.. code-block::

```python
>>> gto.moleintor.getints('int1e_ovlp_sph', atm3, bas3, env3, range(2),
                            range(2,5))
[[ 0.44714688 0.  0.37820346 0. ]
 [ 0.44714688 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(nuc_charge, nucprop={})
Generate the nuclear charge distribution parameter \( \zeta(r) = nuc\text{\_charge} \times \text{Norm} \times \exp(-\zeta r^2) \)

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:

```python
>>> 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:

\[ e = e^{-\alpha \cdot \beta^{i-1}} \quad \text{for} \quad i = 1 \ldots n \]

Args:

- l [int] Angular momentum
- n [int] Number of GTOs

Returns: Formated basis

Examples:

```python
>>> 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:

```python
>>> 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\._fakemol\_for\_charges \((coords, expnt=1e+16)\)

Construct a fake Mole object that holds the charges on the given coordinates (coords). The shape of the charge can be a normal distribution with the Gaussian exponent (expnt).

pyscf.gto.mole\._filatov\_nuc\_mod \((nuc\_charge, nucprop={})\)

Generate the nuclear charge distribution parameter \( zeta \rho(r) = nuc\_charge \times Normal\times \exp(-zeta \times r^2) \)


pyscf.gto.mole\._format\_atom \((atoms, origin=0, axes=None, unit='angstrom')\)

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 separate 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] \((\text{new}_x, \text{new}_y, \text{new}_z)\), new coordinates
- **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.** The internal format is

  \[
  \text{atom} = [(\text{atom1}, (x, y, z)),
  (\text{atom2}, (x, y, z)),
  ...
  (\text{atomN}, (x, y, z))]
  \]

Examples:

```python
>>> gto.format_atom('9,0,0,0; h8l 0 0 1', origin=(1,1,1))
[['F', [-1.0, -1.0, -1.0], ['H8l', [-1.0, -1.0, 0.0]]])
```

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:

```python
>>> gto.format_basis({'H': 'sto-3g', 'H^2': '3-21g'})
{'H': [[0,
    [3.4252509999999999, 0.15432897000000001],
    [0.62391373000000006, 0.53532813999999995],
    [0.16885399999999999, 0.44463454000000002]],
'H^2': [[0,
    [5.4471780000000001, 0.15628500000000001],
    [0.82454700000000003, 0.90469100000000002]],
    [0, [0.18319199999999999, 1.0]]])
```
pyscf.gto.mole.fromstring(string, format='xyz')
Convert the string of the specified format to internal format (in testing)

Supported formats:
- **raw**: Each line is <symbol> <x> <y> <z>
- **xyz**: XYZ cartesian coordinates format
- **zmat**: Z-matrix format

pyscf.gto.mole.gaussian_int(n, alpha)
int_0^inf x^n exp(-alpha x^2) dx

pyscf.gto.mole.gto_norm(l, expnt)
Normalized factor for GTO radial part g = r^l e^{-αr^2}

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


Args:
- **l** (int): angular momentum
- **expnt** : exponent α

Returns: normalization factor

Examples:

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

pyscf.gto.mole.inter_distance(mol, coords=None)
Inter-particle distance array

pyscf.gto.mole.intor_cross(intor, mol1, mol2, comp=None)
1-electron integrals from two molecules like \( \langle \mu | \text{intor} | \nu \rangle \), \(\mu \in \text{mol1}, \nu \in \text{mol2}\)

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

```python
>>> mol1 = gto.M(atom='H 0 0 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  0.37820346  0. ]
 [ 0.04875181  0.44714688  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, nuclear_model=1, nucprop={})

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_ecp_env(mol, _atm, ecp, pre_env=[])  
Generate the input arguments _ecpbas for ECP integrals

pyscf.gto.mole.make_env(  
    atoms, basis, pre_env=[], nucmod={}, nucprop={})

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:**

```python
>>> 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=None)

Total number of contracted 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:

```python
>>> 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=None)
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_basis_set(mol1, mol2)
Check whether two molecules use the same basis sets. The two molecules can have different geometry.

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:

```python
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='ccpvtz')
>>> gto.search_ao_label('Cl.*p')
[19 20 21 22 23 24 25 26 27 28 29 30]
>>> gto.search_ao_label('Cl 4p')
[19 20 21]
>>> gto.search_ao_label(['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

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

Examples:

```python
>>> 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:

```python
>>> 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.sph2spinor_kappa** (**kappa**, **l=None**)  
Real spherical to spinor transformation matrix for kappa

**pyscf.gto.mole.sph2spinor_l** (**l**)  
Real spherical to spinor transformation matrix for angular moment l

**pyscf.gto.mole.sph_labels** (**mol**, **fmt=True**, **base=0**)  
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:

```python
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='sto-3g')
>>> gto.sph_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.spheric_labels** (**mol**, **fmt=True**, **base=0**)  
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:

```python
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='sto-3g')
>>> gto.sph_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, base=0)
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:

```python
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='sto-3g')
>>> gto.sph_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.spinor_labels(mol, fmt=True, base=0)
Labels of spinor GTO functions

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

pyscf.gto.mole.to_uncontracted_cartesian_basis(mol)
Decontract the basis of a Mole or a Cell. Returns a Mole (Cell) object with the uncontracted basis environment and a list of coefficients that transform the uncontracted cartesian basis to the original basis. Each element in the list corresponds to one shell of the original Mole (Cell).

Examples:

```python
>>> mol = gto.M(atom='Ne', basis='ccpvdz')
>>> pmol, ctr_coeff = mol.to_uncontracted_cartesian_basis()
>>> c = scipy.linalg.block_diag(*ctr_coeff)
>>> s = reduce(numpy.dot, (c.T, pmol.intor('int1e_ovlp'), c))
>>> abs(s-mol.intor('int1e_ovlp')).max()
0.0
```

```
pyscf.gto.mole.tofile(mol, filename, format=None)
Write molecular geometry to a file of the required format.

Supported output formats:
raw: Each line is <symbol> <x> <y> <z>
xyz: XYZ cartesian coordinates format
zmat: Z-matrix format

pyscf.gto.mole.tostring(mol, format='raw')
Convert molecular geometry to a string of the required format.

Supported output formats:
raw: Each line is <symbol> <x> <y> <z>
xyz: XYZ cartesian coordinates format
zmat: Z-matrix format

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 0 0; C 0 0 1', charge=1)
>>> mol.tot_electrons()
6

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

Examples:
>>> gto.uncontract(gto.load('sto3g', 'He'))
[[0, [6.36242139, 1]], [0, [1.158923, 1]], [0, [0.31364979, 1]]]

pyscf.gto.mole.uncontracted_basis(_basis)
Uncontract internal format _basis

Examples:
>>> gto.uncontract(gto.load('sto3g', 'He'))
[[0, [6.36242139, 1]], [0, [1.158923, 1]], [0, [0.31364979, 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)

Example:
>>> 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])]
```python
>>> a = """"H
H 1 2.67247631453057
H 1 4.22555607338457 2 50.7684795164077
H 1 2.90305235726773 2 79.3904651036893 3 6.2085462618583"
""
>>> 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 or None] 2S, num. alpha electrons - num. beta electrons to control multiplicity. If spin = None is set, multiplicity will be guessed based on the neutral molecule.
- `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 molecular 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 or [function(nuc_charge, nucprop) => zeta]] Nuclear model. 0 or None means point nuclear model. Other values will enable Gaussian nuclear model. If a function is assigned to this attribute, the function will be called to generate the nuclear charge distribution value “zeta” and the relevant nuclear model will be set to Gaussian model. Default is point nuclear model.
- `nucprop` [dict] Nuclear properties (like g-factor ‘g’, quadrupole moments ‘Q’). It is needed by pyscf.prop module and submodules.
- `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
- `topgroup` [str] Point group of the system.
**groupname** [str] The supported subgroup of the point group. It can be one of Dooh, Coov, 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 is to ensure certain functions being initialized only 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 representes one atom

**natm** : number of atoms

**_bas** : [[atom-id,angular-momentum,num-primitive-GTO,num-contracted-GTO,0,ptr-of-exp,ptr-of-contract-coeff,0],...]] each element representes one shell

**nbas** : number of shells

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

Examples:

```python
>>> 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
```

**ao2mo** *(mo_coeffs, erifile=None, dataname='eri_mo', intor='int2e', **kwargs)*

Integral transformation for arbitrary orbitals and arbitrary integrals. See more detailed documentation in func:ao2mo.kernel.

**Args:**

**mo_coeffs** *(an np array or a list of arrays)* [A matrix of orbital] coefficients if it is a numpy ndarray, or four sets of orbital coefficients, corresponding to the four indices of (ijkl).

**Kwargs:**
erifile (str or h5py File or h5py Group object)  [The file/object] to store the transformed integrals. If not given, the return value is an array (in memory) of the transformed integrals.

dataname [str] Note this argument is effective if erifile is given. 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 specified dataname, the old integrals will be replaced by the new one under the key dataname.

intor (str) [integral name Name of the 2-electron integral. Ref] to getints_by_shell() for the complete list of available 2-electron integral names

Returns: An array of transformed integrals if erifile is not given. Otherwise, return the file/fileobject if erifile is assigned.

Examples:

```python
def runula():
    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))
    eri1 = mol.ao2mo(mo1)
    print(eri1.shape)
    # (55, 55)
    eri1 = mol.ao2mo(mo1, compact=False)
    print(eri1.shape)
    # (100, 100)
    eri1 = mol.ao2mo(eri, (mo1,mo2,mo2,mo2))
    print(eri1.shape)
    # (80, 36)
    eri1 = mol.ao2mo(eri, (mo1,mo2,mo2,mo2), erifile='water.h5')
```
**Returns:** list, each entry is the corresponding start id of spinor function

Examples:

```python
>>> 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=None)*

Offset of every shell in the spherical basis function spectrum

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

Examples:

```python
>>> 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:

```python
>>> 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, unit='Bohr')*

Coordinates (ndarray) of the given atom id

**Args:**

- **atm_id** [int] 0-based

Examples:

```python
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1')
>>> mol.atom_coord(1)
[ 0. 0. 2.07869874]
```
atom_coords (unit='Bohr')
np.asarray([mol.atom_coords(i) for i in range(mol.natm)])

atom_mass_list (mol, isotope_avg=False)
A list of mass for all atoms in the molecule

Kwargs:
   isotope_avg  [boolean] Whether to use the isotope average mass as the atomic mass

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:

```python
>>> 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:

```python
>>> mol.build(atom='H\(^2\) 0 0 0; H 0 0 1.1')
>>> mol.atom_pure_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:

```python
>>> 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:

```python
>>> 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:
```python
>>> 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 \(1 < j \leq l\), \(-l-1\), else \(l\)) of the given shell

**Args:**
- **bas_id** [int] 0-based

**Examples:**
```python
>>> 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:**
```python
>>> 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:**
```python
>>> 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=0, symmetry=None, symmetry_subgroup=None, cart=None)*
Setup molecule 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 molecular 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 to control multiplicity. If setting spin = None , multiplicity will be guessed based on the neutral molecule. 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 that transforms Cartesian GTOs to spherical GTOs for all basis functions
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:

```python
>>> 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(mol, fmt=True, base=0)
Labels of 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.
copy(mol)
Deepcopy of the given Mole object
dumps(mol)
Serialize Mole object to a JSON formatted str.
elements
A list of elements in the molecule
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:**

```python
gto.expand_etbs([(0, 2, 1.5, 2.), (1, 2, 1, 2.)])
```

```python
[[0, [6.0, 1]], [0, [3.0, 1]], [1, [1., 1]], [1, [2., 1]]]
```

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

**Args:** eval_name : str

<table>
<thead>
<tr>
<th>Function</th>
<th>comp</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>“GTOval_sph”</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>“GTOval_ip_sph”</td>
<td>3</td>
<td>nabla</td>
</tr>
<tr>
<td>“GTOval_ig_sph”</td>
<td>3</td>
<td>(#C(0 1) g)</td>
</tr>
<tr>
<td>“GTOval_ipig_sph”</td>
<td>3</td>
<td>(#C(0 1) nabla g)</td>
</tr>
<tr>
<td>“GTOval_cart”</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>“GTOval_ip_cart”</td>
<td>3</td>
<td>nabla</td>
</tr>
<tr>
<td>“GTOval_ig_cart”</td>
<td>3</td>
<td>(#C(0 1) g)</td>
</tr>
</tbody>
</table>

**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:**

**comp** [int] Number of the components of the operator

**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 `dft.gen_grid.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) to store AO values on grids.

**Examples:**

```python
mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz')
cords = 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)
print(ao_value.shape)
(3, 100, 24)
```
**eval_gto** *(mol, eval_name, coords, comp=None, shls_slice=None, non0tab=None, ao_loc=None, out=None)*

Evaluate AO function value on the given grids.

**Args:**

- **eval_name**: str

<table>
<thead>
<tr>
<th>Function</th>
<th>comp</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GTOval_sph&quot;</td>
<td>1</td>
<td>AO&gt;</td>
</tr>
<tr>
<td>&quot;GTOval_ip_sph&quot;</td>
<td>3</td>
<td>nabla AO&gt;</td>
</tr>
<tr>
<td>&quot;GTOval_ig_sph&quot;</td>
<td>3</td>
<td>(#C(0 1) g) AO&gt;</td>
</tr>
<tr>
<td>&quot;GTOval_ipig_sph&quot;</td>
<td>3</td>
<td>(#C(0 1) nabla g) AO&gt;</td>
</tr>
<tr>
<td>&quot;GTOval_cart&quot;</td>
<td>1</td>
<td>AO&gt;</td>
</tr>
<tr>
<td>&quot;GTOval_ip_cart&quot;</td>
<td>3</td>
<td>nabla AO&gt;</td>
</tr>
<tr>
<td>&quot;GTOval_ig_cart&quot;</td>
<td>3</td>
<td>(#C(0 1) g) AO&gt;</td>
</tr>
</tbody>
</table>

**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:**

- **comp** [int] Number of the components of the operator
- **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 dft.gen_grid.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) to store AO values on grids.

**Examples:**

```python
tmp = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz')
tmp = 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 = tmp.eval_gto("GTOval_sph", coords)
print(ao_value.shape)
(100, 24)
ao_value = mol.eval_gto("GTOval_ig_sph", coords)
print(ao_value.shape)
(3, 100, 24)
```
Examples:

```python
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:

```python
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 separate 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), new coordinates
- `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. The internal format is:

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

Examples:

```python
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, ..))], ... ]```

1.7. gto — Molecular structure and GTO basis
Args:

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

Returns: Formated basis

Examples:

```python
gto.format_basis({'H':'sto-3g', 'H^2': '3-21g'})
{'H': [[0, [3.4252509999999999, 0.15432897000000001],
       [0.62391373000000006, 0.53532813999999995],
       [0.16885539999999999, 0.44634540000000002]],
    '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:

```python
{ atom: (nelec,  # core electrons
    ...)
   ...}
```

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

```python
((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 ...))))},
```

fromfile(filename, format=None)
Update the Mole object based on the input geometry file

fromstring(string, format='xyz')
Update the Mole object based on the input geometry string

`gto_norm`(l, expnt)
Normalized factor for GTO radial part $g = r^l e^{-\alpha r^2}$

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


Args:

l (int): angular momentum

expnt : exponent $\alpha$

Returns: normalization factor

Examples:

```python
>>> print(gto_norm(0, 1))
2.5264751109842591
```
has_ecp()
Whether pseudo potential is used in the system.

intor (intor, comp=None, 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:

```python
>>> 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. 0.10289944 0.48176097]]
[-0.48176097 -0.10289944]]
```

intor_asymmetric (intor, comp=None)
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:

```python
>>> 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=None)
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

```python
>>> 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=None)
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:

```python
>>> 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=0, symmetry=None, symmetry_subgroup=None, cart=None)
Setup molecule 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] Allowed memory in MB. If given, overwrite Mole.max_memory
atom [list or str] To define molecular 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 to control multiplicity. If setting spin = None, multiplicity will be guessed based on the neutral molecule. 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.

make_atm_env (atom, ptr=0, nucmod=1, nucprop=None)
Convert the internal format Mole._atom to the format required by libcint integrals

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

make_ecp_env (_atm, _ecp, pre_env=[])
Generate the input arguments _ecpbas for ECP integrals

make_env (atoms, basis, pre_env=[], nucmod={}, nucprop=None)
Generate the input arguments for libcint library based on internal format Mole._atom and Mole._basis

ms
Spin quantum number. multiplicity = ms*2+1

nao
Total number of contracted GTOs for the given 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:
```python
>>> 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=None)*
Total number of contracted 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:**
```python
>>> 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=None)*
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:**
```python
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='ccpvtz')
>>> mol.search_ao_label('Cl.*p')
[19 20 21 22 23 24 25 26 27 28 29 30]
>>> mol.search_ao_label('Cl 2p')
[19 20 21]
>>> mol.search_ao_label(['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:**

```python
>>> 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_origin** (*coord*)

Update common origin for integrals of dipole, rxp etc. **Note** the unit of the coordinates needs to be Bohr

**Examples:**

```python
>>> mol.set_common_origin(0)
>>> mol.set_common_origin((1,0,0))
```

**set_common_origin_** (*coord*)

Update common origin for integrals of dipole, rxp etc. **Note** the unit of the coordinates needs to be Bohr

**Examples:**

```python
>>> mol.set_common_origin(0)
>>> mol.set_common_origin((1,0,0))
```

**set_common_origin** (*coord*)

Update common origin for integrals of dipole, rxp etc. **Note** the unit of the coordinates needs to be Bohr

**Examples:**

```python
>>> mol.set_common_origin(0)
>>> mol.set_common_origin((1,0,0))
```

**set_common_origin_** (*coord*)

Update common origin for integrals of dipole, rxp etc. **Note** the unit of the coordinates needs to be Bohr

**Examples:**

```python
>>> mol.set_common_origin(0)
>>> mol.set_common_origin((1,0,0))
```

1.7. gto — Molecular structure and GTO basis
set_f12_zeta (zeta)
Set zeta for YP exp(-zeta r12)/r12 or STG exp(-zeta r12) type integrals

set_geom_ (atoms_or_coords, unit=None, symmetry=None, inplace=True)
Update 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:

```python
>>> 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:

```python
>>> 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 switch 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 switch off the range-separated Coulomb

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

```python
>>> mol.set_rinv_origin(0)
>>> mol.set_rinv_origin((0,1,0))
```

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

```python
>>> mol.set_rinv_origin(0)
>>> mol.set_rinv_origin((0,1,0))
```

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

```python
>>> mol.set_rinv_origin(0)
>>> mol.set_rinv_origin((0,1,0))
```

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

```python
>>> mol.set_rinv_origin(0)
>>> mol.set_rinv_origin((0,1,0))
```

**set_rinv_zeta**(zn)
Assume the charge distribution on the “rinv_origin”. zeta is the parameter to control the charge distribution: \(\rho(r) = \text{Norm} \times \exp(-zeta \times r^2)\). **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!

**sph2spinor_coeff**
Transformation matrix that transforms real-spherical GTOs to spinor GTOs for all basis functions

Examples:

```python
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1', basis='ccpvtz')
>>> ca, cb = mol.sph2spinor_coeff()
>>> s0 = mol.intor('int1e_ovlp_spinor')
>>> s1 = ca.conj().T.dot(mol.intor('int1e_ovlp_sph')).dot(ca)
>>> s1+= cb.conj().T.dot(mol.intor('int1e_ovlp_sph')).dot(cb)
>>> print(abs(s1-s0).max())
>>> 6.66133814775e-16
```

**sph_labels**(mol, fmt=True, base=0)
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:

```python
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='sto-3g')
>>> gto.sph_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'])
```

**spheric_labels**(mol, fmt=True, base=0)
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:

```python
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='sto-3g')
>>> gto.sph_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'])
```
spinor_labels (mol, fmt=True, base=0)
Labels of spinor GTO functions

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

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

to_uncontracted_cartesian_basis (mol)
Decontract the basis of a Mole or a Cell. Returns a Mole (Cell) object with the uncontracted basis environment and a list of coefficients that transform the uncontracted cartesian basis to the original basis. Each element in the list corresponds to one shell of the original Mole (Cell).

Examples:
```python
>>> mol = gto.M(atom='Ne', basis='ccpvdz')
>>> pmol, ctr_coeff = mol.to_uncontracted_cartesian_basis()
>>> c = scipy.linalg.block_diag(*ctr_coeff)
>>> s = reduce(numpy.dot, (c.T, pmol.intor('int1e_ovlp'), c))
>>> abs(s-mol.intor('int1e_ovlp')).max()
0.0
```

tofile (mol, filename, format=None)
Write molecular geometry to a file of the required format.

Supported output formats:
- raw: Each line is <symbol> <x> <y> <z>
- xyz: XYZ cartesian coordinates format
- zmat: Z-matrix format

tostring (mol, format='raw')
Convert molecular geometry to a string of the required format.

Supported output formats:
- raw: Each line is <symbol> <x> <y> <z>
- xyz: XYZ cartesian coordinates format
- zmat: Z-matrix format

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

Returns: electron number in integer

Examples:
```python
>>> 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.

**with_common_orig** *(coord)*
Retuen a temporary mol context which has the required common origin. The required common origin has no effects out of the temporary context. See also *mol.set_common_origin()*

Examples:

```python
>>> with mol.with_common_origin((1,0,0)):
...    mol.intor('int1e_r', comp=3)
```

**with_common_origin** *(coord)*
Retuen a temporary mol context which has the required common origin. The required common origin has no effects out of the temporary context. See also *mol.set_common_origin()*

Examples:

```python
>>> with mol.with_common_origin((1,0,0)):
...    mol.intor('int1e_r', comp=3)
```

**with_range_coulomb** *(omega)*
Retuen a temporary mol context which has the required parameter omega for long range part of range-separated Coulomb operator. If omega = None, it will be treated as the regular Coulomb operator. See also *mol.set_range_coulomb()*

Examples:

```python
>>> with mol.with_range_coulomb(omega=1.5):
...    mol.intor('int2e')
```

**with_rinv_as_nucleus** *(atm_id)*
Retuen a temporary mol context in which the rinv operator (1/r) is treated like the Coulomb potential of a Gaussian charge distribution \( \rho(r) = \text{Norm} \times \exp(-\zeta \times r^2) \) at the place of the input atm_id.

Examples:

```python
>>> with mol.with_rinv_as_nucleus(3):
...    mol.intor('int1e_rinv')
```

**with_rinv_orig** *(coord)*
Retuen a temporary mol context which has the required origin of 1/r operator. The required origin has no effects out of the temporary context. See also *mol.set_rinv_origin()*

Examples:

```python
>>> with mol.with_rinv_orig((1,0,0)):
...    mol.intor('int1e_rinv')
```

**with_rinv_origin** *(coord)*
Retuen a temporary mol context which has the required origin of 1/r operator. The required origin has no effects out of the temporary context. See also *mol.set_rinv_origin()*

Examples:

```python
>>> with mol.with_rinv_origin((1,0,0)):
...    mol.intor('int1e_rinv')
```
with_rinv_zeta(zeta)

Return a temporary mol context which has the required Gaussian charge distribution placed at
“rinv_origin”: \( \rho(r) = \text{Norm} \times \exp(-zeta \times r^2) \). See also mol.set_rinv_zeta()

Examples:

```python
>>> with mol.with_rinv_zeta(zeta=1.5), mol.with_rinv_origin((1.,0,0)):
...     mol.intor('intle_rinv')
```

moleintor

A low level interface to libcint library. It’s recommended to use the Mole.intor method to drive the integral evaluation
functions.

```python
pyscf.gto.moleintor.ascint3(intor_name)
```

caret cint2 function name to cint3 function name

```python
pyscf.gto.moleintor.getints(intor_name, atm, bas, env, shls_slice=None, comp=None, hermi=0, ao_sym='s1', ao_loc=None, cintopt=None, out=None)
```

1e and 2e integral generator.

**Args:** intor_name : str

<table>
<thead>
<tr>
<th>Function</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>'intle_ovlp'</td>
<td>(1)</td>
</tr>
<tr>
<td>'intle_nuc'</td>
<td>(1 nuc l)</td>
</tr>
<tr>
<td>'intle_kin'</td>
<td>(.5 l p dot p)</td>
</tr>
<tr>
<td>'intle_ia01p'</td>
<td>(#C(0 1) l nabla-rinv l cross p)</td>
</tr>
<tr>
<td>'intle_giao_irjxp'</td>
<td>(#C(0 1) l r cross p)</td>
</tr>
<tr>
<td>'intle_cg_irxp'</td>
<td>(#C(0 1) l rc cross p)</td>
</tr>
<tr>
<td>'intle_giao_allpart'</td>
<td>(-.5 l nabla-rinv l r)</td>
</tr>
<tr>
<td>'intle_cg_allpart'</td>
<td>(-.5 l nabla-rinv l rc)</td>
</tr>
<tr>
<td>'intle_a01gp'</td>
<td>(g l nabla-rinv cross pl)</td>
</tr>
<tr>
<td>'intle_igkin'</td>
<td>(#C(0.5) g l p dot p)</td>
</tr>
<tr>
<td>'intle_igovlp'</td>
<td>(#C(0 1) g l)</td>
</tr>
<tr>
<td>'intle_ignuc'</td>
<td>(#C(0 1) g l nuc l)</td>
</tr>
<tr>
<td>'intle_z'</td>
<td>(l zc l)</td>
</tr>
<tr>
<td>'intle_zz'</td>
<td>(l zc zc l)</td>
</tr>
<tr>
<td>'intle_r'</td>
<td>(l rc l)</td>
</tr>
<tr>
<td>'intle_r2'</td>
<td>(l rc dot rc l)</td>
</tr>
<tr>
<td>'intle_r'</td>
<td>(l rc l)</td>
</tr>
<tr>
<td>'intle_rrr'</td>
<td>(l rc rc rc l)</td>
</tr>
<tr>
<td>'intle_pnucp'</td>
<td>(p* l nuc dot pl)</td>
</tr>
<tr>
<td>'intle_prinvxp'</td>
<td>(p* l rinv cross pl)</td>
</tr>
<tr>
<td>'intle_ipovlp'</td>
<td>(nabla l)</td>
</tr>
<tr>
<td>'intle_ipkin'</td>
<td>(.5 nabla l p dot p)</td>
</tr>
<tr>
<td>'intle_ipnuc'</td>
<td>(nabla l nuc l)</td>
</tr>
<tr>
<td>'intle_iprinv'</td>
<td>(nabla l rinv l)</td>
</tr>
<tr>
<td>'intle_rinv'</td>
<td>(l rinv l)</td>
</tr>
<tr>
<td>'intle_pnucxp'</td>
<td>(p* l nuc cross pl)</td>
</tr>
<tr>
<td>'intle_irp'</td>
<td>(l rc nabla l)</td>
</tr>
<tr>
<td>'intle_irrp'</td>
<td>(l rc re nabla l)</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>“int1e_irpr”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_ggvlp”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_ggkin”</td>
<td>(.5</td>
</tr>
<tr>
<td>“int1e_ggnum”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_grjxp”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_ovlp_spinor”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_nucl_spinor”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_srspn_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_srsr_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_srsrnuc_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_spsp_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_govlp_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_gnum_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_cg_sa10sa01_checker”</td>
<td>(.5</td>
</tr>
<tr>
<td>“int1e_cg_sa10sp_checker”</td>
<td>(.5</td>
</tr>
<tr>
<td>“int1e_cg_sa10nuccsp_checker”</td>
<td>(.5</td>
</tr>
<tr>
<td>“int1e_giao_sa10sa01_checker”</td>
<td>(.5</td>
</tr>
<tr>
<td>“int1e_giao_sa10sp_checker”</td>
<td>(.5</td>
</tr>
<tr>
<td>“int1e_giao_sa10nuccsp_checker”</td>
<td>(.5</td>
</tr>
<tr>
<td>“int1e_sa01sp_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_spspg_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_spgnucsp_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_spgsa01_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_spsp_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_snucl_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_ipovlp_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_ipkin_checker”</td>
<td>(.5</td>
</tr>
<tr>
<td>“int1e_ipnuc_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_iprinv_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_ipsnuccsp_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int1e_ipsprinvsp_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int2e”</td>
<td>( , , )</td>
</tr>
<tr>
<td>“int2e_s_g1”</td>
<td>(#C(0</td>
</tr>
<tr>
<td>“int2e_s_gg1”</td>
<td>(</td>
</tr>
<tr>
<td>“int2e_s_glg2”</td>
<td>(</td>
</tr>
<tr>
<td>“int2e_p1vxp1”</td>
<td>(</td>
</tr>
<tr>
<td>“int2e_s_s”</td>
<td>( , l , )</td>
</tr>
<tr>
<td>“int2e_s_spsp1_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int2e_s_spsp1spsp2_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int2e_s_srsr1_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int2e_s_srsr1srsr2_checker”</td>
<td>(</td>
</tr>
<tr>
<td>“int2e_s_cg_sa10sp1_checker”</td>
<td>(.5</td>
</tr>
<tr>
<td>“int2e_giao_sa10sp1_checker”</td>
<td>(.5</td>
</tr>
<tr>
<td>“int2e_giao_sa10sp1spsp2_checker”</td>
<td>(.5</td>
</tr>
<tr>
<td>“int2e_g1_checker”</td>
<td>(</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>“int2e_spssp1_spinor”</td>
<td>$(g \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_g1ssp2_spinor”</td>
<td>$(g, l \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_spssp1ssp2_spinor”</td>
<td>$(g \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_spv1_spinor”</td>
<td>$(\sigma \cdot p, l)$</td>
</tr>
<tr>
<td>“int2e_vsp1_spinor”</td>
<td>$(l, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_spssp2_spinor”</td>
<td>$(l, \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_spv1spv2_spinor”</td>
<td>$(\sigma \cdot p, l \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_vsp1spv2_spinor”</td>
<td>$(\sigma \cdot p, l \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_spv1ssp2_spinor”</td>
<td>$(\sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_spv1vsp2_spinor”</td>
<td>$(\sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ip1”</td>
<td>$(\nabla, l)$</td>
</tr>
<tr>
<td>“int2e_ip1_spinor”</td>
<td>$(\nabla, l)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid l)$</td>
</tr>
<tr>
<td>“int2e_ipspsp2_spinor”</td>
<td>$(\nabla, l \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1ssp2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1spv2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1spsp2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1vspsp2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1vsv2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1ssp2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1vsv2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1spsp2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1vsv2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1ssp2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1spv2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1spsp2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1vsv2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1ssp2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1vsv2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1spsp2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1vsv2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1ssp2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1vsv2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1spsp2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1vsv2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1ssp2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1vsv2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1spsp2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
<tr>
<td>“int2e_ipspsp1vsv2_spinor”</td>
<td>$(\nabla \sigma \cdot p, \sigma \cdot p \mid \sigma \cdot p, \sigma \cdot p)$</td>
</tr>
</tbody>
</table>

**Kwargs:**

- **atm** [int32 ndarray] libcint integral function argument
- **bas** [int32 ndarray] libcint integral function argument
- **env** [float64 ndarray] libcint integral function argument

**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 (ijkl)
- ‘2kl’ or ‘s2kl’: symmetry between k, l in (ijkl)
- 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.       ]]]]
```
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:

```python
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, (...),
    ...],
    ...
```
**Args:** string : A string in NWChem basis format. Empty links and the lines of “BASIS SET” and “END” will be ignored

**Kwargs:**

- **optimize** [Optimize basis contraction. Convert the segment contracted] basis to the general contracted basis.

**Examples:**

```python
gto.basis.parse(""
... #BASIS SET: (6s,3p) -> [2s,1p]
... C    S
... 71.6168370 0.15432987
... 13.0450960 0.53532814
... 3.5305122 0.4463454
... C    SP
... 2.9412494 -0.09996723 0.15591627
... 0.6834831 0.39951283 0.60768372
... 0.2222899 0.70011547 0.39195739
... ")

>>> gto.basis.parse(""
... He    S
... 13.6267000 0.1752300
... 1.9993500 0.8934830
... 0.3829930 0.0000000
... He    S
... 13.6267000 0.0000000
... 1.9993500 0.0000000
... 0.3829930 1.0000000
... ")

[(0.0, 0.1752300, 0.0), (1.9993500, 0.8934830, 0.0), (0.3829930, 0.0, 1.0)]
```

### 1.8 ao2mo — Integral transformations

The **ao2mo** module implements transformations from AO to MO integrals for various permutational and spin symmetries. A simple example to perform integral transformation:

```python
mol = gto.Mole(atom='H 0 0 0; F 0 0 1.1', basis = 'ccpvdz')
myhf = scf.RHF(mol)
myhf.kernel()
orb = myhf.mo_coeff
eri_4fold = ao2mo.kernel(mol, orb)
```

#### 1.8.1 Examples

Relevant examples:
- examples/ao2mo/00-mo_integrals.py
- examples/ao2mo/01-outcore.py
- examples/ao2mo/10-diff_orbs_for_ijkl.py
- examples/ao2mo/11-ump2.py
- examples/ao2mo/20-eri_grad_hess.py
- examples/ao2mo/21-spin_orbit_coupling.py
1.8.2 Program reference

General Integral transformation module

Simple usage:

```python
define 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)
```

**incore**

`pyscf.ao2mo.incore.full` (*eri_a0*, *mo_coeff*, verbose=0, compact=True, **kwargs)

MO integral transformation for the given orbital.

**Args:**
- *eri_a0* [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:**

```python
>>> 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)
```

**incore**

`pyscf.ao2mo.incore.general` (*eri_a0*, *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_a0* [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 (ijkl)

**Kwargs:**
- *verbose* [int] Print level
compact [bool] When compact is True, depending on the four orbital 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:

```python
>>> 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,mo2,mo3,mo2))
>>> print(eri1.shape)  # (55, 36)
>>> eri1 = ao2mo.incore.general(eri, (mo1,mo2,mo3,mo2))
>>> print(eri1.shape)  # (80, 80)
```

pyscf.ao2mo.incore.half_el (eriAo, mo_coeffs, compact=True)

Given two set of orbitals, half transform the (ij pair of 8-fold or 4-fold AO integrals (ijkl)

Args:

    eriAo [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 (ijkl)

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 sym-
metry, depending on the given orbitals, and the kwargs compact.

Examples:

```python
>>> 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))
```
semi_incore

Created on Thu May 17 11:05:22 2018
@author: Bryan Lau

A module that will do on-disk transformation of two electron integrals, and also return specific slices of (o)ccupied and (v)irtual ones needed for post HF

Comparing to the full in-memory transformation (see incore.py) which holds all intermediates in memory, this version uses less memory but performs slow due to IO overhead.

```
pyscf.ao2mo.semi_incore.general(eri, mo_coeffs, erifile, dataname='eri_mo', ioblk_size=128, compact=True, verbose=3)
```

For the given four sets of orbitals, transfer arbitrary spherical AO integrals to MO integrals on disk. Args:

- **eri** : 8-fold reduced eri vector
- **mo_coeffs** : 4-item list of ndarray

Four sets of orbital coefficients, corresponding to the four indices of (ijkl)

- **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.

- **ioblk_size** [float or int] The block size for IO, large block size may not improve performance

- **compact** [bool] When compact is True, depending on the four orbitals 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

**Pseudocode / algorithm:**

```
u = mu v = nu l = lambda o = sigma
```

Assume eri’s are 8-fold reduced. nij/nkl_pair = npair or i*j/k*l if only transforming a subset

**First half transform:**

```
Initialize half_eri of size (nij_pair,npair)
```

```
For lo = 1 -> npair Unpack row lo Unpack row lo to matrix E_{uv}^[lo] Transform C_{ui}^*+E*C_{nj} -> E_{ij}^[lo] Ravel or pack E_{ij}^[lo] Save E_{ij}^[lo] -> half_eri[:,lo]
```

**Second half transform:**

```
Initialize h5d_eri of size (nij_pair,nkl_pair)
```
For $ij = 1 \rightarrow n_{ij\text{ pair}}$ Load and unpack half_eri[$ij$,] $\rightarrow E_{lo}^{ij}$ Transform $C_{lk}E_{lo}^{ij}C_{ol} \rightarrow E_{kl}^{ij}$ Repack $E_{kl}^{ij} \rightarrow$ h5d_eri[$ij$,]

Each matrix is indexed by the composite index $ij \times kl$, where $ij/kl$ is either $n_{pair}$ or $i_{xj}/k_{xl}$, if only a subset of MOs are being transformed. Since entire rows or columns need to be read in, the arrays are chunked such that $IOBLK\_SIZE = row/col \times$ chunking $col/row$. For example, for the first half transform, we would save in $n_{ij\text{ pair}} \times IOBLK\_SIZE/n_{ij\text{ pair}}$, then load in $IOBLK\_SIZE/n_{kl\text{ pair}} \times n_{pair}$ for the second half transform.

---

As a first guess, the chunking size is $jxl$. If the super-rows/cols are larger than $IOBLK\_SIZE$, then the chunk rectangle $jxl$ is trimmed accordingly. The pathological limiting case is where the dimensions $n_{ao\text{ pair}}$, $n_{ij\text{ pair}}$, or $n_{kl\text{ pair}}$ are so large that the arrays are chunked 1x1, in which case $IOBLK\_SIZE$ needs to be increased.

outcore

pyscf.ao2mo.outcore.full(mol, mo_coeff, erifile, dataname='eri_mo', intor='int2e', aosym='s4', comp=None, 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.

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 $(ijkl)$
‘2kl’ or ‘s2kl’ : symmetry between $k$, $l$ in $(ijkl)$
1 or ‘1’ or ‘s1’: no symmetry
‘a4ij’ : 4-fold symmetry with anti-symmetry between $i$, $j$ in $(ijkl)$ (TODO)
‘a4kl’ : 4-fold symmetry with anti-symmetry between $k$, $l$ in $(ijkl)$ (TODO)
‘a2ij’ : anti-symmetry between $i$, $j$ in $(ijkl)$ (TODO)
‘a2kl’ : anti-symmetry between $k$, $l$ in $(ijkl)$ (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 orbital 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:**

```python
>>> from pyscf import gto
>>> from pyscf import ao2mo
>>> import h5py
>>> def view(h5file, dataname='eri_mo'):
...     f5 = h5py.File(h5file, 'r')
...     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')
>>> 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', aosym='s4', comp=None, 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 wrapper 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-existent 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 (ijkl) with the same set of orbitals.
- **erifile** [str] To store the transformed integrals, in HDF5 format.

**Kwarg:**

- **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.
- **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 (ijkl)
‘2kl’ or ‘s2kl’: symmetry between k, l in (ijkl)
1 or ‘1’ or ‘s1’: no symmetry
‘a4ij’: 4-fold symmetry with anti-symmetry between i, j in (ijkl) (TODO)
‘a4kl’: 4-fold symmetry with anti-symmetry between k, l in (ijkl) (TODO)
‘a2ij’: anti-symmetry between i, j in (ijkl) (TODO)
‘a2kl’: anti-symmetry between k, l in (ijkl) (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 orbital 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:

```python
>>> 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')
>>> mol = numpy.random.random((mol.nao_nr(), 10))
>>> eri1 = ao2mo.outcore.full_iofree(mol, mol)
>>> print(eri1.shape)
(55, 55)
>>> eri1 = ao2mo.outcore.full_iofree(mol, mol, compact=False)
>>> print(eri1.shape)
(100, 100)
>>> eri1 = ao2mo.outcore.full_iofree(mol, mol, intor='int2e_ip1_sph', aosym='s1', comp=3)
>>> print(eri1.shape)
(3, 100, 100)
>>> eri1 = ao2mo.outcore.full_iofree(mol, mol, 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', intor='int2e', aosym='s4', comp=None, 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

1.8. ao2mo — Integral transformations 95
**mo_coeffs** [4-item list of ndarray] Four sets of orbital coefficients, corresponding to the four indices of \((ijlk)\).

**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.

**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 \((ijkl)\)
- ‘2kl’ or ‘s2kl’: symmetry between \(k, l\) in \((ijkl)\)
- 1 or ‘1’ or ‘s1’: no symmetry
- ‘a4ij’: 4-fold symmetry with anti-symmetry between \(i, j\) in \((ijkl)\) (TODO)
- ‘a4kl’: 4-fold symmetry with anti-symmetry between \(k, l\) in \((ijkl)\) (TODO)
- ‘a2ij’: anti-symmetry between \(i, j\) in \((ijkl)\) (TODO)
- ‘a2kl’: anti-symmetry between \(k, l\) in \((ijkl)\) (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 orbital 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:**

```python
>>> from pyscf import gto
>>> from pyscf import ao2mo
>>> import h5py
>>> def view(h5file, dataname='eri_mo'):
...     f5 = h5py.File(h5file, 'r')
...     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')
>>> mol = numpy.random.random((mol.nao_nr(), 10))
>>> ao2mo.outcore.general(mol, (mol, mo2, mo3, mo4), 'oh2.h5')
>>> view('oh2.h5')
dataset ['eri_mo'], shape (80, 24)
>>> ao2mo.outcore.general(mol, (mol, mo2, mo3, mo3), 'oh2.h5')
```
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 `(ijkl)`.

**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’ | 4-fold symmetry (default) |
  | ‘2ij’ or ‘s2ij’ | symmetry between i, j in (ijkl) |
  | ‘2kl’ or ‘s2kl’ | symmetry between k, l in (ijkl) |
  | 1 or ‘1’ | no symmetry |
  | ‘a4ij’ | 4-fold symmetry with anti-symmetry between i, j in (ijkl) (TODO) |
  | ‘a4kl’ | 4-fold symmetry with anti-symmetry between k, l in (ijkl) (TODO) |
  | ‘a2ij’ | anti-symmetry between i, j in (ijkl) (TODO) |
  | ‘a2kl’ | anti-symmetry between k, l in (ijkl) (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 orbital 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:

```python
>>> from pyscf import gto
>>> from pyscf import ao2mo
>>> import h5py

>>> def view(h5file, dataname='eri_mo'):
...     f5 = h5py.File(h5file, 'r')
...     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), compact=False)
>>> print(eri1.shape)
(80, 36)

>>> eri1 = ao2mo.outcore.general_iofree(mol, (mo1, mo2, mo3, mo4), intor='int2e_ip1_sph', aosym='s2kl', comp=3)
>>> print(eri1.shape)
(3, 100, 100)
```

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 (ijkl) 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.

Kwarg:

- **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 (ijkl)
'2kl' or 's2kl': symmetry between k, l in (ijkl)
1 or '1' or 's1': no symmetry
'a4ij': 4-fold symmetry with anti-symmetry between i, j in (ijkl) (TODO)
'a4kl': 4-fold symmetry with anti-symmetry between k, l in (ijkl) (TODO)
'a2ij': anti-symmetry between i, j in (ijkl) (TODO)
'a2kl': anti-symmetry between k, l in (ijkl) (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 orbital 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 performance

Returns: None

addons

class pyscf.ao2mo.addons.load (eri, dataname='eri_mo')
load 2e integrals from hdf5 file

Usage:
with load(eri_file) as eri: print(eri.shape)

pyscf.ao2mo.addons.restore (symmetry, eri, norb, tao=None)
Convert the 2e integrals (in Chemist's notation) 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 (ijkl) (TODO)
's2ij' or '2kl': symmetric kl pair for (ijkl) (TODO)

Note the 4-fold symmetry requires (ijkl) == (ijlk) == (ijkl) while (ijkl) != (klij) is not required.
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

1.8. ao2mo — Integral transformations
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:

```python
>>> 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')
>>> 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,)
```

1.9 df — Density fitting

1.9.1 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 \( (i j | k l) = V_{ij,x} V_{kl,x} \), 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:

```python
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:

```python
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.
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 density 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 density 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.psi4.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.efitbasis` attribute. The input format of auxiliary fitting basis is exactly the same to the input format of orbital `basis` set. For example:

```python
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'
```
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()`:

```python
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 $\zeta$ of the even-tempered auxiliary basis are

$$
\varphi = r^i \exp(-\zeta_{il} r^2), \quad i = 0..n \\
\zeta_{il} = \alpha \times \beta^i : label : etb
$$

The default value of $\beta$ is 2.3. $\alpha$ and the number of auxiliary basis $n$ is determined based on the orbital basis. Given the orbital basis

$$
\chi = r^i \exp(-\alpha_i r^2)
$$

the orbital pair on the same center produces a new one-center basis

$$
\chi \chi' = r^{i+i'} \exp(-(\alpha_i + \alpha_{i'}) r^2) = r^L \exp(-\alpha_L r^2)
$$

The minimal $\alpha_L$ in all orbital pairs is assigned to $\alpha$ in (??). Then $n$ is estimated to make the largest auxiliary exponent $\zeta$ as close as possible to the maximum $\alpha_L$. 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'`: 

```bash
# 109.432313679876
mf = mf.density_fit()
mf.run()  # -109.432329411505
mf.with_df.auxbasis = 'weigend'
mf.run()  # -109.432334646584
```
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:

```python
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(verbos=4)  # converged SCF energy = -
-129.0896469563
mf = scf.RHF(mol).run(verbos=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:

```python
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`

### 1.9.2 Examples

Relevant examples:
- examples/df/00-with_df.py
- examples/df/01-auxbasis.py
- examples/df/10-access_df_integrals.py
- examples/df/11-get_j_io_free.py
- examples/df/40-precompute_df_integrals.py
- examples/df/41-df_integrals_gradients.py
- examples/df/42-overwrite_get_jk.py

### 1.9.3 Program reference

**DF class**

```python
class pyscf.df.df.DF(mol, auxbasis=None)
```

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, i.e., the elements of $V_{x,i,j}$ with $i \geq j$ are exist in the DF integral tensor. Thus the shape of DF integral tensor is $(M,N^2(N+1)/2)$, 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 IO performance.

**reset** *(mol=None)*

Reset mol and clean up relevant attributes for scanner mode.

**df.incore**

*pyscf.df.incore.aux_e1*(mol, auxmol, intor='int3c2e', aosym='s1', comp=None, out=None)

3-center 2-electron AO integrals ($L_{ij}$), where $L$ is the auxiliary basis.

Note aux_e1 is basically analogous to aux_e2 function. It can be viewed as the version of transposed aux_e2 tensor: if comp == 1:

```python
aux_e1 = aux_e2().T
```

else:

```python
aux_e1 = aux_e2().transpose(0,2,1)
```

The same arguments as function aux_e2 can be input to aux_e1.

*pyscf.df.incore.aux_e2*(mol, auxmol, intor='int3c2e', aosym='s1', comp=None, out=None, cinopt=None)

3-center AO integrals ($ij_{\ell}$), where $L$ is the auxiliary basis.

**Kwargs:**

```python
cinopt [Libcint-3.14 and newer version support to compute int3c2e] without the opt for the 3rd index. It can be precomputed to reduce the overhead of cinopt initialization repeatedly.
```

```python
cinopt = gto.moleintor.make_cintopt(mol._atm, mol._bas, mol._env, 'int3c2e')
```

**Returns:** 2D array of (naux,nao*(nao+1)/2) in C-contiguous

*pyscf.df.incore.cholesky_eri*(mol, auxbasis='weigend+etb', auxmol=None, int3c='int3c2e', aosym='s2ij', int2c='int2c2e', comp=1, max_memory=2000, verbose=0, fauxe2=None)

**Returns:** 2D array of (naux,nao*(nao+1)/2) in C-contiguous

*pyscf.df.incore.fill_2c2e*(mol, auxmol, intor='int2c2e', comp=None, hermi=1, out=None)

2-center 2-electron AO integrals for auxiliary basis (auxmol)

**df.outcore**

*pyscf.df.outcore.cholesky_eri*(mol, erifile, auxbasis='weigend+etb', dataname='j3c', tmpdir=None, int3c='int3c2e', aosym='s2ij', int2c='int2c2e', comp=1, max_memory=2000, auxmol=None, verbose=3)

3-index density-fitting tensor.
pyscf.df.outcore.cholesky_eri_b(mol, erifile, auxbasis='weigend+etb', dataname='j3c', int3c='int3c2e', aosym='s2ij', int2c='int2c2e', comp=1, max_memory=2000, auxmol=None, verbose=3)

3-center 2-electron DF tensor.

pyscf.df.outcore.general(mol, mo_coeffs, erifile, auxbasis='weigend+etb', dataname='eri_mo', tmpdir=None, int3c='int3c2e', aosym='s2ij', int2c='int2c2e', comp=1, max_memory=2000, verbose=0, compact=True)

Transform ij of (ijL) to MOs.

df.addons

df.addons.aug_etb(mol, beta=2.0)

To generate the even-tempered auxiliary Gaussian basis

df.addons.aug_etb_for_dfbasis(mol, dfbasis='weigend', beta=2.0, start_at=36)

augment weigend basis with even-tempered gaussian basis exps = alpha*beta^i for i = 1..N

class 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)

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

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 generating auxiliary fitting basis.

df.df

J-metric density fitting

class pyscf.df.df.DF(mol, auxbasis=None)

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 \((M,N^2(N+1)/2), (N+1)/2, N)\).
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 IO performance.

**reset** *(mol=None)*

Reset mol and clean up relevant attributes for scanner mode

class **pyscf.df.df.DF4C** *(mol, auxbasis=None)*

Relativistic 4-component

**df.df_jk**

**pyscf.df.df_jk.density_fit** *(mf, auxbasis=None, with_df=None, only_dfj=False)*

For the given SCF object, update the J, K matrix constructor with corresponding density fitting integrals.

*Args:* mf : an SCF object

*Kwargs:*

auxbasis [str or basis dict] Same format to the input attribute mol.basis. If auxbasis is None, optimal auxiliary basis based on AO basis (if possible) or even-tempered Gaussian basis will be used.

only_dfj [str] Compute Coulomb integrals only and no approximation for HF exchange. Same to RI-JONX in ORCA

*Returns:* An SCF object with a modified J, K matrix constructor which uses density fitting integrals to compute J and K

Examples:

```python
gto.M(atom='H 0 0 0; F 0 0 1', basis='ccpvdz', verbose=0)
```

```python
mf = scf.density_fit(scf.RHF(mol))
mf.scf() -100.005306000435510
```

```python
mol.symmetry = 1
mol.build(0, 0)
mf = scf.density_fit(scf.UHF(mol))
mf.scf() -100.005306000435510
```

**pyscf.df.df_jk.r_get_jk** *(dfobj, dms, hermi=1, with_j=True, with_k=True)*

Relativistic density fitting JK

**df.r_incore**

**df.grad.rhf**

**pyscf.df.grad.rhf.Grad**

alias of **Gradients**

class **pyscf.df.grad.rhf.Grad** *(mf)*

Restricted density-fitting Hartree-Fock gradients
df.grad.uhf

pyscf.df.grad.uhf.Grad
    alias of Gradients
class pyscf.df.grad.uhf.Grades(mf)
    Unrestricted density-fitting Hartree-Fock gradients

df.grad.rks

pyscf.df.grad.rks.get_veff(ks_grad, mol=None, dm=None)
    Coulomb + XC functional

df.grad.ks

pyscf.df.grad.ks.get_veff(ks_grad, mol=None, dm=None)
    Coulomb + XC functional

df.hessian.rhf

Non-relativistic RHF analytical Hessian with density-fitting approximation
Ref: [1] Efficient implementation of the analytic second derivatives of
    Hartree-Fock and hybrid DFT energies: a detailed analysis of different approximations. Dmytro Bykov,
class pyscf.df.hessian.rhf.Hessian(mf)
    Non-relativistic restricted Hartree-Fock hessian

df.hessian.uhf

Non-relativistic UHF analytical Hessian
class pyscf.df.hessian.uhf.Hessian(mf)
    Non-relativistic UHF hessian

df.hessian.rks

Non-relativistic RKS analytical Hessian
class pyscf.df.hessian.rks.Hessian(mf)
    Non-relativistic RKS hessian

df.hessian.ks

Non-relativistic UKS analytical Hessian
class pyscf.df.hessian.ks.Hessian(mf)
    Non-relativistic RKS hessian
1.10 symm – Point group symmetry and spin symmetry

This module offers the functions to detect point group symmetry, basis symmetrization, 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

<table>
<thead>
<tr>
<th>D2h</th>
<th>A1g</th>
<th>B1g</th>
<th>B2g</th>
<th>B3g</th>
<th>A1u</th>
<th>B1u</th>
<th>B2u</th>
<th>B3u</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1g</td>
<td>A1g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1g</td>
<td>B1g</td>
<td>A1g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2g</td>
<td>B2g</td>
<td>B3g</td>
<td>A1g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B3g</td>
<td>B3g</td>
<td>B2g</td>
<td>B1g</td>
<td>A1g</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1u</td>
<td>A1u</td>
<td>B1u</td>
<td>B2u</td>
<td>B3u</td>
<td>A1g</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1u</td>
<td>B1u</td>
<td>A1u</td>
<td>B3u</td>
<td>B2u</td>
<td>B1g</td>
<td>A1g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2u</td>
<td>B2u</td>
<td>B3u</td>
<td>A1u</td>
<td>B1u</td>
<td>B2g</td>
<td>B3g</td>
<td>A1g</td>
<td></td>
</tr>
<tr>
<td>B3u</td>
<td>B3u</td>
<td>B2u</td>
<td>B1u</td>
<td>A1u</td>
<td>B3g</td>
<td>B2g</td>
<td>B1g</td>
<td>A1g</td>
</tr>
</tbody>
</table>

The multiplication table for XOR operator is

<table>
<thead>
<tr>
<th>XOR</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>010</td>
<td>011</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>011</td>
<td>010</td>
<td>000</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>110</td>
<td>001</td>
<td>000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>110</td>
<td>111</td>
<td>100</td>
<td>101</td>
<td>010</td>
<td>011</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>111</td>
<td>110</td>
<td>101</td>
<td>100</td>
<td>011</td>
<td>010</td>
<td>001</td>
<td>000</td>
</tr>
</tbody>
</table>

Comparing the two tables, we notice that the two tables can be changed to each other with the mapping

<table>
<thead>
<tr>
<th>D2h</th>
<th>XOR</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1g</td>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>B1g</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>B2g</td>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>B3g</td>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>A1u</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>B1u</td>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>B2u</td>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>B3u</td>
<td>111</td>
<td>7</td>
</tr>
</tbody>
</table>

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).

<table>
<thead>
<tr>
<th>C2h</th>
<th>XOR</th>
<th>ID</th>
<th>C2v</th>
<th>XOR</th>
<th>ID</th>
<th>D2</th>
<th>XOR</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ag</td>
<td>00</td>
<td>0</td>
<td>A1</td>
<td>00</td>
<td>0</td>
<td>A1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>Bg</td>
<td>01</td>
<td>1</td>
<td>A2</td>
<td>01</td>
<td>1</td>
<td>B1</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>Au</td>
<td>10</td>
<td>2</td>
<td>B1</td>
<td>10</td>
<td>2</td>
<td>B2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>Bu</td>
<td>11</td>
<td>3</td>
<td>B2</td>
<td>11</td>
<td>3</td>
<td>B3</td>
<td>11</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cs</th>
<th>XOR</th>
<th>ID</th>
<th>Ci</th>
<th>XOR</th>
<th>ID</th>
<th>C2</th>
<th>XOR</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>A'</td>
<td>0</td>
<td>0</td>
<td>Ag</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B''</td>
<td>1</td>
<td>1</td>
<td>Au</td>
<td>1</td>
<td>1</td>
<td>B</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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)
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 labelled with the IDs instead of the irreps’ symbols. We can use `symm.addons.irrep_id2name()` to convert the ID to irreps’ symbol, e.g.:

```python
>>> 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.
- 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
- MP2 and CCSD
  Point group symmetry are not supported in CCSD, MP2.

Examples

Relevant examples `examples/symm/30-guess_symmetry.py` `examples/symm/31-symmetrize_space.py` `examples/symm/32-symmetrize_natural_orbital.py` `examples/symm/33-lz_adaption.py`

Program reference

1.10.2 geom

```python
pyscf.symm.geom.alias_axes(axes, ref)
```
Rename axes, make it as close as possible to the ref axes

```python
pyscf.symm.geom.detect_symm(atoms, basis=None, verbose=2)
```
Detect the point group symmetry for given molecule.

```python
pyscf.symm.geom.rotation_mat(vec, theta)
```
rotate angle theta along vec new(x,y,z) = R * old(x,y,z)

```python
pyscf.symm.geom.symm_identical_atoms(gpname, atoms)
```
Requires

1.10.3 Symmetry adapted basis

Generate symmetry adapted basis

```python
pyscf.symm.basis.linearmole_symm_descent(gpname, irrepid)
```
Map irreps to D2h or C2v

1.10.4 addons

```python
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:
```python
>>> 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 symbol, 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:

```python
>>> 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-orthogonal 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:

```python
>>> 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))
```

pyscf.symm.addons.symmetrize_space(mol, mo, s=None, check=True, tol=1e-07)
Symmetrize the given orbital space.

This function is different to the symmetrize_orb(): In this function, the given orbitals are mixed to reveal the symmetry; 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:

```python
>>> 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))

1.10.5 Clebsch Gordon coefficients

pyscf.symm.cg.cg_spin\(l, j\text{double}, m\text{double}, \text{spin}\)
Clebsch Gordon coefficient of \(<l,m,1/2,\text{spin}|j,m\text{spin}\)\>

1.10.6 Parameters

1.10.7 Spherical harmonics

Spherical harmonics

pyscf.symm.sph.cart2spinor\((l)\)
Cartesian to spinor for angular moment \(l\)

pyscf.symm.sph.multipoles\((r, lmax, \text{reorder_dipole=True})\)
Compute all multipoles upto \(lmax\)
rad = numpy.linalg.norm(r, axis=1)
ylms = real_ylm(r/rad.reshape(-1,1), lmax) pol = [rad**l*y for l, y in enumerate(ylms)]

Kwarg:
reorder_p [bool] sort dipole to the order (x,y,z)

pyscf.symm.sph.real2spinor\_whole\((mol)\)
Transformation matrix that transforms real-spherical GTOs to spinor GTOs for all basis functions

Examples:

```python
>>> from pyscf import gto
>>> from pyscf.symm import sph
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1', basis='ccpvtz')
>>> ca, cb = sph.sph2spinor_coeff(mol)
>>> s0 = mol.intor('int1e_ovlp_spinor')
>>> s1 = ca.conj().T.dot(mol.intor('int1e_ovlp_sph')).dot(ca)
>>> s1+= cb.conj().T.dot(mol.intor('int1e_ovlp_sph')).dot(cb)
>>> print(abs(s1-s0).max())
6.66133814775e-16
```

pyscf.symm.sph.real\_sph\_vec\((r, lmax, \text{reorder_p=False})\)
Computes (all) real spherical harmonics up to the angular momentum \(lmax\)

pyscf.symm.sph.sph2spinor\_coeff\((mol)\)
Transformation matrix that transforms real-spherical GTOs to spinor GTOs for all basis functions

Examples:

```python
>>> from pyscf import gto
>>> from pyscf.symm import sph
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1', basis='ccpvtz')
>>> ca, cb = sph.sph2spinor_coeff(mol)
>>> s0 = mol.intor('int1e_ovlp_spinor')
>>> s1 = ca.conj().T.dot(mol.intor('int1e_ovlp_sph')).dot(ca)
>>> s1+= cb.conj().T.dot(mol.intor('int1e_ovlp_sph')).dot(cb)
```
pyscf.symm.sph.sph_pure2real(l, reorder_p=True)
Transformation matrix: from the pure spherical harmonic functions Y_m to the real spherical harmonic functions O_m.

\[ O_m = \sum Y_m' \times U(m',m) \]

\[ Y(-1) = \frac{1}{\sqrt{2}} \{ -iO(-1) + O(1) \}; \quad Y(1) = \frac{1}{\sqrt{2}} \{ -iO(-1) - O(1) \}; \quad Y(-2) = \frac{1}{\sqrt{2}} \{ -iO(-2) + O(2) \}; \quad Y(2) = \frac{1}{\sqrt{2}} \{ iO(-2) + O(2) \} \]

Kwargs: reorder_p (bool): Whether the p functions are in the (x,y,z) order.

Returns: 2D array U_{complex,real}

pyscf.symm.sph.sph_real2pure(l, reorder_p=True)
Transformation matrix: from real spherical harmonic functions to the pure spherical harmonic functions.

Kwargs: reorder_p (bool): Whether the real p functions are in the (x,y,z) order.

1.10.8 Wigner rotation Dmatrix

Wigner rotation D-matrix for real spherical harmonics

pyscf.symm.Dmatrix.Dmatrix(l, alpha, beta, gamma, reorder_p=False)
Wigner rotation D-matrix

\[ D_{mm'} = <lm|R(alpha,beta,gamma)|lm'> \]
alpha, beta, gamma are Euler angles (in z-y-z convention)

Kwargs: reorder_p (bool): Whether to put the p functions in the (x,y,z) order.

pyscf.symm.Dmatrix.dmatrix(l, beta, reorder_p=False)
Wigner small-d matrix (in z-y-z convention)

pyscf.symm.Dmatrix.get_euler_angles(c1, c2)
Find the three Euler angles (alpha, beta, gamma in z-y-z convention) that rotates coordinates c1 to coordinates c2.

\[ yp = \text{numpy.einsum}('j,kj->k', c1[1], \text{geom.rotation_mat}(c1[2], beta)) \]
\[ \text{tmp} = \text{numpy.einsum}('ij,kj->ik', c1, \text{geom.rotation_mat}(c1[2], alpha)) \]
\[ c2 = \text{numpy.einsum}('ij,kj->ik', \text{tmp}, \text{geom.rotation_mat}(c2[2], gamma)) \]

(For backward compatibility) if c1 and c2 are two points in the real space, the Euler angles define the rotation transforms the old coordinates to the new coordinates (new_x, new_y, new_z) in which c1 is identical to c2.

\[ \text{tmp} = \text{numpy.einsum}('j,kj->k', c1, \text{geom.rotation_mat}((0,0,1), gamma)) \]
\[ c2 = \text{numpy.einsum}('j,kj->k', \text{tmp}, \text{geom.rotation_mat}((0,0,1), alpha)) \]

1.11 scf — Mean-field methods

The scf module implements restricted and unrestricted, closed shell and open shell Hartree-Fock methods. To run a simple HF calculation with SCF:

```python
import pyscf
mol = pyscf.M(  
    atom = 'H 0 0 0; F 0 0 1.1',  # in Angstrom
)```
from pyscf import gto, scf
mol = gto.M(atom='H 0 0 0; F 0 0 1.1', # in Angstrom
            basis = 'ccpvdz',
            symmetry = True,
          )
mhf = scf.HF(mol)
mhf.verbose=4
mf = scf.addons.dynamic_sz_(mf)

1.11.1 Examples

Relevant examples examples/scf/00-simple_hf.py examples/scf/01-h2o.py
examples/scf/02-ghf.py examples/scf/02-rohf_uhf.py examples/scf/03-level_shift.py
examples/scf/04-dirac_hf.py examples/scf/05-breit_gaunt.py
examples/scf/10-glycine.py examples/scf/11-linear_dep.py examples/scf/12-fast_hf.py
examples/scf/13-symmetry.py examples/scf/15-initial_guess.py
examples/scf/16-h2_scan.py examples/scf/17-stability.py examples/scf/20-density_fitting.py
examples/scf/21-x2c.py examples/scf/22-newton.py examples/scf/23-decorate_scf.py
examples/scf/24-callback.py examples/scf/24-tune_diis.py examples/scf/30-scan_pes.py
examples/scf/31-cr_atom_rohf_tune_init_guess.py examples/scf/31-v_atom_rohf.py
examples/scf/32-break_spin_symm.py examples/scf/40-apply_electric_field.py
examples/scf/40-customizing_hamiltonian.py examples/scf/41-hf_with_given_densityfit_ints.py
examples/scf/42-remove_linear_dep.py examples/scf/43-custom_get_jk.py
examples/scf/50-mom-deltaSCF.py examples/scf/51-elecoup_mom.py
examples/scf/52-dynamically_control_level_shift.py examples/scf/53-dynamic_sz.py
examples/scf/54-fractional_occupancy.py examples/scf/55-overload_convergence_criteria.py
examples/scf/56-h2_symm_breaking.py examples/scf/71-patch_scf_kernel.py
examples/scf/70-background_and_hot_tuning.py examples/scf/71-patch_scf_kernel.py

1.11.2 Stability analysis

1.11.3 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='0 0 0 0; 0 0 0 1')
mf = scf.UHF(mol)
mf.verbose=4
mf = scf.addons.dynamic_sz_(mf)
This function automatically converges the ground state 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:

```python
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
```

```python
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()
```

### 1.11.4 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 `max_mem`) 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.

### 1.11.5 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 may 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:

```python
import numpy
tfrom 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 de-
composed integrals. See also Saving/Loading DF integral tensor.

### 1.11.6 Program reference

#### Hartree-Fock

Simple usage:

```python
>>> 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.


**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 environment.

**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.

```python
>>> 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 (SSISS). 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. Writing to chkfile can be disabled if this attribute is set to None or False.
conv_tol[float] converge threshold. Default is 1e-9

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’, ‘hcore’, ‘1e’, ‘chkfile’. Default is ‘minao’

DIIS[DIIS class] The class to generate diis object. It can be one of diis.SCF_DIIS, diis.ADIIS, diis.EDIIS.

diis[boolean or object of DIIS class defined in scf.diis.] Default is the object associated to the attribute self.DIIS. Set it to None/False to turn off DIIS. Note if this attribute is initialized as a DIIS object, the SCF driver will use this object in the iteration. The DIIS informations (vector basis and error vector) will be held inside this object. When kernel function is called again, the old states (vector basis and error vector) will be reused.

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.


callback[function(envs_dict) => None] callback function takes one dict as the argument which is generated by the built-in function locals(), so that the callback function can access all local variables in the current environment.

conv_check[bool] An extra cycle to check convergence after SCF iterations.

check_convergence [function(envs) => bool] A hook for overloading convergence criteria in 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:

```python
>>> 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(verbosе=None, with_meta_lowdin=True, **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:

```python
>>> 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'))
-98.4175042429368
```

canonicalize \((mf, mo_coeff, mo_occ, fock=None)\)
Canonicalization diagonalizes the Fock matrix within occupied, open, virtual subspaces separately (without change occupancy).

dip_moment \((mol=None, dm=None, unit='Debye', verbose=3, **kwargs)\)
Dipole moment calculation

\[
\begin{align*}
\mu_x &= -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|x|\mu) + \sum_A Q_A X_A \\
\mu_y &= -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|y|\mu) + \sum_A Q_A Y_A \\
\mu_z &= -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|z|\mu) + \sum_A Q_A Z_A
\end{align*}
\]

where \(\mu_x, \mu_y, \mu_z\) 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:

\[
E = \sum_{ij} h_{ij} \gamma_{ji} + \sum_{ijkl} \frac{1}{2} \gamma_{ji} \gamma_{lk} \langle ik||jl\rangle
\]

Note this function has side effects which cause mf.scf_summary updated.

Args:
mf : an instance of SCF class

Kwargs:
dm  [2D ndarray] one-partial density matrix
h1e  [2D ndarray] Core hamiltonian
vhf  [2D ndarray] HF potential

**Returns:** Hartree-Fock electronic energy and the Coulomb energy

**Examples:**

```python
>>> 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)
>>> mf.energy_elec(dm)
(-1.5176090667746334, 0.60917167853723675)
```

**energy_tot** *(mf, dm=None, h1e=None, vhf=None)*

Total Hartree-Fock energy, electronic part plus nuclear repulsion See `scf.hf.energy_elec()` for the electron part

Note this function has side effects which cause `mf.scf_summary` updated.

**from_chk** *(chkfile=None, project=None)*

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)*

\[ F = h^{\text{core}} + V^{\text{HF}} \]

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)

**Kwargs:**

- **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
- **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 orbital 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, omega=None)
Compute J matrices for all input density matrices

get_jk (mol=None, dm=None, hermi=1, with_j=True, with_k=True, omega=None)
Compute J, K matrices for all input density matrices

Args:
mol : an instance of Mole
dm [ndarray or list of ndarrays] A density matrix or a list of density matrices

Kwarg:
hermi [int] Whether J, K matrix is hermitian

0 : not hermitian and not symmetric
1 : hermitian or symmetric
2 : anti-hermitian

vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices

with_j [boolean] Whether to compute J matrices

with_k [boolean] Whether to compute K matrices

omega [float] Parameter of range-separated Coulomb operator: erf(omega * r12) / r12. If specified, integration are evaluated based on the long-range part of the range-separated Coulomb operator.

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:

```python
>>> 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, omega=None)
Compute K matrices for all input density matrices

get_occ (mf, mo_energy=None, mo_coeff=None)
Label the occupancies for each orbital

Kwarg:
mo_energy [1D ndarray] Orbital energies

mo_coeff [2D ndarray] Orbital coefficients

Examples:

```python
>>> 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])
```
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

Kwarg:

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_le (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=None)
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:

```python
>>> 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 (*args, **kwargs)
SCF main driver

Kwargs:
- dm0 [ndarray] If given, it will be used as the initial guess density matrix

Examples:

```python
>>> 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, **kwargs)
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 carried 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

Returns: A list : pop, charges
**mulliken_pop** *(mol=None, dm=None, s=None, verbose=5)*

Mulliken population analysis

\[ M_{ij} = D_{ij} S_{ji} \]

Mulliken charges

\[ \delta_i = \sum_j M_{ij} \]

**Returns:** A list: pop, charges

- **pop** [nparray] Mulliken population on each atomic orbitals
- **charges** [nparray] Mulliken charges

**mulliken_pop_meta_lowdin_ao** *(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 carried 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

**Returns:** A list: pop, charges

- **pop** [nparray] Mulliken population on each atomic orbitals
- **charges** [nparray] Mulliken charges

**nuc_grad_method** *

Hook to create object for analytical nuclear gradients.

**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 carried 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

**Returns:** A list : pop, charges

- **pop** [nparray] Mulliken population on each atomic orbitals
- **charges** [nparray] Mulliken charges

**reset** (*mol=None*)
Reset mol and clean up relevant attributes for scanner mode

**scf** (*dm0=None, **kwargs*)
SCF main driver

**Kwargs:**

- **dm0** [ndarray] If given, it will be used as the initial guess density matrix

**Examples:**

```python
>>> 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
```

**to_ghf** (*mf*)
Convert the input mean-field object to a GHF/GKS object

**to_rhf** (*mf*)
Convert the input mean-field object to a RHF/ROHF/RKS/ROKS object

**to_uhf** (*mf*)
Convert the input mean-field object to a UHF/UKS object

**update** (*chkfile=None*)
Read attributes from the chkfile then replace the attributes of current object. It’s an alias of function `update_from_chk_`

**update_** (*chkfile=None*)
Read attributes from the chkfile then replace the attributes of current object. It’s an alias of function `update_from_chk_`

**update_from_chk** (*chkfile=None*)
Read attributes from the chkfile then replace the attributes of current object. It’s an alias of function `update_from_chk_`

**update_from_chk_** (*chkfile=None*)
Read attributes from the chkfile then replace the attributes of current object. It’s an alias of function `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. Writing to chkfile can be disabled if this
attribute is set to None or False.

conv_tol [float] converge threshold. Default is 1e-9

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’, ‘hcore’, ‘1e’, ‘chkfile’. Default is
‘minao’

DIIS [DIIS class] The class to generate diis object. It can be one of diis.SCF_DIIS, diis.ADIIS,
diis.EDIIS.

diis [boolean or object of DIIS class defined in scf.diis.] Default is the object associated to the
attribute self.DIIS. Set it to None/False to turn off DIIS. Note if this attribute is inialized as a
DIIS object, the SCF driver will use this object in the iteration. The DIIS informations (vector basis
and error vector) will be held inside this object. When kernel function is called again, the old states
(vector basis and error vector) will be reused.

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.


callback [function(envs_dict) => None] callback function takes one dict as the argument which is gener-
ated by the builtin function locals(), so that the callback function can access all local variables
in the current environment.

conv_check [bool] An extra cycle to check convergence after SCF iterations.

check_convergence [function(envs) => bool] A hook for overloading convergence criteria in SCF itera-
tions.

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:

```python
>>> 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
```
CASCI (obj, *args, **kwargs)

Args:
    mf_or_mol [SCF object or Mole object] SCF or Mole to define the problem size.
    ncas [int] Number of active orbitals.
    nelecas [int or a pair of int] Number of electrons in active space.

Kwargs:
    ncore [int] Number of doubly occupied core orbitals. If not presented, this parameter can be automatically determined.

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 electron numbers.
    natorb [bool] Whether to transform natural orbital in active space. Be cautious of this parameter when CASCI/CASSCF are combined with DMRG solver or selected CI solver because DMRG and selected CI are not invariant to the rotation in active space. False by default.
    canonicalization [bool] Whether to canonicalize orbitals. Note that canonicalization does not change the orbitals in active space by default. It only diagonalizes core and external space of the general Fock matrix. To get the natural orbitals in active space, attribute natorb need to be enabled. True by default.
    sorting_mo_energy [bool] Whether to sort the orbitals based on the diagonal elements of the general Fock matrix. Default is False.
    fcisolver [an instance of FCI solver] The pyscf.fci module provides several FCI solver for different scenario. Generally, fci.direct_spin1.FCISolver can be used for all RHF-CASSCF. However, a proper FCI solver can provide better performance and better numerical stability. One can either use fci.solver() function to pick the FCI solver by the program or manually assign the FCI solver to this attribute, e.g.

```python
>>> 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 FCI solver by setting e.g.:

```python
>>> mc.fcisolver.max_cycle = 30
>>> mc.fcisolver.conv_tol = 1e-7
```

For more details of the parameter for FCI solver, See fci.

Saved results
    e_tot [float] Total MCSCF energy (electronic energy plus nuclear repulsion)
    e_cas [float] CAS space FCI energy
ci [ndarray] CAS space FCI coefficients

**mo_coeff** [ndarray] When canonicalization is specified, the orbitals are canonical orbitals which make the general Fock matrix (Fock operator on top of MCSCF 1-particle density matrix) diagonalized within each subspace (core, active, external). If natorb (natural orbitals in active space) is specified, the active segment of the mo_coeff is natural orbitals.

**mo_energy** [ndarray] Diagonal elements of general Fock matrix (in mo_coeff representation).

Examples:
```python
>>> 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
```

**CASCI** *(obj, *args, **kwargs)*

CASSCF

**Args:**

- **mf_or_mol** [SCF object or Mole object] SCF or Mole to define the problem size.
- **ncas** [int] Number of active orbitals.
- **nelecas** [int or a pair of int] Number of electrons in active space.

**Kwargs:**

- **ncore** [int] Number of doubly occupied core orbitals. If not presented, this parameter can be automatically determined.

**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 electron numbers.
- **natorb** [bool] Whether to transform natural orbital in active space. Be cautious of this parameter when CASCI/CASSCF are combined with DMRG solver or selected CI solver because DMRG and selected CI are not invariant to the rotation in active space. False by default.
- **canonicalization** [bool] Whether to canonicalize orbitals. Note that canonicalization does not change the orbitals in active space by default. It only diagonalizes core and external space of the general Fock matrix. To get the natural orbitals in active space, attribute natorb need to be enabled. True by default.
- **sorting_mo_energy** [bool] Whether to sort the orbitals based on the diagonal elements of the general Fock matrix. Default is False.
- **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 assign the FCISolver to this attribute, e.g.

```python
>>> 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.:

```python
>>> 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)
- `e_cas` [float] CAS space FCI energy
- `ci` [ndarray] CAS space FCI coefficients
  When canonicalization is specified, the orbitals are canonical orbitals which make the general Fock matrix (Fock operator on top of MCSCF 1-particle density matrix) diagonalized within each subspace (core, active, external). If natorb (natural orbitals in active space) is specified, the active segment of the `mo_coeff` is natural orbitals.
- `mo_energy` [ndarray] Diagonal elements of general Fock matrix (in `mo_coeff` representation).

Examples:

```python
>>> 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 preferred. 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 allowed 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.

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 environment.

scale_restoration [float] When a step of orbital rotation moves out of trust region, the orbital optimization will be restored to previous state and the step size of the orbital rotation needs to be reduced. scale_restoration controls how much to scale down the step size.

Saved results

e_tot [float] Total MCSCF energy (electronic energy plus nuclear repulsion)

e_cas [float] CAS space FCI energy

ci [ndarray] CAS space FCI coefficients

mo_coeff [ndarray] Optimized CASSCF orbitals coefficients. When canonicalization is specified, the returned orbitals make the general Fock matrix (Fock operator on top of MCSCF 1-particle density matrix) diagonalized within each subspace (core, active, external). If natorb (natural orbitals in active space) is specified, the active segment of the mo_coeff is natural orbitals.

mo_energy [ndarray] Diagonal elements of general Fock matrix (in mo_coeff representation).

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

Gradients \((obj, *args, **kwargs)\)
Non-relativistic restricted Hartree-Fock gradients

TDA \((obj, *args, **kwargs)\)
Tamm-Dancoff approximation

Attributes:
- **conv_tol** [float] Diagonalization convergence tolerance. Default is 1e-9.
- **nstates** [int] Number of TD states to be computed. Default is 3.

Saved results:
- **converged** [bool] Diagonalization converged or not
- **e** [1D array] excitation energy for each excited state.
- **xy** [A list of two 2D arrays] The two 2D arrays are Excitation coefficients \(X\) (shape 
  \([\text{nocc}, \text{nvir}]\)) and de-excitation coefficients \(Y\) (shape 
  \([\text{nocc}, \text{nvir}]\)) for each excited state. 
  \((X,Y)\) are normalized to 1/2 in RHF/RKS methods and normalized to 1 for UHF/UKS 
  methods. In the TDA calculation, \(Y = 0\).

TDHF \((obj, *args, **kwargs)\)
Time-dependent Hartree-Fock

Attributes:
- **conv_tol** [float] Diagonalization convergence tolerance. Default is 1e-9.
- **nstates** [int] Number of TD states to be computed. Default is 3.

Saved results:
- **converged** [bool] Diagonalization converged or not
- **e** [1D array] excitation energy for each excited state.
- **xy** [A list of two 2D arrays] The two 2D arrays are Excitation coefficients \(X\) (shape 
  \([\text{nocc}, \text{nvir}]\)) and de-excitation coefficients \(Y\) (shape 
  \([\text{nocc}, \text{nvir}]\)) for each excited state. 
  \((X,Y)\) are normalized to 1/2 in RHF/RKS methods and normalized to 1 for UHF/UKS 
  methods. In the TDA calculation, \(Y = 0\).

convert_from \((mf)\)
Convert the input mean-field object to RHF/ROHF

get_jk \((mol=None, dm=None, hermi=1, with_j=True, with_k=True, omega=None)\)
Compute \(J, K\) matrices for all input density matrices

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 : not hermitian and not symmetric
1 : hermitian or symmetric
2 : anti-hermitian

vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices

with_j [boolean] Whether to compute J matrices
with_k [boolean] Whether to compute K matrices

omega [float] Parameter of range-separated Coulomb operator: erf(omega * r12) / r12. If specified, integration are evaluated based on the long-range part of the range-separated Coulomb operator.

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:

```python
>>> 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_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:

```python
>>> 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)
```
```python
>>> 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
```

**spin_square** *(mo_coeff=None, s=None)*

Spin square and multiplicity of RHF determinant

**stability** *(internal=True, external=False, verbose=None)*

RHF/RKS stability analysis.

See also pyscf.scf.stability.rhf_stability function.

**Kwargs:**

- **internal** [bool] Internal stability, within the RHF optimization space.

**Returns:** New orbitals that are more close to the stable condition. The return value includes two set of orbitals. The first corresponds to the internal stability and the second corresponds to the external stability.

```python
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. Writing to chkfile can be disabled if this attribute is set to None or False.
- **conv_tol** [float] converge threshold. Default is 1e-9
- **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’, ‘hcore’, ‘1e’, ‘chkfile’. Default is ‘minao’
- **DIIS** [DIIS class] The class to generate diis object. It can be one of diis.SCF_DIIS, diis.ADIIS, diis.EDIIS.
- **diis** [boolean or object of DIIS class defined in scf.diis] Default is the object associated to the attribute self.DIIS. Set it to None/False to turn off DIIS. Note if this attribute is inialized as a DIIS object, the SCF driver will use this object in the iteration. The DIIS informations (vector basis and error vector) will be held inside this object. When kernel function is called again, the old states (vector basis and error vector) will be reused.
- **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.

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 environment.

conv_check [bool] An extra cycle to check convergence after SCF iterations.

check_convergence [function(envs) => bool] A hook for overloading convergence criteria in 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:

```python
>>> 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
```

CASSCI (obj, *args, **kwargs)

Args:

mf_or_mol [SCF object or Mole object] SCF or Mole to define the problem size.

ncas [int] Number of active orbitals.

nelecars [int or a pair of int] Number of electrons in active space.

Kwargs:

ncore [int] Number of doubly occupied core orbitals. If not presented, this parameter can be automatically determined.

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.

nelecars [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 electron numbers.

natorb [bool] Whether to transform natural orbital in active space. Be cautious of this parameter when CASSCI/CASSCF are combined with DMRG solver or selected CI solver because DMRG and selected CI are not invariant to the rotation in active space. False by default.
canonicalization [bool] Whether to canonicalize orbitals. Note that canonicalization does not change the orbitals in active space by default. It only diagonalizes core and external space of the general Fock matrix. To get the natural orbitals in active space, attribute natorb need to be enabled. True by default.

sorting_mo_energy [bool] Whether to sort the orbitals based on the diagonal elements of the general Fock matrix. Default is False.

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 assign the FCISolver to this attribute, e.g.

```python
>>> 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.:

```python
>>> 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)
e_cas [float] CAS space FCI energy
ci [ndarray] CAS space FCI coefficients
mo_coeff [ndarray] When canonicalization is specified, the orbitals are canonical orbitals which make the general Fock matrix (Fock operator on top of MCSCF 1-particle density matrix) diagonalized within each subspace (core, active, external). If natorb (natural orbitals in active space) is specified, the active segment of the mo_coeff is natural orbitals.
mo_energy [ndarray] Diagonal elements of general Fock matrix (in mo_coeff representation).

Examples:

```python
>>> from pyscf import gto, scf, mcscf
gf = gto.M(atom='N 0 0 0; N 0 0 1', basis='ccpvdz', verbose=0)
>>> mf = scf.RHF(gf)
mf.scf()
>>> mc = mcscf.CASCI(mf, 6, 6)
>>> mc.kernel()[0]
-108.980200816243354
```

CASSCF (obj, *args, **kwargs)

CASC

Args:

mf_or_mol [SCF object or Mole object] SCF or Mole to define the problem size.
ncas [int] Number of active orbitals.
nelec [int or a pair of int] Number of electrons in active space.

Kwarggs:
ncore [int] Number of doubly occupied core orbitals. If not presented, this parameter can be automatically determined.

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 electron numbers.

natorb [bool] Whether to transform natural orbital in active space. Be cautious of this parameter when CASCI/CASSCF are combined with DMRG solver or selected CI solver because DMRG and selected CI are not invariant to the rotation in active space. False by default.

canonicalization [bool] Whether to canonicalize orbitals. Note that canonicalization does not change the orbitals in active space by default. It only diagonalizes core and external space of the general Fock matrix. To get the natural orbitals in active space, attribute natorb need to be enabled. True by default.

sorting_mo_energy [bool] Whether to sort the orbitals based on the diagonal elements of the general Fock matrix. Default is False.

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 assign the FCISolver to this attribute, e.g.

```python
>>> 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.: 

```python
>>> 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)

e_cas [float] CAS space FCI energy

ci [ndarray] CAS space FCI coefficients

mo_coeff [ndarray] When canonicalization is specified, the orbitals are canonical orbitals which make the general Fock matrix (Fock operator on top of MCSCF 1-particle density matrix) diagonalized within each subspace (core, active, external). If natorb (natural orbitals in active space) is specified, the active segment of the mo_coeff is natural orbitals.

mo_energy [ndarray] Diagonal elements of general Fock matrix (in mo_coeff representation).

Examples:
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_steps` [float] The step size for orbital rotation. Small step (0.005 - 0.05) is preferred. 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 allowed 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` [float, 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.
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 environment.

scale_restoration [float] When a step of orbital rotation moves out of trust region, the orbital optimization will be restored to previous state and the step size of the orbital rotation needs to be reduced. scale_restoration controls how much to scale down the step size.

Saved results

e_tot [float] Total MCSCF energy (electronic energy plus nuclear repulsion)

e_cas [float] CAS space FCI energy

ci [ndarray] CAS space FCI coefficients

mo_coeff [ndarray] Optimized CASSCF orbitals coefficients. When canonicalization is specified, the returned orbitals make the general Fock matrix (Fock operator on top of MCSCF 1-particle density matrix) diagonalized within each subspace (core, active, external). If natorb (natural orbitals in active space) is specified, the active segment of the mo_coeff is natural orbitals.

mo_energy [ndarray] Diagonal elements of general Fock matrix (in mo_coeff representation).

Examples:

```python
>>> 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
```

Gradients (obj, *args, **kwargs)

Non-relativistic ROHF gradients

analyze (verbose=None, with_meta_lowdin=True, **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 separately (without change occupancy).

check_sanity ()

Check input of class/object attributes, check whether a class method is overwritten. It does not check the attributes which are prefixed with "_". The return value of method set is the object itself. This allows a series of functions/methods to be executed in pipe.

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:

```python
>>> 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

\[
V^\alpha_{ij} = \sum_{kl} (ij|kl)(\gamma^\alpha_{lk} + \gamma^\beta_{lk}) - \sum_{kl} (il|kj)\gamma^\alpha_{lk} \\
V^\beta_{ij} = \sum_{kl} (ij|kl)(\gamma^\alpha_{lk} + \gamma^\beta_{lk}) - \sum_{kl} (il|kj)\gamma^\beta_{lk}
\]

Args:  
mol : an instance of Mole

dm [a list of ndarrays] A list of density matrices, stored as (alpha,alpha,...,beta,beta,...)

Kwarg:

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^\alpha (and V^\beta) can be a list matrices, corresponding to the input density matrices.

Examples:

```python
>>> import numpy
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; O 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
```
make_rdm1  
\[ \text{make_rdm1}(\text{mo_coeff} = \text{None}, \text{mo_occ} = \text{None}, **\text{kwargs}) \]
One-particle electron density matrix. \( \text{mo_occ} \) is a 1D array, with occupancy 1 or 2.

spin_square  
\[ \text{spin_square}(\text{mo_coeff} = \text{None}, s = \text{None}) \]
Spin square and multiplicity of RHF determinant

stability  
\[ \text{stability}(\text{internal} = \text{True}, \text{external} = \text{False}, \text{verbose} = \text{None}) \]
ROHF/ROKS stability analysis.

See also pyscf.scf.stability.rohf_stability function.

Kwargs:
- \text{internal} \ [\text{bool}] Internal stability, within the RHF optimization space.
- \text{external} \ [\text{bool}] External stability. It is not available in current version.

Returns: The return value includes two set of orbitals which are more close to the required stable condition.

class pyscf.scf.uhf.UHF (mol)  
SCF base class. non-relativistic RHF.

Attributes:
- \text{verbose} \ [\text{int}] Print level. Default value equals to Mole.verbose
- \text{max_memory} \ [\text{float or int}] Allowed memory in MB. Default equals to Mole.max_memory
- \text{chkfile} \ [\text{str}] checkpoint file to save MOs, orbital energies etc. Writing to chkfile can be disabled if this attribute is set to None or False.
- \text{conv_tol} \ [\text{float}] converge threshold. Default is 1e-9
- \text{conv_tol_grad} \ [\text{float}] gradients converge threshold. Default is sqrt(\text{conv_tol})
- \text{max_cycle} \ [\text{int}] max number of iterations. Default is 50
- \text{init_guess} \ [\text{str}] initial guess method. It can be one of ‘minao’, ‘atom’, ‘hcore’, ‘1e’, ‘chkfile’. Default is ‘minao’
- DIIS \ [\text{DIIS class}] The class to generate diis object. It can be one of diis.SCF_DIIS, diis.ADIIS, diis.EDIIS.
- diis \ [\text{boolean or object of DIIS class defined in scf.diis}] Default is the object associated to the attribute self.DIIS. Set it to None/False to turn off DIIS. Note if this attribute is initialized as a DIIS object, the SCF driver will use this object in the iteration. The DIIS informations (vector basis and error vector) will be held inside this object. When kernel function is called again, the old states (vector basis and error vector) will be reused.
- diis_space \ [\text{int}] DIIS space size. By default, 8 Fock matrices and errors vector are stored.
- diis_start_cycle \ [\text{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.


**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 environment.

**conv_check** [bool] An extra cycle to check convergence after SCF iterations.

**check_convergence** [function(envs) => bool] A hook for overloading convergence criteria in 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:**

```python
>>> 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:**

```python
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='cc-pvdz', 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
```

**Gradients**(obj, *args, **kwargs)

Non-relativistic unrestricted Hartree-Fock gradients

**canonicalize**(mf, mo_coeff, mo_occ, fock=None)

Canonicalization diagonalizes the UHF Fock matrix within occupied, virtual subspaces separately (without change occupancy).

**convert_from**(mf)

Create UHF object based on the RHF/ROHF object
**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.

\[
S_{12} = \langle \Psi_A | \Psi_B \rangle = (\det U)(\det V^\dagger) \prod_{i=1}^{2N} \lambda_{ii}
\]

where \( U, V, \lambda \) are unitary matrices and single values generated by single value decomposition (SVD) of the overlap matrix \( O \) which is the overlap matrix of two sets of molecular orbitals:

\[
U^\dagger O V = \Lambda
\]

**Args:**

- **mo1, mo2** [2D ndarrays] Molecular orbital coefficients
- **occ1, occ2** [2D ndarrays] occupation numbers

**Return:**

A list: the product of single values: float x_a, x_b: 1D ndarrays \( U\Lambda^{-1}V^\dagger \) They are used to calculate asymmetric density matrix

**energy_elec** (*mf, dm=None, h1e=None, vhf=None*)

Electronic energy of Unrestricted Hartree-Fock

Note this function has side effects which cause mf.scf_summary updated.

**Returns:** Hartree-Fock electronic energy and the 2-electron part contribution

**get_jk** (*mol=None, dm=None, hermi=1, with_j=True, with_k=True, omega=None*)

Coulomb (J) and exchange (K)

**Args:**

- **dm** [a list of 2D arrays or a list of 3D arrays] \( \alpha, \beta \) or \( \alpha, \alpha, \beta, \beta, \ldots \)

**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 Mol
- **dm** [a list of ndarrays] A list of density matrices, stored as \( \alpha, \alpha, \beta, \beta, \ldots \)

**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: $V_{hf} = (V^\alpha, V^\beta)$. $V^\alpha$ (and $V^\beta$) can be a list matrices, corresponding to the input density matrices.

Examples:

```python
>>> 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] Molecular orbital coefficients
- occ1, occ2: 2D ndarrays Occupation numbers
- x: 2D ndarrays $U\Lambda^{-1}V^\dagger$. See also det_ovlp()

Return: A list of 2D ndarrays for alpha and beta spin

Examples:

```python
>>> 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, **kwargs)

One-particle density matrix

**spin_square** (mo_coeff=None, s=None)

Spin square and multiplicity of UHF determinant

$$S^2 = \frac{1}{2}(S_+S_- + S_-S_+) + S^2_z$$

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 \( S_+ S_- = \sum_i s_i s_{i-} \),

\[
\sum_i \langle \text{UHF} | s_i s_{i-} | \text{UHF} \rangle = \sum_{pq} \langle p | s_+ s_- | q \rangle \gamma_{pq} = n_\alpha
\]

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

\[
\langle S_- S_+ \rangle = -(\langle j^\beta | i^\alpha \rangle \langle j^\alpha | i^\beta \rangle)
\]

2. For \( S_z^2 \)

(a) same electron

\[
\langle s_z^2 \rangle = \frac{1}{4}(n_\alpha + n_\beta)
\]

(a) different electrons

\[
\frac{1}{2} \sum_{ij} \left( \langle ij | 2s_z 2s_z | ij \rangle - \langle ij | 2s_z 2s_z | ji \rangle \right)
\]

\[
= \frac{1}{4} \left( \langle i^\alpha | j^\alpha \rangle \langle j^\alpha | i^\alpha \rangle - \langle i^\beta | j^\beta \rangle \langle j^\beta | i^\beta \rangle - \langle i^\beta | j^\alpha \rangle \langle j^\alpha | i^\beta \rangle + \langle i^\alpha | j^\beta \rangle \langle j^\beta | i^\alpha \rangle \right)
\]

\[
- \frac{1}{4} \left( \langle i^\beta | j^\beta \rangle \langle j^\beta | i^\beta \rangle + \langle i^\alpha | j^\alpha \rangle \langle j^\alpha | i^\alpha \rangle \right)
\]

\[
= \frac{1}{4} \left( n_\alpha^2 - n_\alpha n_\beta - n_\beta n_\alpha + n_\beta^2 \right) - \frac{1}{4} (n_\alpha + n_\beta)
\]

\[
= \frac{1}{4} \left( (n_\alpha - n_\beta)^2 - (n_\alpha + n_\beta) \right)
\]

In total

\[
\langle S_z^2 \rangle = \frac{1}{2} (n_\alpha - \sum_{ij} \langle i^\alpha | j^\beta \rangle \langle j^\beta | i^\alpha \rangle + n_\beta - \sum_{ij} \langle i^\beta | j^\alpha \rangle \langle j^\alpha | i^\beta \rangle) + \frac{1}{4} (n_\alpha - n_\beta)^2
\]

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:
```python
>>> 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
```

**stability** *(internal=True, external=False, verbose=None)*

Stability analysis for RHF/RKS method.

See also pyscf.scf.stability.uhf_stability function.

**Args:** mf : UHF or UKS object

**Kwargs:**

- **internal** [bool] Internal stability, within the UHF space.
- **external** [bool] External stability. Including the UHF -> GHF and real -> complex stability analysis.

**Returns:** New orbitals that are more close to the stable condition. The return value includes two set of orbitals. The first corresponds to the internal stability and the second corresponds to the external stability.

---

**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. Writing to chkfile can be disabled if this attribute is set to None or False.
- **conv_tol** [float] converge threshold. Default is 1e-9
- **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’, ‘hcore’, ‘1e’, ‘chkfile’. Default is ‘minao’
- **DIIS** [DIIS class] The class to generate diis object. It can be one of diis.SCF_DIIS, diis.ADIIS, diis.EDIIS.
- **diis** [boolean or object of DIIS class defined in scf.diis] Default is the object associated to the attribute self.DIIS. Set it to None/False to turn off DIIS. Note if this attribute is inialized as a DIIS object, the SCF driver will use this object in the iteration. The DIIS informations (vector basis and error vector) will be held inside this object. When kernel function is called again, the old states (vector basis and error vector) will be reused.
- **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.


callback [function(enss_dict) => None] callback function takes one dict as the argument which is generated by the built-in function locals(), so that the callback function can access all local variables in the current environment.

conv_check [bool] An extra cycle to check convergence after SCF iterations.

check_convergence [function(enss) => bool] A hook for overloading convergence criteria in 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:

```python
>>> 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
```

**CASCI** *(obj, *args, **kwargs)*

**Args:**

- **mf_or_mol** [SCF object or Mole object] SCF or Mole to define the problem size.
- **ncas** [int] Number of active orbitals.
- **nelecas** [int or a pair of int] Number of electrons in active space.

**Kwargs:**

- **ncore** [int] Number of doubly occupied core orbitals. If not presented, this parameter can be automatically determined.

**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 electron numbers.
natorb [bool] Whether to transform natural orbital in active space. Be cautious of this parameter when CASCI/CASSCF are combined with DMRG solver or selected CI solver because DMRG and selected CI are not invariant to the rotation in active space. False by default.

canonicalization [bool] Whether to canonicalize orbitals. Note that canonicalization does not change the orbitals in active space by default. It only diagonalizes core and external space of the general Fock matrix. To get the natural orbitals in active space, attribute natorb need to be enabled. True by default.

sorting_mo_energy [bool] Whether to sort the orbitals based on the diagonal elements of the general Fock matrix. Default is False.

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 assign the FCISolver to this attribute, e.g.

```python
>>> from pyscf import fci
>>> mc = mcsf.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.:

```python
>>> 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)

e_cas [float] CAS space FCI energy

ci [ndarray] CAS space FCI coefficients

mo_coeff [ndarray] When canonicalization is specified, the orbitals are canonical orbitals which make the general Fock matrix (Fock operator on top of MCSCF 1-particle density matrix) diagonalized within each subspace (core, active, external). If natorb (natural orbitals in active space) is specified, the active segment of the mo_coeff is natural orbitals.

mo_energy [ndarray] Diagonal elements of general Fock matrix (in mo_coeff representation).

Examples:

```python
>>> from pyscf import gto, scf, mcsf
>>> mol = gto.M(atom='N 0 0 0; N 0 0 1', basis='ccpvdz', verbose=0)
>>> mf = scf.RHF(mol)
>>> mf.scf()
>>> mc = mcsf.CASSCI(mf, 6, 6)
>>> mc.kernel()[0]
-108.980200816243354
```

CASSCF (obj, *args, **kwargs)

Args:

- mf_or_mol [SCF object or Mole object] SCF or Mole to define the problem size.
ncas [int] Number of active orbitals.
nelecas [int or a pair of int] Number of electrons in active space.

Kwargs:
ncore [int] Number of doubly occupied core orbitals. If not presented, this parameter can be automatically determined.

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 electron numbers.
natorb [bool] Whether to transform natural orbital in active space. Be cautious of this parameter when CASCI/CASSCF are combined with DMRG solver or selected CI solver because DMRG and selected CI are not invariant to the rotation in active space. False by default.
canonicalization [bool] Whether to canonicalize orbitals. Note that canonicalization does not change the orbitals in active space by default. It only diagonalizes core and external space of the general Fock matrix. To get the natural orbitals in active space, attribute natorb need to be enabled. True by default.
sorting_mo_energy [bool] Whether to sort the orbitals based on the diagonal elements of the general Fock matrix. Default is False.
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 assign the FCISolver to this attribute, e.g.

```python
>>> 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.:

```python
>>> 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)
e_cas [float] CAS space FCI energy

1.11. scf — Mean-field methods

151
**mo_coeff** [ndarray] When canonicalization is specified, the orbitals are canonical orbitals which make the general Fock matrix (Fock operator on top of MCSCF 1-particle density matrix) diagonalized within each subspace (core, active, external). If natorb (natural orbitals in active space) is specified, the active segment of the mo_coeff is natural orbitals.

**mo_energy** [ndarray] Diagonal elements of general Fock matrix (in mo_coeff representation).

Examples:

```python
>>> 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 preferred. 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 allowed 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** [float, 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.

```
>>> 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)
```
```python
>>> mc.conv_tol = 1e-10
>>> mc.ah_conv_tol = 1e-5
>>> mc.kernel()[0]
-109.044401898486001
>>> mc.ah_conv_tol = 1e-10
>>> mc.kernel()[0]
-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 environment.

scale_restoration [float] When a step of orbital rotation moves out of trust region, the orbital optimization will be restored to previous state and the step size of the orbital rotation needs to be reduced. scale_restoration controls how much to scale down the step size.

Saved results

- **e_tot** [float] Total MCSCF energy (electronic energy plus nuclear repulsion)
- **e_cas** [float] CAS space FCI energy
- **ci** [ndarray] CAS space FCI coefficients
- **mo_coeff** [ndarray] Optimized CASSCF orbitals coefficients. When canonicalization is specified, the returned orbitals make the general Fock matrix (Fock operator on top of MCSCF 1-particle density matrix) diagonalized within each subspace (core, active, external). If natorb (natural orbitals in active space) is specified, the active segment of the mo_coeff is natural orbitals.
- **mo_energy** [ndarray] Diagonal elements of general Fock matrix (in mo_coeff representation).

Examples:

```python
>>> from pyscf import gto, scf, mcscf
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]
-109.044401887945668
```

Gradients *(obj, *args, **kwargs)*

Non-relativistic restricted Hartree-Fock gradients

TDA *(obj, *args, **kwargs)*

Tamm-Dancoff approximation

Attributes:

- **conv_tol** [float] Diagonalization convergence tolerance. Default is 1e-9.
- **nstates** [int] Number of TD states to be computed. Default is 3.

Saved results:

- **converged** [bool] Diagonalization converged or not
**TDHF** *(obj, *args, **kwargs)*

Time-dependent Hartree-Fock

**Attributes:**

- `nstates` [int] Number of TD states to be computed. Default is 3.

**Saved results:**

- `converged` [bool] Diagonalization converged or not
- `e` [1D array] excitation energy for each excited state.
- `xy` [A list of two 2D arrays] The two 2D arrays are Excitation coefficients X (shape [nocc, nvir]) and de-excitation coefficients Y (shape [nocc, nvir]) for each excited state. (X,Y) are normalized to 1/2 in RHF/RKS methods and normalized to 1 for UHF/UKS methods. In the TDA calculation, Y = 0.

**convert_from_(mf)**

Convert the input mean-field object to RHF/ROHF

**get_jk** *(mol=None, dm=None, hermi=1, with_j=True, with_k=True, omega=None)*

Compute J, K matrices for all input density matrices

**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 : not hermitian and not symmetric
  1 : hermitian or symmetric
  2 : anti-hermitian

- `vhfopt` : A class which holds precomputed quantities to optimize the computation of J, K matrices
- `with_j` [boolean] Whether to compute J matrices
- `with_k` [boolean] Whether to compute K matrices
- `omega` [float] Parameter of range-separated Coulomb operator: erf(omega * r12 ) / r12. If specified, integration are evaluated based on the long-range part of the range-separated Coulomb operator.

**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:**

```python
>>> from pyscf import gto, scf
>>> from pyscf.scf import _vhf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
```
```python
>>> 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)
```

def 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

class vfopt : 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:
```python
>>> 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
```

def spin_square(mo_coeff=None, s=None)
Spin square and multiplicity of RHF determinant

stability (internal=True, external=False, verbose=None)
RHF/RKS stability analysis.

See also pyscf.scf.stability.rhf_stability function.

Kwargs:

internal [bool] Internal stability, within the RHF optimization space.


Returns: New orbitals that are more close to the stable condition. The return value includes two set of orbitals. The first corresponds to the internal stability and the second corresponds to the external stability.
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. Writing to chkfile can be disabled if this attribute is set to None or False.
    conv_tol  [float] converge threshold. Default is 1e-9
    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’, ‘hcore’, ‘1e’, ‘chkfile’. Default is ‘minao’
    DIIS  [DIIS class] The class to generate diis object. It can be one of diis.SCF_DIIS, diis.ADIIS, diis.EDIIS.
    diis  [boolean or object of DIIS class defined in scf.diis.] Default is the object associated to the attribute self.DIIS. Set it to None/False to turn off DIIS. Note if this attribute is initialized as a DIIS object, the SCF driver will use this object in the iteration. The DIIS informations (vector basis and error vector) will be held inside this object. When kernel function is called again, the old states (vector basis and error vector) will be reused.
    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.
    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 environment.
    conv_check  [bool] An extra cycle to check convergence after SCF iterations.
    check_convergence  [function(envs) => bool] A hook for overloading convergence criteria in 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:
```python
>>> 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, with_meta_lowdin=True, **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:

```python
>>> 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 separately (without change occupancy).

**dip_moment** *(mol=None, dm=None, unit='Debye', verbose=3, **kwargs)*

Dipole moment calculation

\[
\begin{align*}
\mu_x &= -\sum_{\mu} \sum_{\nu} P_{\mu\nu}(\nu|x|\mu) + \sum_{A} Q_{A}X_{A} \\
\mu_y &= -\sum_{\mu} \sum_{\nu} P_{\mu\nu}(\nu|y|\mu) + \sum_{A} Q_{A}Y_{A} \\
\mu_z &= -\sum_{\mu} \sum_{\nu} P_{\mu\nu}(\nu|z|\mu) + \sum_{A} Q_{A}Z_{A}
\end{align*}
\]

where \(\mu_x, \mu_y, \mu_z\) 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:

\[
E = \sum_{ij} h_{ij} \gamma_{ji} + \frac{1}{2} \sum_{ijkl} \gamma_{ji} \gamma_{lk} \langle ik||jl\rangle
\]

Note this function has side effects which cause mf.scf_summary updated.

**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:

```python
>>> 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)
>>> mf.energy_elec(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

Note this function has side effects which cause mf.scf_summary updated.

from_chk \( (chkfile=None, project=None) \)

Read the HF results from checkpoint file, then project it to the basis defined by mol

**Returns:** Density matrix, 2D ndarray

get_fock \( (mf, h1e=None, s1e=None, vhf=None, dm=None, cycle=-1, diis=None, diis_start_cycle=None, level_shift_factor=None, damp_factor=None) \)

\[ F = h^{\text{core}} + V^{\text{HF}} \]

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)

**Kwargs:**

- **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
- **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 orbital gradients

**Args:**

- **mo_coeff** [2D ndarray] Orbital coefficients
- **mo_occ** [1D ndarray] Orbital occupancy
- **fock** [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, omega=None)*

Compute J matrices for all input density matrices

**get_jk** *(mol=None, dm=None, hermi=1, with_j=True, with_k=True, omega=None)*

Compute J, K matrices for all input density matrices

**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 : not hermitian and not symmetric
  - 1 : hermitian or symmetric
  - 2 : anti-hermitian

- **vhfopt** : A class which holds precomputed quantities to optimize the computation of J, K matrices
- **with_j** [boolean] Whether to compute J matrices
- **with_k** [boolean] Whether to compute K matrices
- **omega** [float] Parameter of range-separated Coulomb operator: `erf(omega * r12)/r12`. If specified, integration are evaluated based on the long-range part of the range-separated Coulomb operator.

**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:**

```python
>>> 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, omega=None)*

Compute K matrices for all input density matrices

**get_occ** *(mf, mo_energy=None, mo_coeff=None)*

Label the occupancies for each orbital
Kwargs:

- **mo_energy** [1D ndarray] Orbital energies
- **mo_coeff** [2D ndarray] Orbital coefficients

Examples:

```python
>>> 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:

```python
>>> 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_le** (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=None)
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:
```python
>>> 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(*args, **kwargs)
SCF main driver

Kwargs:

dm0 [ndarray] If given, it will be used as the initial guess density matrix

Examples:
```python
>>> 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, **kwargs)
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 carried 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

Returns: A list : pop, charges

pop  [narray] Mulliken population on each atomic orbitals
charges  [narray] Mulliken charges

mulliken_pop (mol=None, dm=None, s=None, verbose=5) Mulliken population analysis

\[ M_{ij} = D_{ij} S_{ji} \]

Mulliken charges

\[ \delta_i = \sum_j M_{ij} \]

Returns: A list : pop, charges

pop  [narray] Mulliken population on each atomic orbitals
charges  [narray] Mulliken charges

mulliken_pop_meta_lowdin_ao (*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 carried out within each subsets.

Args: mol : an instance of Mole
dm  [narray or 2-item list of narray] 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

Returns: A list : pop, charges

pop  [narray] Mulliken population on each atomic orbitals
charges  [narray] Mulliken charges

nuc_grad_method () Hook to create object for analytical nuclear gradients.

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 carried out within each subsets.

Args: mol : an instance of Mole
dm  [narray or 2-item list of narray] 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

**Returns:**
A list : pop, charges

- **pop** [ndarray] Mulliken population on each atomic orbitals
- **charges** [ndarray] Mulliken charges

**reset** *(mol=None)*
Reset mol and clean up relevant attributes for scanner mode

**scf** *(dm0=None, **kwargs)*
SCF main driver

**Kwargs:**
- **dm0** [ndarray] If given, it will be used as the initial guess density matrix

**Examples:**

```python
>>> 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
```

**to_ghf** *(mf)*
Convert the input mean-field object to a GHF/GKS object

**to_rhf** *(mf)*
Convert the input mean-field object to a RHF/ROHF/RKS/ROKS object

**to_uhf** *(mf)*
Convert the input mean-field object to a UHF/UKS object

**update** *(chkfile=None)*
Read attributes from the chkfile then replace the attributes of current object. It’s an alias of function `update_from_chk_`

**update_** *(chkfile=None)*
Read attributes from the chkfile then replace the attributes of current object. It’s an alias of function `update_from_chk_`

**update_from_chk** *(chkfile=None)*
Read attributes from the chkfile then replace the attributes of current object. It’s an alias of function `update_from_chk_`
Pyscf documentation:

- `pyscf.scf.hf.analyze(mf, verbose=5, with_meta_lowdin=True, **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:`
  ```python
  >>> 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 separately (without change occupancy).

- `pyscf.scf.hf.dip_moment(mol, dm, unit='Debye', verbose=3, **kwargs)`
  Dipole moment calculation
  \[
  \mu_x = -\sum_\mu \sum_\nu P_{\mu\nu} \langle \nu | x | \mu \rangle + \sum_A \sum_\nu Q_A X_A \\
  \mu_y = -\sum_\mu \sum_\nu P_{\mu\nu} \langle \nu | y | \mu \rangle + \sum_A \sum_\nu Q_A Y_A \\
  \mu_z = -\sum_\mu \sum_\nu P_{\mu\nu} \langle \nu | z | \mu \rangle + \sum_A \sum_\nu Q_A Z_A 
  \]
  where \( \mu_x, \mu_y, \mu_z \) 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, with_j=True, with_k=True)`
  Compute J, K matrices in terms of the given 2-electron integrals and density matrix:
  \( J \sim \text{numpy.einsum}('pqrs,qp->rs', \text{eri}, \text{dm}) \)
  \( K \sim \text{numpy.einsum}('pqrs,qr->ps', \text{eri}, \text{dm}) \)
  Args:
  - eri [ndarray] 8-fold or 4-fold ERIs or 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
  Kwarg:
  - 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:**

```python
>>> 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:

\[ E = \sum_{ij} h_{ij} \gamma_{ji} + \frac{1}{2} \sum_{ijkl} \gamma_{ji} \gamma_{lk} \langle ik||jl\rangle \]

Note this function has side effects which cause mf.scf_summary updated.

**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:**

```python
>>> 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
Note this function has side effects which cause mf.scf_summary updated.

```python
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)
```

\[
F = h^{\text{core}} + V^{\text{HF}}
\]

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)

**Kwargs:**
- **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
- **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.

```python
pyscf.scf.hf.get_grad(mo_coeff, mo_occ, fock_ao)
```

RHF orbital gradients

**Args:**
- **mo_coeff** [2D ndarray] Orbital 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.

```python
pyscf.scf.hf.get_hcore(mol)
```

Core Hamiltonian

**Examples:**

```python
>>> 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]])
```

```python
pyscf.scf.hf.get_init_guess(mol, key='minao')
```

Generate density matrix for initial guess

**Kwargs:**

```python
pyscf.scf.hf.get_jk(mol, dm, hermi=1, vhfopt=None, with_j=True, with_k=True, omega=None)
```

Compute J, K matrices for all input density matrices

**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 : not hermitian and not symmetric
1 : hermitian or symmetric
2 : anti-hermitian

**vhfopt**: A class which holds precomputed quantities to optimize the computation of J, K matrices

**with_j** [boolean] Whether to compute J matrices

**with_k** [boolean] Whether to compute K matrices

**omega** [float] Parameter of range-separated Coulomb operator: \( \text{erf}(\omega \cdot r_{12}) / r_{12} \). If specified, integration are evaluated based on the long-range part of the range-separated Coulomb operator.

**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**:

```python
>>> 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] Orbital energies
- **mo_coeff** [2D ndarray] Orbital coefficients

**Examples**:

```python
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 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
vho : A class which holds precomputed quantities to optimize the computation of J, K matrices

Returns: matrix Vhf = 2J - K. Vhf can be a list matrices, corresponding to the input density matrices.

Examples:

```python
>>> 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=None)
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:

```python
>>> 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] mf object holds all parameters to control SCF. One can modify its member functions 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_veff
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 environment.

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:

```python
>>> 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 $\Delta$ to virtual orbitals

$$FC = SCE$$  

$$F = F + SAC^dS$$  

$$\Lambda_{ij} = \begin{cases} 
\delta_{ij}\Delta & i \in \text{virtual} \\
0 & \text{otherwise} 
\end{cases}$$

Returns: New Fock matrix, 2D ndarray
pyscf.scf.hf.make_rdm1 (mo_coeff, mo_occ, **kwargs)

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 carried 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

Returns: A list : pop, charges

pop [nparray] Mulliken population on each atomic orbitals
charges [nparray] Mulliken charges

pyscf.scf.hf.mulliken_pop (mol, dm, s=None, verbose=5)

Mulliken population analysis

\[ M_{ij} = D_{ij} S_{ji} \]

Mulliken charges

\[ \delta_i = \sum_j M_{ij} \]

Returns: A list : pop, charges

pop [nparray] Mulliken population on each atomic orbitals
charges [nparray] Mulliken charges

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 carried 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

**Returns:** A list : pop, charges

- **pop** [narray] Mulliken population on each atomic orbitals
- **charges** [narray] Mulliken charges

```python
pyscf.scf.hf.pack_uniq_var(x, mo_occ)
```
Extract the unique variables from the full orbital-gradients (or orbital-rotation) matrix

```python
pyscf.scf.hf.uniq_var_indices(mo_occ)
```
Indicies of the unique variables for the orbital-gradients (or orbital-rotation) matrix.

```python
pyscf.scf.hf.unpack_uniq_var(dx, mo_occ)
```
Fill the full orbital-gradients (or orbital-rotation) matrix with the unique variables.

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. Writing to chkfile can be disabled if this attribute is set to None or False.
- **conv_tol** [float] converge threshold. Default is 1e-9
- **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’, ‘hcore’, ‘1e’, ‘chkfile’. Default is ‘minao’
- **DIIS** [DIIS class] The class to generate diis object. It can be one of diis.SCF_DIIS, diis.ADIIS, diis.EDIIS.
- **diis** [boolean or object of DIIS class defined in scf.diis.] Default is the object associated to the attribute self.DIIS. Set it to None/False to turn off DIIS. Note if this attribute is inialized as a DIIS object, the SCF driver will use this object in the iteration. The DIIS informations (vector basis and error vector) will be held inside this object. When kernel function is called again, the old states (vector basis and error vector) will be reused.
- **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.
**callback** [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the built-in function `locals()`, so that the callback function can access all local variables in the current environment.

**conv_check** [bool] An extra cycle to check convergence after SCF iterations.

**check_convergence** [function(envs) => bool] A hook for overloading convergence criteria in 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:**

```python
>>> 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:**

```python
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='cc-pvdz', 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
```

**Gradients** *(obj, *args, **kwargs)*

Non-relativistic unrestricted Hartree-Fock gradients

**canonicalize** *(mf, mo_coeff, mo_occ, fock=None)*

Canonicalization diagonalizes the UHF Fock matrix within occupied, virtual subspaces separately (without change occupancy).

**convert_from_** *(mf)*

Create UHF object based on the RHF/ROHF object

**det_ovlp** *(mol1, mo2, occ1, occ2, ovlp=None)*

Calculate the overlap between two different determinants. It is the product of single values of molecular
orbital overlap matrix.

\[ S_{12} = \langle \Psi_A | \Psi_B \rangle = (\det U)(\det V^\dagger) \prod_{i=1}^{2N} \lambda_{ii} \]

where \( U, V, \lambda \) are unitary matrices and single values generated by single value decomposition (SVD) of the overlap matrix \( O \) which is the overlap matrix of two sets of molecular orbitals:

\[ U^\dagger O V = \Lambda \]

**Args:**

- `mo1, mo2` [2D ndarrays] Molecular orbital coefficients
- `occ1, occ2`: 2D ndarrays occupation numbers

**Return:**

A list: the product of single values: float \( x_a, x_b \): 1D ndarrays \( U\Lambda^{-1}V^\dagger \) They are used to calculate asymmetric density matrix

**energy_elec** *(mf, dm=None, h1e=None, vhf=None)*

Electronic energy of Unrestricted Hartree-Fock

Note this function has side effects which cause `mf.scf_summary` updated.

**Returns:** Hartree-Fock electronic energy and the 2-electron part contribution

**get_jk** *(mol=None, dm=None, hermi=1, with_j=True, with_k=True, omega=None)*

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

\[
V_{ij}^\alpha = \sum_{kl} (ij|kl)(\gamma_{ik}^\alpha + \gamma_{lk}^\alpha) - \sum_{kl} (il|kj)\gamma_{lk}^\alpha \\
V_{ij}^\beta = \sum_{kl} (ij|kl)(\gamma_{ik}^\beta + \gamma_{lk}^\beta) - \sum_{kl} (il|kj)\gamma_{lk}^\beta
\]

**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: $V_{hf} = (V^\alpha, V^\beta)$. $V^\alpha$ (and $V^\beta$) can be a list matrices, corresponding to the input density matrices.

Examples:

```python
>>> 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] Molecular orbital coefficients

$occ1$, $occ2$: 2D ndarrays Occupation numbers

$x$: 2D ndarrays $U\Lambda^{-1}V^\dagger$. See also `det_ovlp()`

Return: A list of 2D ndarrays for alpha and beta spin

Examples:

```python
>>> 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` ($mo$=None, $mo_occ$=None, **kwargs)

One-particle density matrix

Returns: A list of 2D ndarrays for alpha and beta spins

`spin_square` ($mo$=None, $s$=None)

Spin square and multiplicity of UHF determinant

$$S^2 = \frac{1}{2}(S_+S_- + S_-S_+) + S_z^2$$

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,
\[
\langle S_+S_- \rangle = \langle ij|s_{+}s_{-}|ij\rangle - \langle ij|s_{+}s_{-}|ji\rangle = -(i^\alpha|j^\beta)(j^\beta|i^\alpha)
\]

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
\[
\frac{1}{2} \sum_{ij} \left( \langle ij|2s_{z1}s_{z2}|ij\rangle - \langle ij|2s_{z1}s_{z2}|ji\rangle \right)
= \frac{1}{4} \left( (i^\alpha|i^\alpha)(j^\alpha|j^\alpha) - (i^\alpha|i^\beta)(j^\beta|j^\alpha) - (i^\beta|i^\alpha)(j^\alpha|j^\beta) + (i^\beta|i^\beta)(j^\beta|j^\beta) \right)
- \frac{1}{4} \left( (i^\alpha|j^\alpha)(j^\alpha|i^\alpha) + (i^\beta|j^\beta)(j^\beta|i^\beta) \right)
= \frac{1}{4}(n_\alpha^2 - n_\alpha n_\beta - n_\beta n_\alpha + n_\beta^2) - \frac{1}{4}(n_\alpha + n_\beta)
= \frac{1}{4}((n_\alpha - n_\beta)^2 - (n_\alpha + n_\beta))
\]

In total
\[
\langle S_z^2 \rangle = \frac{1}{2}(n_\alpha - \sum_{ij} (i^\alpha|j^\beta)(j^\beta|i^\alpha) + n_\beta - \sum_{ij} (i^\beta|j^\alpha)(j^\alpha|i^\beta)) + \frac{1}{4}(n_\alpha - n_\beta)^2
\]

**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:**
```python
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz', charge=1, orbital=False, verbose=0)
>>> mf = scf.UHF(mol)
>>> mf.kernel()
-75.623975516256706
```

11. scf — Mean-field methods

PySCF Documentation, Release 1.7.0

- 1.11. scf — Mean-field methods

175
```python
>>> 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
```

**stability**  
(*internal=True, external=False, verbose=None*)

Stability analysis for RHF/RKS method.

See also pyscf.scf.stability.uhf_stability function.

**Args:**  
mf : UHF or UKS object

**Kwarg:**

*internal* [bool] Internal stability, within the UHF space.

*external* [bool] External stability. Including the UHF -> GHF and real -> complex stability analysis.

**Returns:** New orbitals that are more close to the stable condition. The return value includes two set of orbitals. The first corresponds to the internal stability and the second corresponds to the external stability.

**pyscf.scf.uhf.analyze**  
(*mf, verbose=5, with_meta_lowdin=True, **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.

\[
S_{12} = \langle \Psi_A | \Psi_B \rangle = (\det U)(\det V^\dagger) \prod_{i=1}^{2N} \lambda_{ii}
\]

where \( U, V, \lambda \) are unitary matrices and single values generated by single value decomposition(SVD) of the overlap matrix \( O \) which is the overlap matrix of two sets of molecular orbitals:

\[
U^\dagger O V = \Lambda
\]

**Args:**

*mo1, mo2* [2D ndarrays] Molecular orbital coefficients

*occ1, occ2*: 2D ndarrays occupation numbers

**Return:**

A list: the product of single values: float \( x_a, x_b \): 1D ndarrays \( U\Lambda^{-1}V^\dagger \) They are used to calculate asymmetric density matrix

**pyscf.scf.uhf.energy_elec**  
(*mf, dm=None, h1e=None, vhf=None*)

Electronic energy of Unrestricted Hartree-Fock

Note this function has side effects which cause mf.scf_summary updated.

**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: \(V_{hf} = (V^\alpha, V^\beta)\). \(V^\alpha\) (and \(V^\beta\)) can be a list matrices, corresponding to the input density matrices.

Examples:

```python
>>> 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_chkfile(mol, chkfile_name, project=None)

Read SCF chkfile and make the density matrix for UHF initial guess.

Kwargs:
- project [None or bool] Whether to project chkfile’s orbitals to the new basis. Note when the geometry of the chkfile and the given molecule are very different, this projection can produce very poor initial guess. In PES scanning, it is recommended to switch off project.

If project is set to None, the projection is only applied when the basis sets of the chkfile’s molecule are different to the basis sets of the given molecule (regardless whether the geometry of the two molecules are different). Note the basis sets are considered to be different if the two molecules are derived from the same molecule with different ordering of atoms.
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] Molecular orbital coefficients
- **occ1**, **occ2**: 2D ndarrays Occupation numbers
- **x**: 2D ndarrays \( U\Lambda^{-1}V^\dagger \). See also det_ovlp()

**Return:** A list of 2D ndarrays for alpha and beta spin

**Examples:**
```python
>>> 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, **kwargs)
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.spin_square (mo, s=1)
Spin square and multiplicity of UHF determinant

\[
S^2 = \frac{1}{2}(S_+S_- + S_-S_+) + S_+^2
\]

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

$$\frac{1}{2} \sum_{ij} \left( \langle ij | 2s_{z1} s_{z2} | ij \rangle - \langle ij | 2s_{z1} s_{z2} | ji \rangle \right)$$

$$= \frac{1}{4} \left( \langle i^\alpha | i^\alpha \rangle \langle j^\alpha | j^\alpha \rangle - \langle i^\alpha | j^\alpha \rangle \langle j^\alpha | i^\alpha \rangle - \langle i^\beta | i^\beta \rangle \langle j^\beta | j^\beta \rangle + \langle i^\beta | j^\beta \rangle \langle j^\beta | i^\beta \rangle \right)$$

$$- \frac{1}{4} \left( \langle i^\alpha | j^\alpha \rangle \langle j^\alpha | i^\alpha \rangle + \langle i^\beta | j^\beta \rangle \langle j^\beta | i^\beta \rangle \right)$$

$$= \frac{1}{4} (n_\alpha^2 - n_\alpha n_\beta + n_\beta n_\alpha + n_\beta^2) - \frac{1}{4} (n_\alpha + n_\beta)$$

$$= \frac{1}{4} (n_\alpha - n_\beta)^2 - (n_\alpha + n_\beta)$$

In total

$$\langle S_z^2 \rangle = \frac{1}{2} (n_\alpha - \sum_{ij} \langle i^\alpha | j^\beta \rangle \langle j^\beta | i^\alpha \rangle + n_\beta - \sum_{ij} \langle i^\beta | j^\alpha \rangle \langle j^\alpha | i^\beta \rangle) + \frac{1}{4} (n_\alpha - n_\beta)^2$$

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:

```python
gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz', charge=1, spin=1, verbose=0)  # mol
mf = scf.UHF(mol)  # mf
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.SymAdaptedRHF(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. Writing to chkfile can be disabled if this attribute is set to None or False.
    conv_tol  [float] converge threshold. Default is 1e-9
    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’, ‘hcore’, ‘1e’, ‘chkfile’. Default is ‘minao’
    DIIS  [DIIS class] The class to generate diis object. It can be one of diis.SCF_DIIS, diis.ADIIS, diis.EDIIS.
    diis  [boolean or object of DIIS class defined in scf.diis.] Default is the object associated to the attribute self.DIIS. Set it to None/False to turn off DIIS. Note if this attribute is initialized as a DIIS object, the SCF driver will use this object in the iteration. The DIIS informations (vector basis and error vector) will be held inside this object. When kernel function is called again, the old states (vector basis and error vector) will be reused.
    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.
    callback  [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the built-in function locals(), so that the callback function can access all local variables in the current environment.
    conv_check  [bool] An extra cycle to check convergence after SCF iterations.
    check_convergence  [function(envs) => bool] A hook for overloading convergence criteria in 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:

```python
>>> 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:

```python
>>> 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}
```

**analyze** *(verbose=None, with_meta_lowdin=True, **kwargs)*

Analyze the given SCF object: print orbital energies, occupancies; print orbital coefficients; Occupancy for each irreps; Mulliken population analysis

**canonicalize** *(mf, mo_coeff, mo_occ, fock=None)*

Canonicalization diagonalizes the Fock matrix in occupied, open, virtual subspaces separately (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()
mf.scf()
-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.HF_Symmetry(Mol)

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. Writing to chkfile can be disabled if this attribute is set to None or False.

cov_tol [float] converge threshold. Default is 1e-9

cov_tol_grad [float] gradients converge threshold. Default is sqrt(cov_tol)

max_cycle [int] max number of iterations. Default is 50

init_guess [str] initial guess method. It can be one of ‘minao’, ‘atom’, ‘hcore’, ‘1e’, ‘chkfile’. Default is ‘minao’

DIIS [DIIS class] The class to generate diis object. It can be one of diis.SCF_DIIS, diis.ADIIS, diis.EDIIS.

diis [boolean or object of DIIS class defined in scf.diis.] Default is the object associated to the attribute self.DIIS. Set it to None/False to turn off DIIS. Note if this attribute is inialized as a DIIS object, the SCF driver will use this object in the iteration. The DIIS informations (vector basis and error vector) will be held inside this object. When kernel function is called again, the old states (vector basis and error vector) will be reused.

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.


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 environment.

conv_check [bool] An extra cycle to check convergence after SCF iterations.

check_convergence [function(envs) => bool] A hook for overloading convergence criteria in SCF iterations.
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:

```python
>>> 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:

```python
>>> 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 separately (without change occupancy).

eig (fock, 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, with_meta_lowdin=True, **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 separately (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:

```python
>>> 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.SymAdaptedUHF(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. Writing to chkfile can be disabled if this attribute is set to None or False.
- **conv_tol** [float] converge threshold. Default is 1e-9
- **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’, ‘hcore’, ‘1e’, ‘chkfile’. Default is ‘minao’
- **DIIS** [DIIS class] The class to generate diis object. It can be one of diis.SCF_DIIS, diis.ADIIS, diis.EDIIS.
- **diis** [boolean or object of DIIS class defined in scf.diis.] Default is the object associated to the attribute self.DIIS. Set it to None/False to turn off DIIS. Note if this attribute is inialized as a DIIS object, the SCF driver will use this object in the iteration. The DIIS informations (vector basis and error vector) will be held inside this object. When kernel function is called again, the old states (vector basis and error vector) will be reused.
- **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.
callback [function(envs_dict) => None] callback function takes one dict as the argument which
is generated by the built-in function locals(), so that the callback function can access
all local variables in the current environment.
conv_check [bool] An extra cycle to check convergence after SCF iterations.
check_convergence [function(envs) => bool] A hook for overloading convergence
criteria in 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:

```python
gto.M(atom='H 0 0 0; H 0 0 1.1', basis='cc-pvdz')
scf.hf.SCF(mol)
verbose = 0
level_shift = .4
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:

```python
gto.M(atom='H 0 0 0; H 0 0 1.1', basis='cc-pvdz', charge=1, spin=1, verbose=0)
scf.UHF(mol)
kernel()
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:
```python
>>> 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 separately (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:
```python
>>> 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.

canonicalize(mf, mo_coeff, mo_occ, fock=None)
Canonicalization diagonalizes the UHF Fock matrix in occupied, virtual subspaces separately (without change occupancy).

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:

\textbf{irrep\_nelec} [dict] The number of alpha/beta electrons for each irrep \{'ir\_name':(int,int), ...\}.

Examples:

```python
>>> 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)}
```

---

Restricted coupled perturbed Hartree-Fock solver

```
pyscf.scf.cphf.kernel (fvind, mo_energy, mo_occ, h1, s1=None, max_cycle=20, tol=1e-09, hermi=False, verbose=2)
```

\textbf{Args:}

fvind [function] Given density matrix, compute \((ij|kl)D_{lk}*2 - (ij|kl)D_{jk}\)

\textbf{Kwargs:}

hermi [boolean] Whether the matrix defined by fvind is Hermitian or not.

```
pyscf.scf.cphf.solve (fvind, mo_energy, mo_occ, h1, s1=None, max_cycle=20, tol=1e-09, hermi=False, verbose=2)
```

\textbf{Args:}

fvind [function] Given density matrix, compute \((ij|kl)D_{lk}*2 - (ij|kl)D_{jk}\)

\textbf{Kwargs:}

hermi [boolean] Whether the matrix defined by fvind is Hermitian or not.

```
pyscf.scf.cphf.solve_nos1 (fvind, mo_energy, mo_occ, h1, max_cycle=20, tol=1e-09, hermi=False, verbose=2)
```

For field independent basis. First order overlap matrix is zero

```
pyscf.scf.cphf.solve_withs1 (fvind, mo_energy, mo_occ, h1, s1, max_cycle=20, tol=1e-09, hermi=False, verbose=2)
```

For field dependent basis. First order overlap matrix is non-zero. The first order orbitals are set to \(C^1_{ij} = -1/2 \ S1 e_1 = h_1 - s1^*e0 + (e0_\cdot j - e0_\cdot i)*c1 + vhf[c1]\)

\textbf{Kwargs:}

hermi [boolean] Whether the matrix defined by fvind is Hermitian or not.

\textbf{Returns:} First order orbital coefficients (in MO basis) and first order orbital energy matrix

---

Non-relativistic generalized Hartree-Fock

```
class pyscf.scf.ghf.GHF (mol)
```

SCF base class. non-relativistic RHF.

\textbf{Attributes:}

\textbf{verbose} [int] Print level. Default value equals to \texttt{Mole.verbose}

---

1.11. scf — Mean-field methods
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. Writing to chkfile can be disabled if this attribute is set to None or False.

conv_tol  [float] converge threshold. Default is 1e-9

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’, ‘hcore’, ‘1e’, ‘chkfile’. Default is ‘minao’

DIIS  [DIIS class] The class to generate diis object. It can be one of diis.SCf_DIIS, diis.ADIIS, diis.EDIIS.

diis  [boolean or object of DIIS class defined in scf.diis.] Default is the object associated to the attribute self.DIIS. Set it to None/False to turn off DIIS. Note if this attribute is inialized as a DIIS object, the SCF driver will use this object in the iteration. The DIIS informations (vector basis and error vector) will be held inside this object. When kernel function is called again, the old states (vector basis and error vector) will be reused.

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.


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 environment.

conv_check  [bool] An extra cycle to check convergence after SCF iterations.

check_convergence  [function(envs) => bool] A hook for overloading convergence criteria in 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:

```python
>>> 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 GHF method**  
GHF orbital coefficients are 2D array. Let \( \text{nao} \) be the number of spatial AOs, \( \text{mo\_coeff}[:\text{nao}] \) are the coefficients of AO with alpha spin; \( \text{mo\_coeff}[:\text{nao*2}] \) are the coefficients of AO with beta spin.

**convert_from_(mf)**  
Create GHF object based on the RHF/UHF object

**det_ovlp**(\( \text{mo1}, \text{mo2}, \text{occ1}, \text{occ2}, \text{ovlp}=\text{None} \))  
Calculate the overlap between two different determinants. It is the product of single values of molecular orbital overlap matrix.

**Return:**  
A list: the product of single values: \( x_a: \mathbf{U}^{-1} \mathbf{V}^\dagger \) They are used to calculate asymmetric density matrix

**get_jk**(\( \text{mol}=\text{None}, \text{dm}=\text{None}, \text{hermi}=0, \text{with}_j=\text{True}, \text{with}_k=\text{True}, \text{omega}=\text{None} \))  
Compute J, K matrices for all input density matrices

**Args:** \( \text{mol} \): an instance of \text{Mole} 
\( \text{dm} \) [ndarray or list of ndarrays] A density matrix or a list of density matrices

**Kwargs:**
- **hermi** [int] Whether J, K matrix is hermitian
  - 0 : not hermitian and not symmetric
  - 1 : hermitian or symmetric
  - 2 : anti-hermitian

- **vhfopt** : A class which holds precomputed quantities to optimize the computation of J, K matrices
- **with_j** [boolean] Whether to compute J matrices
- **with_k** [boolean] Whether to compute K matrices
- **omega** [float] Parameter of range-separated Coulomb operator: \( \text{erf}(\omega \ast \mathbf{r}_{12})/\mathbf{r}_{12} \). If specified, integration are evaluated based on the long-range part of the range-separated Coulomb operator.

**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:**

```python
>>> 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)
```

**init_guess_by_atom**(\( \text{mol}=\text{None} \))  
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=None)*

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:**

```python
>>> 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]])
```

**spin_square** *(mo_coeff=None, s=None)*

Spin of the GHF wavefunction

\[ S^2 = \frac{1}{2}(S_+S_- + S_-S_+) + S_z^2 \]

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 \( S_+S_- = \sum_i s_{i+}s_{i-} \),
     \[
     \sum_i \langle UHF | s_{i+}s_{i-} | UHF \rangle = \sum_{pq} \langle p | s_{i+}s_{i-} | q \rangle \gamma_{pq} = n_\alpha
     \]
   
   2) different electrons \( S_+S_- = \sum_i s_{i+}s_{j-}, (i \neq j) \). There are in total \( n(n-1) \) terms. As a two-particle operator,

   \[
   \langle S_+S_- \rangle = \sum_{ij} \left( \langle i^\alpha | i^\beta \rangle \langle j^\beta | j^\alpha \rangle - \langle i^\alpha | j^\beta \rangle \langle j^\alpha | i^\beta \rangle \right)
   \]

2. **Similarly, for** \( S_-S_+ \)

   - (a) same electron
     \[
     \sum_i \langle s_{i-}s_{i+} \rangle = n_\beta
     \]
   
   - (b) different electrons
     \[
     \langle S_-S_+ \rangle = \sum_{ij} \left( \langle i^\beta | i^\alpha \rangle \langle j^\alpha | j^\beta \rangle - \langle i^\beta | j^\alpha \rangle \langle j^\alpha | i^\beta \rangle \right)
     \]

2. **For** \( S_z^2 \)

   - (a) same electron
     \[
     \langle s_z^2 \rangle = \frac{1}{4}(n_\alpha + n_\beta)
     \]
   
   - (a) different electrons
\[
\sum_{ij} (\langle ij | s_{z1}s_{z2} | ij \rangle - \langle ij | s_{z1}s_{z2} | ji \rangle ) = \frac{1}{4} \sum_{ij} (\langle i^\alpha | j^\alpha \rangle \langle j^\beta | j^\beta \rangle - \langle i^\beta | i^\beta \rangle \langle j^\alpha | j^\alpha \rangle + \langle i^\alpha | i^\alpha \rangle \langle j^\beta | j^\beta \rangle - \langle i^\beta | j^\beta \rangle \langle j^\alpha | i^\alpha \rangle )
\]

\[
= \frac{1}{4} \sum_{ij} (\langle i^\alpha | i^\alpha \rangle \langle j^\alpha | j^\alpha \rangle - \langle i^\alpha | j^\alpha \rangle \langle j^\beta | i^\alpha \rangle - \langle i^\beta | j^\beta \rangle \langle j^\alpha | i^\beta \rangle + \langle i^\beta | i^\beta \rangle \langle j^\alpha | j^\alpha \rangle )
\]

\[
= \frac{1}{4} \left( (\langle i^\alpha | i^\alpha \rangle - \langle i^\beta | j^\beta \rangle )^2 - \frac{1}{4} \sum_{ij} |\langle i^\alpha | j^\alpha \rangle - \langle i^\beta | j^\beta \rangle |^2 \right)
\]

\[
= \frac{1}{4} (n_\alpha - n_\beta)^2 - \frac{1}{4} \sum_{ij} |\langle i^\alpha | j^\alpha \rangle - \langle i^\beta | j^\beta \rangle |^2
\]

**Args:**

- `mo` [a list of 2 ndarrays] Occupied alpha and occupied beta orbitals

**Kwarg:**

- `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:**

```python
>>> 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.ghf.analyze** *(mf, verbose=5, with_meta_lowdin=True, **kwargs)*

Analyze the given SCF object: print orbital energies, occupancies; print orbital coefficients; Mulliken population analysis; Dipole moment

**pyscf.scf.ghf.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.

**Return:**

A list: the product of single values: float x_a: \(U^{\dagger}A^{-1}V\) They are used to calculate asymmetric density matrix

**pyscf.scf.ghf.get_jk** *(mol, dm, hermi=0, with_j=True, with_k=True, jkbuild=<function get_jk>, 
omega=None)*

Compute J, K matrices for all input density matrices

**Args:**

- `mol` : an instance of Mole
- `dm` [ndarray or list of ndarrays] A density matrix or a list of density matrices

**Kwarg:**

- `hermi` [int] Whether J, K matrix is hermitian
0: not hermitian and not symmetric
1: hermitian or symmetric
2: anti-hermitian

\texttt{vhfopt}: A class which holds precomputed quantities to optimize the computation of J, K matrices

\texttt{with\_j} [boolean] Whether to compute J matrices

\texttt{with\_k} [boolean] Whether to compute K matrices

\texttt{omega} [float] Parameter of range-separated Coulomb operator: \( \text{erf}(\omega \ast r_{12}) / r_{12} \). If specified, integration are evaluated based on the long-range part of the range-separated Coulomb operator.

\textbf{Returns:} Depending on the given \( \text{dm} \), 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.

\textbf{Examples:}

```python
>>> 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)
```

\texttt{pyscf.scf.ghf.guess_orbspin} (\texttt{mo\_coeff})

Guess the orbital spin (alpha 0, beta 1, unknown -1) based on the orbital coefficients

\texttt{pyscf.scf.ghf.init_guess_by_chkfile} (\texttt{mol}, \texttt{chkfile\_name}, \texttt{project=None})

Read SCF chkfile and make the density matrix for GHF initial guess.

\textbf{Kwargs:}

\texttt{project} [None or bool] Whether to project chkfile’s orbitals to the new basis. Note when the geometry of the chkfile and the given molecule are very different, this projection can produce very poor initial guess. In PES scanning, it is recommended to switch off project.

If project is set to None, the projection is only applied when the basis sets of the chkfile’s molecule are different to the basis sets of the given molecule (regardless whether the geometry of the two molecules are different). Note the basis sets are considered to be different if the two molecules are derived from the same molecule with different ordering of atoms.

\texttt{pyscf.scf.ghf.mulliken_meta} (\texttt{mol}, \texttt{dm\_ao}, \texttt{verbose=5}, \texttt{pre\_orth\_method='ANO'}, \texttt{s=None})

Mulliken population analysis, based on meta-Lowdin AOs.

\texttt{pyscf.scf.ghf.mulliken_pop} (\texttt{mol}, \texttt{dm}, \texttt{s=None}, \texttt{verbose=5})

Mulliken population analysis

\texttt{pyscf.scf.ghf.spin\_square} (\texttt{mo}, \texttt{s=1})

Spin of the GHF wavefunction

\[ S^2 = \frac{1}{2}(S_+ S_- + S_- S_+) + S_z^2 \]

where \( S_+ = \sum_i s_{i \uparrow} \) is effective for all beta occupied orbitals; \( S_- = \sum_i s_{i \downarrow} \) is effective for all alpha occupied orbitals.

**There are two possibilities for** \( S_+ S_- \)

(a) same electron \( S_+ S_- = \sum_i s_{i \uparrow} s_{i \downarrow} \).
\[ \sum_i \langle UHF | s_i^+ s_i^- | UHF \rangle = \sum_{pq} \langle p | s_i^+ s_i^- | q \rangle \gamma_{qp} = n_\alpha \]

2) different electrons \( S_+ S_- = \sum_{i \neq j} s_i^+ s_j^- \). There are in total \( n(n - 1) \) terms. As a two-particle operator,

\[ \langle S_+ S_- \rangle = \sum_{ij} \left( \langle i^\alpha | i^\beta \rangle \langle j^\beta | j^\alpha \rangle - \langle i^\alpha | j^\beta \rangle \langle j^\beta | i^\alpha \rangle \right) \]

2. Similarly, for \( S_- S_+ \)

(a) same electron

\[ \sum_i \langle s_i^- s_i^+ \rangle = n_\beta \]

(a) different electrons

\[ \langle S_- S_+ \rangle = \sum_{ij} \left( \langle i^\beta | i^\alpha \rangle \langle j^\alpha | j^\beta \rangle - \langle i^\beta | j^\alpha \rangle \langle j^\alpha | i^\beta \rangle \right) \]

2. For \( S^2 \)

(a) same electron

\[ \langle s^2 \rangle = \frac{1}{4}(n_\alpha + n_\beta) \]

(a) different electrons

\[
\sum_{ij} \left( \langle ij | s_{z1} s_{z2} | ij \rangle - \langle ij | s_{z1} s_{z2} | ji \rangle \right) \\
= \frac{1}{4} \sum_{ij} \left( \langle i^\alpha | i^\alpha \rangle \langle j^\alpha | j^\alpha \rangle - \langle i^\alpha | i^\beta \rangle \langle j^\beta | j^\beta \rangle - \langle i^\beta | i^\alpha \rangle \langle j^\alpha | j^\beta \rangle + \langle i^\beta | i^\beta \rangle \langle j^\beta | j^\beta \rangle \right) \\
- \frac{1}{4} \sum_{ij} \left( \langle i^\alpha | j^\alpha \rangle \langle j^\alpha | i^\alpha \rangle - \langle i^\beta | j^\beta \rangle \langle j^\beta | i^\beta \rangle - \langle i^\beta | j^\beta \rangle \langle j^\beta | i^\beta \rangle + \langle i^\beta | j^\beta \rangle \langle j^\beta | i^\beta \rangle \right) \\
= \frac{1}{4} \sum_{ij} \left| \langle i^\alpha | i^\alpha \rangle - \langle i^\beta | i^\beta \rangle \right|^2 \\
= \frac{1}{4} \left( n_\alpha - n_\beta \right)^2 - \frac{1}{4} \sum_{ij} \left| \langle i^\alpha | j^\alpha \rangle - \langle i^\beta | j^\beta \rangle \right|^2 
\]

Args:
- \( \text{mo} \) [a list of 2 ndarrays] Occupied alpha and occupied beta orbitals

Kwargs:
- \( \text{s} \) [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:

```python
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz', charge=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]) 
```

1.11. scf — Mean-field methods
Non-relativistic generalized Hartree-Fock with point group symmetry.

class pyscf.scf.ghf_symm.GHF (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. Writing to chkfile can be disabled if this
   attribute is set to None or False.
   conv_tol  [float] converge threshold. Default is 1e-9
   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’, ‘hcore’, ‘1e’, ‘chkfile’. Default is
   ‘minao’
   DIIS  [DIIS class] The class to generate diis object. It can be one of diis.SCF_DIIS, diis.ADIIS, diis.EDIIS.
   diis  [boolean or object of DIIS class defined in scf.diis.] Default is the object associated to the
   attribute self.DIIS. Set it to None/False to turn off DIIS. Note if this attribute is inialized as a
   DIIS object, the SCF driver will use this object in the iteration. The DIIS informations (vector basis
   and error vector) will be held inside this object. When kernel function is called again, the old states
   (vector basis and error vector) will be reused.
   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.
   callback  [function(envs_dict) => None] callback function takes one dict as the argument which is gener-
   ated by the builtin function locals(), so that the callback function can access all local variables
   in the current environment.
   conv_check  [bool] An extra cycle to check convergence after SCF iterations.
   check_convergence  [function(envs) => bool] A hook for overloading convergence criteria in SCF iter-
   ations.

   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:

```python
cmol = gto.M(atom='H 0 0 0; H 0 0 1.1', basis='cc-pvdz')
cmf = scf.hf.SCF(cmol)
cmf.verbose = 0
cmf.level_shift = .4
cmf.scf()  
-1.0811707843775884
```

**Attributes for GHF method** GHF orbital coefficients are 2D array. Let nao be the number of spatial AOs, mo_coeff[:nao] are the coefficients of AO with alpha spin; mo_coeff[nao:nao*2] are the coefficients of AO with beta spin.

**Attributes for symmetry allowed GHF:**

**irrep_nelec** [dict] Specify the number of electrons for particular irrep {'ir_name':int,...}. For the irreps not listed in these dicts, the program will choose the occupancy based on the orbital energies.

**canonicalize** (mf, mo_coeff, mo_occ, fock=None)
Canonicalization diagonalizes the UHF Fock matrix in occupied, virtual subspaces separately (without change occupancy).

**get_irrep_nelec** (mol=None, mo_coeff=None, mo_occ=None, s=None)
Electron numbers for each irreducible representation.

**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.

```python
cmf.scf.hf_symm.canonicalize(mf, mo_coeff, mo_occ, fock=None)
cmf.scf.hf_symm.get_irrep_nelec(mol, mf.mo_coeff, mf.mo_occ)  
{'A1': 6, 'A2': 0, 'B1': 2, 'B2': 2}
```

Restricted Open-shell Hartree-Fock
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. Writing to chkfile can be disabled if this attribute is set to None or False.
- `conv_tol` [float] converge threshold. Default is 1e-9
- `conv_tol_grad` [float] gradients converge threshold. Default is \(\sqrt{\text{conv}_\text{tol}}\)
- `max_cycle` [int] max number of iterations. Default is 50
- `DIIS` [DIIS class] The class to generate diis object. It can be one of `diis.SCF_DIIS`, `diis.ADIIS`, `diis.EDIIS`.
- `diis` [boolean or object of DIIS class defined in scf.diis] Default is the object associated to the attribute `self.DIIS`. Set it to None/False to turn off DIIS. Note if this attribute is inialized as a DIIS object, the SCF driver will use this object in the iteration. The DIIS informations (vector basis and error vector) will be held inside this object. When kernel function is called again, the old states (vector basis and error vector) will be reused.
- `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.
- `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 environment.
- `conv_check` [bool] An extra cycle to check convergence after SCF iterations.
- `check_convergence` [function(envs) => bool] A hook for overloading convergence criteria in 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

CASSI \((obj, *args, **kwargs)\)

**Args:**
- `mf_or_mol` [SCF object or Mole object] SCF or Mole to define the problem size.
- `ncas` [int] Number of active orbitals.
- `nelecas` [int or a pair of int] Number of electrons in active space.

**Kwargs:**
- `ncore` [int] Number of doubly occupied core orbitals. If not presented, this parameter can be automatically determined.

**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 \((\text{nelec}_\alpha, \text{nelec}_\beta)\)
- `ncore` [int or tuple of int] Core electron number. In UHF-CASSCF, it’s a tuple to indicate the different core electron numbers.
- `natorb` [bool] Whether to transform natural orbital in active space. Be cautious of this parameter when CASSCI/CASSCF are combined with DMRG solver or selected CI solver because DMRG and selected CI are not invariant to the rotation in active space. False by default.
- `canonicalization` [bool] Whether to canonicalize orbitals. Note that canonicalization does not change the orbitals in active space by default. It only diagonalizes core and external space of the general Fock matrix. To get the natural orbitals in active space, attribute `natorb` need to be enabled. True by default.
- `sorting_mo_energy` [bool] Whether to sort the orbitals based on the diagonal elements of the general Fock matrix. Default is False.
- `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 assign the FCISolver to this attribute, e.g.

```python
>>> from pyscf import fci
>>> mc = mcsf.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)
- **e_cas** [float] CAS space FCI energy
- **ci** [ndarray] CAS space FCI coefficients
- **mo_coeff** [ndarray] When canonicalization is specified, the orbitals are canonical orbitals which make the general Fock matrix (Fock operator on top of MCSCF 1-particle density matrix) diagonalized within each subspace (core, active, external). If natorb (natural orbitals in active space) is specified, the active segment of the mo_coeff is natural orbitals.
- **mo_energy** [ndarray] Diagonal elements of general Fock matrix (in mo_coeff representation).

Examples:

```python
>>> 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** *(obj, *args, **kwargs)*

CASSCF

**Args:**

- **mf_or_mol** [SCF object or Mole object] SCF or Mole to define the problem size.
- **ncas** [int] Number of active orbitals.
- **nelecas** [int or a pair of int] Number of electrons in active space.

**Kwargs:**

- **ncore** [int] Number of doubly occupied core orbitals. If not presented, this parameter can be automatically determined.

**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 electron numbers.
- **natorb** [bool] Whether to transform natural orbital in active space. Be cautious of this parameter when CASCI/CASSCF are combined with DMRG solver or selected CI solver because DMRG and selected CI are not invariant to the rotation in active space. False by default.
**canonicalization** [bool] Whether to canonicalize orbitals. Note that canonicalization does not change the orbitals in active space by default. It only diagonalizes core and external space of the general Fock matrix. To get the natural orbitals in active space, attribute natorb need to be enabled. True by default.

**sorting_mo_energy** [bool] Whether to sort the orbitals based on the diagonal elements of the general Fock matrix. Default is False.

**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 assign the FCISolver to this attribute, e.g.

```python
>>> 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.:

```python
>>> 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)

**e_cas** [float] CAS space FCI energy

**ci** [ndarray] CAS space FCI coefficients

**mo_coeff** [ndarray] When canonicalization is specified, the orbitals are canonical orbitals which make the general Fock matrix (Fock operator on top of MCSCF 1-particle density matrix) diagonalized within each subspace (core, active, external). If natorb (natural orbitals in active space) is specified, the active segment of the mo_coeff is natural orbitals.

**mo_energy** [ndarray] Diagonal elements of general Fock matrix (in mo_coeff representation).

**Examples:**

```python
>>> 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]
-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 preferred. 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 allowed 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  [float, 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.

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 built-in function locals(), so that the callback function can access all local variables in the current environment.

scale_restoration  [float] When a step of orbital rotation moves out of trust region, the orbital optimization will be restored to previous state and the step size of the orbital rotation needs to be reduced. scale_restoration controls how much to scale down the step size.

Saved results

e_tot  [float] Total MCSCF energy (electronic energy plus nuclear repulsion)

e_cas  [float] CAS space FCI energy

ci  [ndarray] CAS space FCI coefficients
**mo_coeff** [ndarray] Optimized CASSCF orbitals coefficients. When canonicalization is specified, the returned orbitals make the general Fock matrix (Fock operator on top of MCSCF 1-particle density matrix) diagonalized within each subspace (core, active, external). If natorb (natural orbitals in active space) is specified, the active segment of the mo_coeff is natural orbitals.

**mo_energy** [ndarray] Diagonal elements of general Fock matrix (in mo_coeff representation).

Examples:

```python
>>> 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
```

Gradients *(obj, *args, **kwargs)*

Non-relativistic ROHF gradients

analyze *(verbose=None, with_meta_lowdin=True, **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 separately (without change occupancy).

check_sanity ()

Check input of class/object attributes, check whether a class method is overwritten. It does not check the attributes which are prefixed with "_". The return value of method set is the object itself. This allows a series of functions/methods to be executed in pipe.

eig *(fock, s)*

Solver for generalized eigenvalue problem

\[
HC = SC \]

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_roothena_fock()

get_grad *(mo_coeff, mo_occ, fock=None)*

ROHF gradients is the off-diagonal block \([co + cv + ov]\), where \([cc \ co \ cv]\) \([oo \ ov \ vv]\)

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:

```python
>>> 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: \( V_{hf} = (V^\alpha, V^\beta) \). \( V^\alpha \) (and \( V^\beta \)) can be a list matrices, corresponding to the input density matrices.

Examples:

```python
>>> 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, **kwargs)
One-particle density matrix. mo_occ is a 1D array, with occupancy 1 or 2.

spin_square (mo_coeff=None, s=None)
Spin square and multiplicity of RHF determinant

stability (internal=True, external=False, verbose=None)
ROHF/ROKS stability analysis.

See also pyscf.scf.stability.rohf_stability function.

Kwargs:
internal [bool] Internal stability, within the RHF optimization space.

external [bool] External stability. It is not available in current version.

Returns: The return value includes two set of orbitals which are more close to the required stable condition.

pyscf.scf.rohf.analyze(mf, verbose=5, with_meta_lowdin=True, **kwargs)
Analyze the given SCF object: print orbital energies, occupancies; print orbital coefficients; Mulliken population analysis

pyscf.scf.rohf.canonicalize(mf, mo_coeff, mo_occ, fock=None)
Canonicalization diagonalizes the Fock matrix within occupied, open, virtual subspaces separately (without change occupancy).

pyscf.scf.rohf.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()

pyscf.scf.rohf.get_grad(mo_coeff, mo_occ, fock)
ROHF gradients is the off-diagonal block \([co + cv + ov]\), where \([cc \ co \ cv]\ [oc \ oo \ ov]\ [vc \ vo \ vv]\)

pyscf.scf.rohf.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:

```python
>>> 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])
```

pyscf.scf.rohf.get_roothaan_fock(focka_fockb, dma_dmb, s)

<table>
<thead>
<tr>
<th>space</th>
<th>closed</th>
<th>open</th>
<th>virtual</th>
</tr>
</thead>
<tbody>
<tr>
<td>closed</td>
<td>Fc</td>
<td>Fb</td>
<td>Fc</td>
</tr>
<tr>
<td>open</td>
<td>Fb</td>
<td>Fc</td>
<td>Fa</td>
</tr>
<tr>
<td>virtual</td>
<td>Fc</td>
<td>Fa</td>
<td>Fc</td>
</tr>
</tbody>
</table>

where Fc = (Fa + Fb) / 2

Returns: Roothaan effective Fock matrix

pyscf.scf.rohf.make_rdm1(mo_coeff, mo_occ, **kwargs)
One-particle densit matrix. mo_occ is a 1D array, with occupancy 1 or 2.

Unrestricted coupled pertubed Hartree-Fock solver

pyscf.scf.ucphf.kernel(fvind, mo_energy, mo_occ, h1, s1=None, max_cycle=20, tol=1e-09, hermi=False, verbose=2)

Args:

fvind [function] Given density matrix, compute \((ij|kl)D_{lk}*2 - (ij|kl)D_{jk}\)

pyscf.scf.ucphf.solve(fvind, mo_energy, mo_occ, h1, s1=None, max_cycle=20, tol=1e-09, hermi=False, verbose=2)

Args:
fvind [function] Given density matrix, compute (ijkl)\*2 - (ijkl)D_{lk}

pyscf.scf.ucphf.solve_nos1(fvind, mo_energy, mo_occ, h1, max_cycle=20, tol=1e-09, hermi=False, verbose=2)

For field independent basis. First order overlap matrix is zero

pyscf.scf.ucphf.solve_withs1(fvind, mo_energy, mo_occ, h1, s1, max_cycle=20, tol=1e-09, hermi=False, verbose=2)

For field dependent basis. First order overlap matrix is non-zero. The first order orbitals are set to C^1_{ij} = -1/2 S1 e1 = h1 - s1*e0 + (e0_j-e0_i)*c1 + vhf[c1]

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, **kwargs)

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. Writing to chkfile can be disabled if this attribute is set to None or False.

conv_tol [float] converge threshold. Default is 1e-9

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’, ‘hcore’, ‘1e’, ‘chkfile’. Default is ‘minao’

DIIS [DIIS class] The class to generate diis object. It can be one of diis.SCF_DIIS, diis.ADIIS, diis.EDIIS.

diis [boolean or object of DIIS class defined in scf.diis.] Default is the object associated to the attribute self.DIIS. Set it to None/False to turn off DIIS. Note if this attribute is initialized as a DIIS object, the SCF driver will use this object in the iteration. The DIIS informations (vector basis and error vector) will be held inside this object. When kernel function is called again, the old states (vector basis and error vector) will be reused.

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.

204 Chapter 1. Contents
**direct_scf** [bool] Direct SCF is used by default.


**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 environment.

**conv_check** [bool] An extra cycle to check convergence after SCF iterations.

**check_convergence** [function(envs) => bool] A hook for overloading convergence criteria in 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:

```python
>>> 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()
-1.0811707843765884
```

Attributes for Dirac-Hartree-Fock

**with_ssss** [bool, for Dirac-Hartree-Fock only] If False, ignore small component integrals (SSISS). 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:

```python
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1', basis='cc-pvdz', 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
```

**Gradients** *(obj, *args, **kwargs)*

Unrestricted Dirac-Hartree-Fock gradients

**dip_moment** *(mol=None, dm=None, unit='Debye', verbose=3, **kwargs)*

---

1.11. scf — Mean-field methods
Dipole moment calculation

\[
\begin{align*}
\mu_x &= -\sum_{\mu} \sum_{\nu} P_{\mu\nu}(\nu|x|\mu) + \sum_A Q_A X_A \\
\mu_y &= -\sum_{\mu} \sum_{\nu} P_{\mu\nu}(\nu|y|\mu) + \sum_A Q_A Y_A \\
\mu_z &= -\sum_{\mu} \sum_{\nu} P_{\mu\nu}(\nu|z|\mu) + \sum_A Q_A Z_A 
\end{align*}
\]

where \(\mu_x, \mu_y, \mu_z\) 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

**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.

**mulliken_pop** *(mol=None, dm=None, s=None, verbose=5)*  
Mulliken population analysis

\[
M_{ij} = D_{ij} S_{ji}
\]

Mulliken charges

\[
\delta_i = \sum_j M_{ij}
\]

**reset** *(mol)*  
Reset mol and clean up relevant attributes for scanner mode

**pyscf.scf.dhf.dip_moment** *(mol, dm, unit='Debye', verbose=3, **kwargs)*  
Dipole moment calculation

\[
\begin{align*}
\mu_x &= -\sum_{\mu} \sum_{\nu} P_{\mu\nu}(\nu|x|\mu) + \sum_A Q_A X_A \\
\mu_y &= -\sum_{\mu} \sum_{\nu} P_{\mu\nu}(\nu|y|\mu) + \sum_A Q_A Y_A \\
\mu_z &= -\sum_{\mu} \sum_{\nu} P_{\mu\nu}(\nu|z|\mu) + \sum_A Q_A Z_A 
\end{align*}
\]

where \(\mu_x, \mu_y, \mu_z\) 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.dhf.get_grad** *(mo_coeff, mo_occ, fock_ao)*  
DHF Gradients

**pyscf.scf.dhf.get_init_guess** *(mol, key='minao')*  
Generate density matrix for initial guess

**Kwargs:**

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_chkfile(mol, chkfile_name, project=None)
Read SCF chkfile and make the density matrix for 4C-DHF initial guess.

**Kwargs:**

- **project** [None or bool] Whether to project chkfile’s orbitals to the new basis. Note when the geometry of the chkfile and the given molecule are very different, this projection can produce very poor initial guess. In PES scanning, it is recommended to switch off project.

  If project is set to None, the projection is only applied when the basis sets of the chkfile’s molecule are different to the basis sets of the given molecule (regardless whether the geometry of the two molecules are different). Note the basis sets are considered to be different if the two molecules are derived from the same molecule with different ordering of atoms.

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.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.dhf.time_reversal_matrix(mol, mat)
\[ T(A_{ij}) = A^{[T(i),T(j)]^*} \]

**addons**

pyscf.scf.addons.convert_to_ghf(mf, out=None, remove_df=False)
Convert the given mean-field object to the generalized HF/KS object

Note if mf is an second order SCF object, the second order object will not be converted (in other words, only the underlying SCF object will be converted)

**Args:** mf : SCF object

**Kwargs**

- **remove_df** [bool] Whether to convert the DF-SCF object to the normal SCF object. This conversion is not applied by default.

**Returns:** An generalized SCF object
pyscf.scf.addons.convert_to_rhf(mf, out=None, remove_df=False)
Convert the given mean-field object to the restricted HF/KS object

Note if mf is an second order SCF object, the second order object will not be converted (in other words, only the underlying SCF object will be converted)

Args: mf : SCF object

Kwarg:

remove_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

dynscf.scf.addons.convert_to_uhf(mf, out=None, remove_df=False)
Convert the given mean-field object to the unrestricted HF/KS object

Note if mf is an second order SCF object, the second order object will not be converted (in other words, only the underlying SCF object will be converted)

Args: mf : SCF object

Kwarg:

remove_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

dynscf.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)

dynscf.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)

dynscf.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

dynscf.scf.addons.fast_newton(mf, mo_coeff=None, mo_occ=None, dm0=None, auxbasis=None, dual_basis=None, **newton_kwargs)
This is a wrap function which combines several operations. This function first setup the initial guess from density fitting calculation then use for Newton solver and call Newton solver. Newton solver attributes [max_cycle_inner, max_stepsize, ah_start_tol, ah_conv_tol, ah_grad_trust_region, ...] can be passed through **newton_kwargs.

dynscf.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

dynscf.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

dynscf.scf.addons.get_ghf_orbspin(mo_energy, mo_occ, is_rhf=None)
Spin of each GHF orbital when the GHF orbitals are converted from RHF/UHF orbitals

For RHF orbitals, the orbspin corresponds to first occupied orbitals then unoccupied orbitals. In the occupied orbital space, if degenerated, first alpha then beta, last the (open-shell) singly occupied (alpha) orbitals. In the unoccupied orbital space, first the (open-shell) unoccupied (beta) orbitals if applicable, then alpha and beta orbitals

For UHF orbitals, the orbspin corresponds to first occupied orbitals then unoccupied orbitals.
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_dm_nr2nr(mol1, dm1, mol2)
Project density matrix representation from basis set 1 (mol1) to basis set 2 (mol2).

\[
\langle \text{AO}^2 | \text{DM} \text{AO}^2 | \text{AO}^2 \rangle = \langle \text{AO}^2 | P \text{DM} \text{AO}^1P | \text{AO}^2 \rangle
\]

There are three relevant functions: project_dm_nr2nr() is the projection for non-relativistic (scalar) basis, project_dm_nr2r() projects from non-relativistic to relativistic basis. project_dm_r2r() is the projection between relativistic (spinor) basis.

pyscf.scf.addons.project_mo_nr2nr(mol1, mo1, mol2)
Project orbital coefficients from basis set 1 (C1 for mol1) to basis set 2 (C2 for mol2).

\[
\langle \psi_1 | = \langle \text{AO}^1 | C_1
\]

\[
\langle \psi_2 | = P \langle \psi_1 | = \langle \text{AO}^2 | S^{-1} \langle \text{AO}^2 | \text{AO}^1 \rangle | C_1 = \langle \text{AO}^2 | C_2
\]

\[
\sum_{i=1}^{N} S_{\text{AO}^1}^{-1} \langle \text{AO}^2 | \text{AO}^1 \rangle C_1 = S_{\text{AO}^1}^{-1} \langle \text{AO}^2 | \text{AO}^1 \rangle C_1
\]

There are three relevant functions: project_mo_nr2nr() is the projection for non-relativistic (scalar) basis, project_mo_nr2r() projects from non-relativistic to relativistic basis. project_mo_r2r() is the projection between relativistic (spinor) basis.

pyscf.scf.addons.remove_linear_dep(mf, threshold=1e-08, lindep=1e-10)

Args:

threshold [float] The threshold under which the eigenvalues of the overlap matrix are discarded to avoid numerical instability.

lindep [float] The threshold that triggers the special treatment of the linear dependence issue.

pyscf.scf.addons.remove_linear_dep_(mf, threshold=1e-08, lindep=1e-10)

Args:

threshold [float] The threshold under which the eigenvalues of the overlap matrix are discarded to avoid numerical instability.

lindep [float] The threshold that triggers the special treatment of the linear dependence issue.

pyscf.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, incore=False)
Ref: JCP, 132, 054109

1.11. scf — Mean-field methods
class `pyscf.scf.diis.EDIIS`(dev=None, filename=None, incore=False)

SCF-EDIIS Ref: JCP 116, 8255

`pyscf.scf.diis.get_err_vec`(s, d, f)

error vector = SDF - FDS

jk

General JK contraction function for * arbitrary integrals * 4 different molecules * multiple density matrices * arbitrary basis subset for the 4 indices

`pyscf.scf.jk.get_jk`(mols, dms, scripts=["ijkl,ji->kl"], intor='int2e_sph', aosym='s1', comp=None, hermi=0, shls_slice=None, verbose=2, vhfopt=None)

Compute J/K matrices for the given density matrix

Args:
mols : an instance of Mole or a list of Mole objects
dms [ndarray or list of ndarrays] A density matrix or a list of density matrices

Kwargs:
hermi [int] Whether the returned J (K) matrix is hermitian

0: no hermitian or symmetric
1: hermitian
2: anti-hermitian

intor [str] 2-electron integral name. See `getints()` 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 (ijkl)
‘2kl’ or ‘s2kl’: symmetry between k, l in (ijkl)
1 or ‘1’ or ‘s1’: no symmetry
‘a4ij’: 4-fold symmetry with anti-symmetry between i, j in (ijkl)
‘a4kl’: 4-fold symmetry with anti-symmetry between k, l in (ijkl)
‘a2ij’: anti-symmetry between i, j in (ijkl)
‘a2kl’: anti-symmetry between k, l in (ijkl)

comp [int] Components of the integrals, e.g. cint2e_ip_sph has 3 components.

scripts [string or a list of strings] Contraction description (following numpy.einsum convention) based on letters [ijkl]. Each script will be one-to-one applied to each entry of dms. So it must have the same number of elements as the dms, len(scripts) == len(dms).

shls_slice [8-element list] (ish_start, ish_end, jsh_start, jsh_end, ksh_start, ksh_end, lsh_start, lsh_end)

Returns: Depending on the number of density matrices, the function returns one J/K matrix or a list of J/K matrices (the same number of entries as the input dms). Each JK matrices may be a 2D array or 3D array if the AO integral has multiple components.
Examples:

```python
>>> from pyscf import gto
>>> mol = gto.M(atom='H 0 -.5 0; H 0 .5 0', basis='cc-pvdz')
>>> nao = mol.nao_nr()
>>> dm = numpy.random.random((nao,nao))
>>> # Default, Coulomb matrix
>>> vj = get_jk(mol, dm)
>>> # Coulomb matrix with 8-fold permutation symmetry for AO integrals
>>> vj = get_jk(mol, dm, 'ijkl,ji->kl', aosym='s8')
>>> # Exchange matrix with 8-fold permutation symmetry for AO integrals
>>> vk = get_jk(mol, dm, 'ijkl,jk->il', aosym='s8')
>>> # Compute coulomb and exchange matrices together
>>> vj, vk = get_jk(mol, dm, ('ijkl,ji->kl','ijkl,li->kj'), aosym='s8')
>>> # Analytical gradients for coulomb matrix
>>> j1 = get_jk(mol, dm, 'ijkl,lk->ij', intor='int2e_ip1_sph', aosym='s2kl', comp=3)
>>> # contraction across two molecules
>>> moll = gto.M(atom='He 2 0 0', basis='6-31g')
>>> naol = moll.nao_nr()
>>> dml = numpy.random.random((naol,naol))
>>> # Coulomb interaction between two molecules, note 4-fold symmetry can be applied
>>> jcross = get_jk((moll,moll,moll,moll), dm, scripts='ijkl,lk->ij', aosym='s4')
>>> ecul = numpy.einsum('ij,ij', jcross, dml)
>>> # Exchange interaction between two molecules, no symmetry can be used
>>> kcross = get_jk((moll,moll,moll,moll), dm, scripts='ijkl,jk->il')
>>> ex = numpy.einsum('ij,ji', kcross, dml)
>>> # Analytical gradients for coulomb matrix between two molecules
>>> jcross1 = get_jk((moll,moll,moll,moll), dm, scripts='ijkl,lk->ij', intor='int2e_ip1_sph', comp=3)
>>> # Analytical gradients for coulomb interaction between 1s density and the other molecule
>>> jpart1 = get_jk((moll,moll,moll,moll), dm, scripts='ijkl,lk->ij', intor='int2e_ip1_sph', comp=3, shls_slice=(0,1,0,1,0,mol.nbas,0,mol.nbas))
```

```

pyscf.scf.jk.jk_build(mols, dms, scripts=['ijkl,ji->kl'], intor='int2e_sph', aosym='s1', comp=None, hermi=0, shls_slice=None, verbose=2, vhfopt=None)
```

Compute J/K matrices for the given density matrix

**Args:**
- `mols` : an instance of Mole or a list of Mole objects
- `dms` [ndarray or list of ndarrays] A density matrix or a list of density matrices

**Kwargs:**
- `hermi` [int] Whether the returned J (K) matrix is hermitian
  - 0 : no hermitian or symmetric
  - 1 : hermitian
  - 2 : anti-hermitian
- `intor` [str] 2-electron integral name. See `get ints()` 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 (ijkl)
- ‘2kl’ or ‘s2kl’: symmetry between k, l in (ijkl)
- 1 or ‘1’ or ‘s1’: no symmetry
- ‘a4ij’ : 4-fold symmetry with anti-symmetry between i, j in (ijkl)
- ‘a4kl’ : 4-fold symmetry with anti-symmetry between k, l in (ijkl)
- ‘a2ij’ : anti-symmetry between i, j in (ijkl)
- ‘a2kl’ : anti-symmetry between k, l in (ijkl)

**comp** [int] Components of the integrals, e.g. cint2e_ip_sph has 3 components.

**scripts** [string or a list of strings] Contraction description (following numpy.einsum convention) based on letters [ijkl]. Each script will be one-to-one applied to each entry of dms. So it must have the same number of elements as the dms, len(scripts) == len(dms).

**shls_slice** [8-element list] (ish_start, ish_end, jsh_start, jsh_end, ksh_start, ksh_end, lsh_start, lsh_end)

**Returns:** Depending on the number of density matrices, the function returns one J/K matrix or a list of J/K matrices (the same number of entries as the input dms). Each JK matrices may be a 2D array or 3D array if the AO integral has multiple components.

Examples:

```python
>>> from pyscf import gto
>>> mol = gto.M(atom='H 0 .5 0; H 0 .5 0', basis='cc-pvdz')
>>> nao = mol.nao_nr()
>>> dm = numpy.random.random((nao,nao))
>>> # Default, Coulomb matrix
>>> vj = get_jk(mol, dm)
>>> # Coulomb matrix with 8-fold permutation symmetry for AO integrals
>>> vj = get_jk(mol, dm, 'ijkl,ji->kl', aosym='s8')
>>> # Exchange matrix with 8-fold permutation symmetry for AO integrals
>>> vk = get_jk(mol, dm, 'ijkl,jk->il', aosym='s8')
>>> # Compute coulomb and exchange matrices together
>>> vj, vk = get_jk(mol, dm, 'ijkl,ji->kl', 'ijkl,li->kj', aosym='s8')
>>> # Analytical gradients for coulomb matrix
>>> j1 = get_jk(mol, dm, 'ijkl,ik->ij', intor='int2e_ip1_sph', aosym='s2kl', comp=3)
>>> # contraction across two molecules
>>> mol1 = gto.M(atom='He 2 0 0', basis='6-31g')
>>> naol = mol1.nao_nr()
>>> dm1 = numpy.random.random((naol,naol))
>>> # Coulomb interaction between two molecules, note 4-fold symmetry can be applied
>>> jcross = get_jk((mol,mol,mol,mol), dm, scripts='ijkl,ik->ij', aosym='s4')
>>> ecoul = numpy.einsum('ij,ij', jcross, dm1)
>>> # Exchange interaction between two molecules, no symmetry can be used
>>> kcross = get_jk((mol,mol,mol,mol), dm, scripts='ijkl,ik->il')
>>> ex = numpy.einsum('ij,ji', kcross, dm1)
```
newton_ah

stability

Wave Function Stability Analysis
Ref. JCP, 66, 3045 JCP, 104, 9047
See also tddft/rhf.py and scf/newton_ah.py

pyscf.scf.stability.rhf_stability(mf, internal=True, external=False, verbose=None)
Stability analysis for RHF/RKS method.

Args:
mf : RHF or RKS object

Kwargs:

internal [bool] Internal stability, within the RHF space.


Returns: New orbitals that are more close to the stable condition. The return value includes two set of orbitals. The first corresponds to the internal stability and the second corresponds to the external stability.

pyscf.scf.stability.rohf_stability(mf, internal=True, external=False, verbose=None)
Stability analysis for ROHF/ROKS method.

Args:
mf : ROHF or ROKS object

Kwargs:

internal [bool] Internal stability, within the RHF space.

external [bool] External stability. It is not available in current version.

Returns: The return value includes two set of orbitals which are more close to the required stable condition.

pyscf.scf.stability.uhf_stability(mf, internal=True, external=False, verbose=None)
Stability analysis for RHF/RKS method.

Args:
mf : UHF or UKS object

Kwargs:

internal [bool] Internal stability, within the UHF space.


Returns: New orbitals that are more close to the stable condition. The return value includes two set of orbitals. The first corresponds to the internal stability and the second corresponds to the external stability.
stability_slow

Wave Function Stability Analysis
Ref. JCP, 66, 3045 JCP, 104, 9047

1.12 dft — Density functional theory

The dft module implements Kohn-Sham density functional theory. Also see scf. A minimal example:

```python
from pyscf import gto, dft
mol = gto.M(atom='H 0 0 0; F 0.9 0 0', basis='sto3g')
mf = dft.RKS(mol)
mf.xc = 'lda,vwn'
mf.kernel()
```

1.12.1 Examples

Relevant Examples: examples/pbc/00-simple_dft.py examples/pbc/02-gks.py examples/pbc/11-grid_scheme.py examples/pbc/12-camb3lyp.py examples/pbc/13-rsh_dft.py examples/pbc/21-x2c.py examples/pbc/23-decorate_scf.py examples/pbc/24-custom_xc_functional.py examples/pbc/24-define_xc_functional.py examples/pbc/32-broken_symmetry_dft.py examples/pbc/32-xcfun_as_default.py examples/pbc/33-nlc_functionals.py

1.12.2 Customizing XC functional

XC functional of DFT methods can be customized. The simplest way to customize XC functional is to assign a string expression to mf.xc:

```python
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 rules, as described below.

- 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, e.g. `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:

```python
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)
    exc, fxc, kxc = return exc, vxc, fxc, kxc
def 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.

1.12.3 Program reference

Non-relativistic restricted Kohn-Sham
class pyscf.dft.rks.KohnShamDFT(mf)

Attributes for Kohn-Sham DFT:

- `xc` [str] ‘X_name,C_name’ for the XC functional. Default is ‘lda,vwn’
- `nlc` [str] ‘NLC_name’ for the NLC functional. Default is ‘’ (i.e., None)
- `omega` [float] Omega of the range-separated Coulomb operator e^(-omega r_{12}^2) / r_{12}
- `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.COV_ALENT_RADII

1.12. dft — Density functional theory
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
```

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. Writing to chkfile can be disabled if this attribute is set to None or False.
- **conv_tol** [float] converge threshold. Default is 1e-9
- **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’, ‘hcore’, ‘1e’, ‘chkfile’. Default is ‘minao’
- **DIIS** [DIIS class] The class to generate diis object. It can be one of diis.SCF_DIIS, diis.ADIIS, diis.EDIIS.
diis  [boolean or object of DIIS class defined in scf.diis.] Default is the object associated to the attribute self.DIIS. Set it to None/False to turn off DIIS. Note if this attribute is initialized as a DIIS object, the SCF driver will use this object in the iteration. The DIIS informations (vector basis and error vector) will be held inside this object. When kernel function is called again, the old states (vector basis and error vector) will be reused.

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.


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 environment.

conv_check  [bool] An extra cycle to check convergence after SCF iterations.

check_convergence  [function(envs) => bool] A hook for overloading convergence criteria in 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:

```python
>>> 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 Kohn-Sham DFT:

xc  [str] ‘X_name,C_name’ for the XC functional. Default is ‘lda,vwn’

nlc  [str] ‘NLC_name’ for the NLC functional. Default is ‘’ (i.e., None)

omega  [float] Omega of the range-separated Coulomb operator $e^{-\omega r_{12}^2} / r_{12}$

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.sgl_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=None, h1e=None, vhf=None)
Electronic part of RKS energy.

Note this function has side effects which cause mf.scf_summary updated.

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 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.

```python
pyscf.dft.rks.energy_elec(ks, dm=None, h1e=None, vhf=None)
```

Electronic part of RKS energy.

Note this function has side effects which cause mf.scf_summary updated.

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 contribution

```python
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.

---

1.12. dft — Density functional theory 219
2: anti-hermitian

Returns: matrix $V_{eff} = J + V_{xc}$. $V_{eff}$ can be a list matrices, if the input $dm$ is a list of density matrices.

Non-relativistic Unrestricted Kohn-Sham

```python
class pyscf.dft.uks.UKS(mol)
    Unrestricted Kohn-Sham See pyscf/dft.py RKS class for document 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() for more details.
```

```python
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() for more details.
```

Generalized Kohn-Sham

```python
class pyscf.dft.gks.GKS(mol)
    Generalized Kohn-Sham
    energy_elec(ks, dm=None, h1e=None, vhf=None)
        Electronic part of RKS energy.
        Note this function has side effects which cause mf.scf_summary updated.
        Args: ks : an instance of DFT class
            dm [2D ndarray] one-particle density matrix
            h1e [2D ndarray] Core hamiltonian
        Returns: RKS electronic energy and the 2-electron contribution
        get_veff(ks, mol=None, dm=None, dm_last=0, vhf_last=0, hermi=1)
            Coulomb + XC functional for GKS.
```

```python
pyscf.dft.gks.get_veff(ks, mol=None, dm=None, dm_last=0, vhf_last=0, hermi=1)
    Coulomb + XC functional for GKS.
```

Generalized Kohn-Sham

```python
class pyscf.dft.gks_symm.GKS(mol)
    Restricted Kohn-Sham
    energy_elec(ks, dm=None, h1e=None, vhf=None)
        Electronic part of RKS energy.
        Note this function has side effects which cause mf.scf_summary updated.
        Args: ks : an instance of DFT class
            dm [2D ndarray] one-particle density matrix
            h1e [2D ndarray] Core hamiltonian
        Returns: RKS electronic energy and the 2-electron contribution
        get_veff(ks, mol=None, dm=None, dm_last=0, vhf_last=0, hermi=1)
            Coulomb + XC functional for GKS.
```

Non-relativistic Restricted Kohn-Sham

```python
pyscf.dft.rks_symm.RKS
    alias of SymAdaptedRKS
```
pyscf.dft.rks_symm.ROKS
alias of SymAdaptedROKS

class pyscf.dft.rks_symm.SymAdaptedRKS(mol)
  Restricted Kohn-Sham

  energy_elec(ks, dm=None, h1e=None, vhf=None)
  Electronic part of RKS energy.
  Note this function has side effects which cause mf.scf_summary updated.
  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 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
  Kwargrs:
    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.

class pyscf.dft.rks_symm.SymAdaptedROKS(mol)
  Restricted Kohn-Sham

  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() for more details.

Non-relativistic Unrestricted Kohn-Sham

class pyscf.dft.uks_symm.SymAdaptedUKS(mol)
  Restricted Kohn-Sham

  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() for more details.
pyscf.dft.uks_symm.UKS
alias of SymAdaptedUKS

XC functional, the interface to libxc (http://www.tddft.org/programs/octopus/wiki/index.php/Libxc)
pyscf.dft.libxc.define_xc ni, description, xctype='LDA', hyb=0, rsh=(0, 0, 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.

Kwargs:

xctype [str] ‘LDA’ or ‘GGA’ or ‘MGGA’
hyb [float] hybrid functional coefficient
rsh [a list of three floats] coefficients (omega, alpha, beta) for range-separated hybrid functional. omega is the exponent factor in attenuated Coulomb operator $e^{-\omega r_{12}}/r_{12}$ alpha is the coefficient for long-range part, hybrid coefficient can be obtained by alpha + beta

Examples:

```python
>>> 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, rsh=(0, 0, 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.

Kwargs:

xctype [str] ‘LDA’ or ‘GGA’ or ‘MGGA’
hyb [float] hybrid functional coefficient
rsh [a list of three floats] coefficients (omega, alpha, beta) for range-separated hybrid functional. omega is the exponent factor in attenuated Coulomb operator $e^{-\omega r_{12}}/r_{12}$ alpha is the coefficient for long-range part, hybrid coefficient can be obtained by alpha + beta
Examples:

```python
>>> 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, omega=None, 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” (exact exchange) is appeared in the string, the HF part will be skipped. If an empty string “""" is given, the returns exc, vxc,... will be vectors of zeros.
- **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, v2tau2, v2rhola, v2rhota, v2sigma2, v2sigmatau)
- 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_u, u_u_ud, u_d_dd, d_d_u, d_d_u, d_d_dd) | v3sigma3[:,10] = (uu_uu_u, uu_uu_ud, uu_ud_dd, uu_ud_ud, uu_dd_dd, uu_dd_ud, uu_dd_dd, uu_dd_dd, uu_dd_dd, uu_dd_dd)

See also libxc itrf.c

```python
pyscf.dft.libxc.hybrid_coeff(xc_code, spin=0)
```
Support recursively defining hybrid functional

```python
pyscf.dft.libxc.nlc_coeff(xc_code)
```
Get NLC coefficients

```python
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 in string, the entire string is considered as a compound XC functional (including both X and C functionals, such as b3lyp).
  - To input only X functional (without C functional), leave the second part blank. E.g. description='slater', means pure LDA functional.
  - To neglect X functional (just apply C functional), leave the first part blank. E.g. description=',vwn' means pure VWN functional.
  - If compound XC functional is specified, no matter whether it is in the X part (the string in front of comma) or the C part (the string behind comma), both X and C functionals of the compound XC functional will be used.
- 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 can have at most one factor. If the factor is not given, it is set to 1. Compound functional can be scaled as a unit. For example ‘0.5*b3lyp’ is equivalent to ‘HF*0.1 + .04*LDA + .36*B88, .405*LYP + .095*VWN’

String “HF” stands for exact exchange (HF K matrix). Putting “HF” in correlation functional part is the same to putting “HF” in exchange part.

String “RSH” means range-separated operator. Its format is RSH(omega, alpha, beta). Another way to input RSH is to use keywords SR_HF and LR_HF: “SR_HF(0.1) * alpha_plus_beta” and “LR_HF(0.1) * alpha” where the number in parenthesis is the value of omega.

Be careful with the libxc convention on GGA functional, in which the LDA contribution has been 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, v2tau2, v2rhoatl, v2sigmalapl, v2sigmatau)
• kxc for restricted case: v3rho3, v3rho2sigma, v3rhosigma2, v3sigma3, v3rho2tau, v3rhosigmatau, v3hotau2, v3sigmalapl, v3sigmatau2, v3tau3
• kxc for unrestricted case: | v3rho3[:,4] = (u_u_u, u_u_d, u_d_d, d_d_d) | v3rho2sigma[:,9] = (u_u_u, u_u_d, u_d_dd, d_d_u, d_d_d, d_dd_u, d_dd_d) | v3rhosigma2[:,12] = (uu_uu, uu_ud, uu_dd, ud_ud, ud_dd, ud_dd, d_uu, d_u_d, d_d_dd, d_dd_dd) | v3sigma3[:,10] = (uu_uu, uu_ud, uu_dd, uu_dd, uu_ud, uu_dd, dd_dd, dd_dd, dd_dd, dd_dd) | v3rho2tau | v3rhosigmatau | v3hotau2 | v3sigmalapl | v3sigmatau2 | v3tau3

see also libxc_itrf.c

1.12. dft — Density functional theory 225
Convert the XC functional name to libxc library internal ID.

Range-separated parameter and HF exchange components: omega, alpha, beta

\[
\text{Exc}_{RSH} = c_{LR} \cdot \text{LR}_{HFX} + c_{SR} \cdot \text{SR}_{HFX} + (1-c_{SR}) \cdot \text{Ex}_{SR} + (1-c_{LR}) \cdot \text{Ex}_{LR} + \text{Ec} = \alpha
\]

\[
\text{SR}_{HFX} = \langle \pi | e^{-\omega r_{12}}/r_{12} | \eta \rangle \quad \text{LR}_{HFX} = \langle \pi | (1-e^{-\omega r_{12}})/r_{12} | \eta \rangle
\]

\[
\alpha = c_{LR} - \beta = \text{hyb} - \alpha
\]

4-component Dirac-Kohn-Sham

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.
2. Delley radial grids. Ref. JCP, 104, 9848 log2 algorithm

\[
\text{pyscf.dft.radi.mura_knowles}(n, \text{charge=\texttt{None}}, *\text{args}, **\text{kwargs})
\]
Mura-Knowles (JCP, 104, 9848) log3 quadrature radial grids

\[
\text{pyscf.dft.radi.treutler}(n, *\text{args}, **\text{kwargs})
\]
Treutler-Ahlrichs (JCP 102, 346 (M4)) radial grids

\[
\text{pyscf.dft.radi.treutler_ahlrichs}(n, *\text{args}, **\text{kwargs})
\]
Treutler-Ahlrichs (JCP 102, 346 (M4)) radial grids

\[
\text{pyscf.dft.radi.treutler_atomic_radii_adjust}(\text{mol}, \text{atomic_radii})
\]
Treutler atomic radii adjust function: JCP, 102, 346

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


class \texttt{pyscf.dft.gen_grid.Grids}\texttt{(mol)}
DFT mesh grids

**Attributes for Grids:**

- \texttt{level} [int (0 - 9)] big number for large mesh grids, default is 3
- \texttt{atomic_radii} [1D array]
  - \texttt{radi.BRAGG_RADIIS} (default)
  - \texttt{radi.COAVALENT_RADIIS}
  - \texttt{None} : to switch off atomic radii adjustment
- \texttt{radii_adjust} [function(mol, atomic_radii) \Rightarrow (function(atom_id, atom_id, g) \Rightarrow 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
- \texttt{radi_method} [function(n) \Rightarrow (rad_grids, rad_weights)] scheme for radial grids, can be one of | radi.treutler (default) | radi.delley | radi.mura_knowles | radi.gauss_chebyshev
- \texttt{becke_scheme} [function(v) \Rightarrow array_like_v] weight partition function, can be one of | gen_grid.original_becke (default) | gen_grid.stratmann
- \texttt{prune} [function(nuc, rad_grids, n_ang) \Rightarrow 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
- \texttt{symmetry} [bool] whether to symmetrize mesh grids (TODO)
- \texttt{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.
- \texttt{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:

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

def \texttt{gen_atomic_grids}(mol, \texttt{atom_grid=\texttt{None}}, \texttt{radi_method=\texttt{None}}, \texttt{level=\texttt{None}}, \texttt{prune=\texttt{None}}, **\texttt{kwargs})
Generate number of radial grids and angular grids for the given molecule.
**Returns:** A dict, with the atom symbol for the dict key. For each atom type, the dict value has two items: one is the meshgrid coordinates wrt the atom center; the second is the volume of that grid.

```python
def gen_partition(mol, atom_grids_tab, radii_adjust=None, atomic_radii=None):
    # Implementation details...
```

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.

```python
def make_mask(mol=None, coords=None, relativity=0, sshl_slice=None, verbose=None):
    # Implementation details...
```

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.
- `sshl_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.

```python
def reset(mol=None):
    # Implementation details...
```

Reset mol and clean up relevant attributes for scanner mode

```python
def pyscf.dft.gen_grid.gen_atomic_grids(mol, atom_grids={}, radi_method=<function
                                       gaussian_chebyshev>, level=3, prune=<function
                                       mwchem_prune>, **kwargs):
    # Implementation details...
```

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=None, coods=None, verbose=None)

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**

```python
(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, 0.9486306 , 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, 2.64561657,
 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, 3.30702072, 3.30702072, 3.30702072,
 3.30702072, 3.30702072, 3.30702072, 3.30702072,
 3.30702072, 3.30702072, 3.30702072, 3.30702072,
 3.30702072, 3.30702072, 3.30702072, 3.30702072,
 3.30702072, 3.30702072, 3.30702072, 3.30702072,
 3.30702072, 3.30702072, 3.30702072, 3.30702072,
 3.30702072, 3.30702072, 3.30702072, 3.30702072,
 3.30702072, 3.30702072, 3.30702072, 3.30702072,
 3.30702072, 3.30702072, 3.30702072, 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.sgl_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.

Kwarg:

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)

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.

Kwarg:

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:

```python
>>> 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),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 != 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 returned matrix is the Vxc matrix of alpha-spin. It is computed 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:

```python
>>> 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')
```

dft

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

Kwags:

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.get_rho (ni, mol, dm, grids, max_memory=2000)

Density in real space

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.


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:

```python
>>> 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)
>>> nao = mol.nao_nr()
>>> dm = numpy.random.random((nao,nao))
>>> ni = dft.numint.NumInt()
>>> nelec, exc, vxc = ni.nr_rks(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.


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:

```python
def nr_rks_fxc_st(ni, mol, grids, xcode, 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
```

```python
def nr_rks_vxc(ni, mol, grids, xcode, 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.
    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.
```

```python
>>> 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)
>>> nao = mol.nao_nr()
>>> dm = numpy.random.random((nao, nao))
>>> ni = dft.numint.NumInt()
>>> nelec, exc, vxc = ni.nr_rks(mol, grids, 'lda,vwn', dm)
```

```python
def nr_uks(ni, mol, grids, xcode, 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.
    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.

Examples:

```python
>>> 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)
>>> nao = mol.nao_nr()
>>> dm = numpy.random.random((2,nao,nao))
>>> ni = dft.numint.NumInt()
>>> nelec, exc, vxc = ni.nr_uks(mol, grids, 'lda,vwn', dm)
```

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.
- `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:

```python
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.
- `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.

Examples:

```python
>>> 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)
>>> nao = mol.nao_nr()
>>> dm = numpy.random.random((2,nao,nao))
>>> ni = dft.numint.NumInt()
>>> nelec, exc, vxc = ni.nr_uks(mol, grids, 'lda,vwn', dm)
```

pyscf.dft.numint.nr_vxc(mol, grids, xc_code, dms, spin=0, relativity=0, hermi=0, max_memory=2000, verbose=None)

Evaluate RKS/UKS XC functional and potential matrix on given meshgrids for a set of density matrices. See nr_rks() and nr_uks() for more details.

Args:
mol : an instance of Mole
grids [an instance of Grids] grids.coords and grids.weights are needed for coordinates and weights of meshgrids.
dms [2D array or 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:

```python
>>> 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)
>>> nao = mol.nao_nr()
>>> dm = numpy.random.random((2,nao,nao))
>>> nelec, exc, vxc = dft.numint.nr_vxc(mol, grids, 'lda,vwn', dm, spin=1)
```


pyscf.dft.xcfun.define_xc(ni, description, xctype='LDA', hyb=0, rsh=(0, 0, 0))

Define XC functional. See also eval_xc() for the rules of input description.

Args:
i : 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.
Kwargs:

    **xctype** [str] 'LDA' or 'GGA' or 'MGGA'

    **hyb** [float] hybrid functional coefficient

    **rsh** [a list of three floats] coefficients (omega, alpha, beta) for range-separated hybrid functional. omega is the exponent factor in attenuated Coulomb operator $e^{\omega r_{12}}/r_{12}$. alpha is the coefficient for long-range part, hybrid coefficient can be obtained by alpha + beta

Examples:

```python
>>> 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
...     xc = (vrho, None, None, None)
...     return exc, xc, fxc, kxc
>>> define_xc_(mf._numint, eval_xc, xctype='LDA')
>>> mf.kernel()
48.8525211046668
```

```
pyscf.dft.xcfun.define_xc_(ni, description, xctype='LDA', hyb=0, rsh=(0, 0, 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.

Kwargs:

    **xctype** [str] 'LDA' or 'GGA' or 'MGGA'

    **hyb** [float] hybrid functional coefficient

    **rsh** [a list of three floats] coefficients (omega, alpha, beta) for range-separated hybrid functional. omega is the exponent factor in attenuated Coulomb operator $e^{\omega r_{12}}/r_{12}$. alpha is the coefficient for long-range part, hybrid coefficient can be obtained by alpha + beta

Examples:

```python
>>> 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
...     xc = (vrho, None, None, None)
...     return exc, xc, fxc, kxc
>>> define_xc_(mf._numint, eval_xc, xctype='LDA')
>>> mf.kernel()
48.8525211046668
```
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

pycsf.dft.xcfun.eval_XC(xc_code, rho, spin=0, relativity=0, deriv=1, omega=None, verbose=None)

Interface to call xcfun library to evaluate XC functional, potential and functional derivatives.

See also pycsf.dft.libxc.eval_xc()

pycsf.dft.xcfun.nlc_coeff(xc_code)

Get NLC coefficients

pycsf.dft.xcfun.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 in string, the entire string is considered as a compound XC functional (including both X and C functionals, such as b3lyp).

  - To input only X functional (without C functional), leave the second part blank. E.g. description='slater,' means pure LDA functional.

  - To neglect X functional (just apply C functional), leave the first part blank. E.g. description=',vwn' means pure VWN functional.

  - If compound XC functional is specified, no matter whehter it is in the X part (the string in front of comma) or the C part (the string behind comma), both X and C functionals of the compound XC functional will be used.

• 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 can have at most one factor. If the factor is not given, it is set to 1. Compound functional can be scaled as a unit. For example ‘0.5*b3lyp’ is equivalent to ‘HF*0.1 + .04*LDA + .36*B88, .405*LYP + .095*VWN’

• String “HF” stands for exact exchange (HF K matrix). Putting “HF” in correlation functional part is the same to putting “HF” in exchange part.

• String “RSH” means range-separated operator. Its format is RSH(omega, alpha, beta). Another way to input RSH is to use keywords SR_HF and LR_HF: “SR_HF(0.1) * alpha_plus_beta” and “LR_HF(0.1) * alpha” where the number in parenthesis is the value of omega.

• Be careful with the convention on GGA functional, in which the LDA contribution has been included.

pycsf.dft.xcfun.rsh_coeff(xc_code)

Get Range-separated-hybrid coefficients

1.12. dft — Density functional theory 239
1.13 dftd3 — DFT plus Dispersion Correction

An interface to libdftd3: https://github.com/cuanto/libdftd3 A minimal example for this:

```python
from pyscf import gto, scf, dftd3
mol = gto.Mole()
    mol.atom = ''' O 0.00000000 0.00000000 -0.11081188
                  H -0.00000000 -0.84695236 0.59109389
                  H -0.00000000 0.89830571 0.52404783 '''
mol.basis = 'sto3g'
mol.build()
mf = dftd3.dftd3(scf.RHF(mol))
mf.kernel()
```

1.13.1 Examples

Relevant examples examples/dftd3/00-hf_with_dftd3.py

1.13.2 Program reference

**itrf**

DFT-D3 interface.

This interface is based on the open source project https://github.com/cuanto/libdftd3

```
pyscf.dftd3.itrf.dftd3(scf_method)
    Apply DFT-D3 corrections to SCF or MCSCF methods
    Args: scf_method : a HF or DFT object
    Returns: Same method object as the input scf_method with DFT-D3 energy corrections
    Examples:

    >>> mol = gto.M(atom='H 0 0 0; F 0 0 1', basis='ccpvdz', verbose=0)
    >>> mf = dftd3.dftd3(scf.RHF(mol))
    >>> mf.kernel()
    -101.940495711284
```

```
pyscf.dftd3.itrf.grad(scf_grad)
    Apply DFT-D3 corrections to SCF or MCSCF nuclear gradients methods
    Args:
        scf_grad [a HF or DFT gradient object (grad.HF or grad.RKS etc)] Once this function is applied on the SCF object, it affects all post-HF calculations eg MP2, CCSD, MCSCF etc
    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, -0.29235976, -0.38245077]]

1.14 grad — Analytical nuclear gradients

The `grad` module provides gradients for mean-field and correlated methods. For example, the RHF gradient can be computed by:

```python
from pyscf import gto, scf
mol = gto.M(
    atom = [['O', 0., 0., 0.],
            ['H', 0., -0.757, 0.587],
            ['H', 0., 0.757, 0.587]],
    basis = '631g')
mf = scf.RHF(mol)
mf.kernel()
g = mf.nuc_grad_method()
g.kernel()
```

1.14.1 Examples

Relevant examples

- examples/grad/01-scf_grad.py
- examples/grad/02-dft_grad.py
- examples/grad/03-mp2_grad.py
- examples/grad/04-cisd_grad.py
- examples/grad/05-ccsd_grad.py
- examples/grad/10-excited_state_cisd_grad.py
- examples/grad/11-excited_state_casci_grad.py
- examples/grad/16-scan_force.py

1.14.2 Program reference

Analytical nuclear gradients

Simple usage:

```python
>>> 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()
```

casci

CASCI analytical nuclear gradients


pyscf.grad.casci.Grad

alias of `Gradients`
class pyscf.grad.casci.Gradients(mc)
    Non-relativistic restricted Hartree-Fock gradients

    as_scanner (mcscf_grad, state=None)
    Generating a nuclear gradients scanner/solver (for geometry optimizer).
        The returned solver is a function. This function requires one argument “mol” as input and returns energy and
        first order nuclear derivatives.
        The solver will automatically use the results of last calculation as the initial guess of the new calculation.
        All parameters assigned in the nuc-grad object and 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, mcscf
        >>> mol = gto.M(atom='N 0 0 0; N 0 0 1.1', verbose=0)
        >>> mc_grad_scanner = mcscf.CASCI(scf.RHF(mol), 4, 4).nuc_grad_method().as_scanner()
        >>> etot, grad = mc_grad_scanner(gto.M(atom='N 0 0 0; N 0 0 1.1'))
        >>> etot, grad = mc_grad_scanner(gto.M(atom='N 0 0 0; N 0 0 1.5'))

pyscf.grad.casci.as_scanner (mcscf_grad, state=None)
    Generating a nuclear gradients scanner/solver (for geometry optimizer).
        The returned solver is a function. This function requires one argument “mol” as input and returns energy and
        first order nuclear derivatives.
        The solver will automatically use the results of last calculation as the initial guess of the new calculation. All
        parameters assigned in the nuc-grad object and 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, mcscf
        >>> mol = gto.M(atom='N 0 0 0; N 0 0 1.1', verbose=0)
        >>> mc_grad_scanner = mcscf.CASCI(scf.RHF(mol), 4, 4).nuc_grad_method().as_scanner()
        >>> etot, grad = mc_grad_scanner(gto.M(atom='N 0 0 0; N 0 0 1.1'))
        >>> etot, grad = mc_grad_scanner(gto.M(atom='N 0 0 0; N 0 0 1.5'))

casscf

CASSCF analytical nuclear gradients

pyscf.grad.casscf.Grad
    alias of Gradients

class pyscf.grad.casscf.Gradients(mc)
    Non-relativistic restricted Hartree-Fock gradients

    as_scanner (mcscf_grad)
    Generating a nuclear gradients scanner/solver (for geometry optimizer).
The returned solver is a function. This function requires one argument “mol” as input and returns energy and first order nuclear derivatives.

The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters assigned in the nuc-grad object and 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:

```python
>>> from pyscf import gto, scf, mcscf

>>> mol = gto.M(atom='N 0 0 0; N 0 0 1.1', verbose=0)

>>> mc_grad_scanner = mcscf.CASSCF(scf.RHF(mol), 4, 4).nuc_grad_method().as_scanner()

>>> etot, grad = mc_grad_scanner(gto.M(atom='N 0 0 0; N 0 0 1.1'))

>>> etot, grad = mc_grad_scanner(gto.M(atom='N 0 0 0; N 0 0 1.5'))
```

**pyscf.grad.casscf.as_scanner** *(mcscf_grad)*

Generating a nuclear gradients scanner/solver (for geometry optimizer).

The returned solver is a function. This function requires one argument “mol” as input and returns energy and first order nuclear derivatives.

The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters assigned in the nuc-grad object and 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:

```python
>>> from pyscf import gto, scf, mcscf

>>> mol = gto.M(atom='N 0 0 0; N 0 0 1.1', verbose=0)

>>> mc_grad_scanner = mcscf.CASSCF(scf.RHF(mol), 4, 4).nuc_grad_method().as_scanner()

>>> etot, grad = mc_grad_scanner(gto.M(atom='N 0 0 0; N 0 0 1.1'))

>>> etot, grad = mc_grad_scanner(gto.M(atom='N 0 0 0; N 0 0 1.5'))
```

**ccsd**

CCSD analytical nuclear gradients

**pyscf.grad.ccsd.as_scanner** *(grad_cc)*

Generating a nuclear gradients scanner/solver (for geometry optimizer).

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)).nuc_grad_method().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'))

ccsd_slow

RCCSD
Ref: JCP, 90, 1752

ccsd_t
cisd

CISD analytical nuclear gradients

pyscf.grad.cisd.as_scanner(grad_ci, state=0)
Generating a nuclear gradients scanner/solver (for geometry optimizer).

The returned solver is a function. This function requires one argument “mol” as input and returns total CISD energy.

The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters assigned in the CISD 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, ci
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1')
>>> ci_scanner = ci.CISD(scf.RHF(mol)).nuc_grad_method().as_scanner()
>>> e_tot, grad = ci_scanner(gto.M(atom='H 0 0 0; F 0 0 1.1'))
>>> e_tot, grad = ci_scanner(gto.M(atom='H 0 0 0; F 0 0 1.5'))

dhf

Relativistic Dirac-Hartree-Fock

pyscf.grad.dhf.Grad alias of Gradients
class pyscf.grad.dhf.Gradients(scf_method)
Unrestricted Dirac-Hartree-Fock gradients

as_scanner(mf_grad)
Generating a nuclear gradients scanner/solver (for geometry optimizer).

The returned solver is a function. This function requires one argument “mol” as input and returns energy and first order nuclear derivatives.
The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters assigned in the nuc-grad object and 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:

```python
>>> from pyscf import gto, scf, grad
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1')
>>> hf_scanner = scf.RHF(mol).apply(grad.RHF).as_scanner()
>>> e_tot, grad = hf_scanner(gto.M(atom='H 0 0 0; F 0 0 1.1'))
>>> e_tot, grad = hf_scanner(gto.M(atom='H 0 0 0; F 0 0 1.5'))
```

**extra_force**(atom_id, envs)

Hook for extra contributions in analytical gradients.

Contributions like the response of auxiliary basis in density fitting method, the grid response in DFT numerical integration can be put in this function.

class pyscf.grad.dhf.GradientsBasics(method)

Basic nuclear gradient functions for 4C relativistic methods

pyscf.grad.dhf.get_coulomb_hf(mol, dm, level='SSSS')

Dirac-Hartree-Fock Coulomb repulsion

pyscf.grad.dhf.get_veff(mol, dm, level='SSSS')

Dirac-Hartree-Fock Coulomb repulsion

mp2

MP2 analytical nuclear gradients

pyscf.grad.mp2.as_scanner(grad_mp)

Generating a nuclear gradients scanner/solver (for geometry optimizer).

The returned solver is a function. This function requires one argument “mol” as input and returns total MP2 energy.

The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters assigned in the MP2 and the underlying SCF objects (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:

```python
>>> from pyscf import gto, scf, mp
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1')
>>> mp2_scanner = mp.MP2(scf.RHF(mol)).nuc_grad_method().as_scanner()
>>> e_tot, grad = mp2_scanner(gto.M(atom='H 0 0 0; F 0 0 1.1'))
>>> e_tot, grad = mp2_scanner(gto.M(atom='H 0 0 0; F 0 0 1.5'))
```

rhf

Non-relativistic Hartree-Fock analytical nuclear gradients
pyscf.grad.rhf.

```
alias of Gradients
```  

```
class pyscf.grad.rhf.Gradients (method)
    Non-relativistic restricted Hartree-Fock gradients

    as_scanner (mf_grad)
        Generating a nuclear gradients scanner/solver (for geometry optimizer).
        The returned solver is a function. This function requires one argument “mol” as input and returns energy and first order nuclear derivatives.
        The solver will automatically use the results of last calculation as the initial guess of the new calculation.
        All parameters assigned in the nuc-grad object and 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, grad
        >>> mol = gto.M(atom='H 0 0 0; F 0 0 1')
        >>> hf_scanner = scf.RHF(mol).apply(grad.RHF).as_scanner()
        >>> e_tot, grad = hf_scanner(gto.M(atom='H 0 0 0; F 0 0 1.1'))
        >>> e_tot, grad = hf_scanner(gto.M(atom='H 0 0 0; F 0 0 1.5'))
```

```
extra_force (atom_id, envs)
    Hook for extra contributions in analytical gradients.
    Contributions like the response of auxiliary basis in density fitting method, the grid response in DFT numerical integration can be put in this function.
```

```
class pyscf.grad.rhf.GradientsBasics (method)
    Basic nuclear gradient functions for non-relativistic methods

    as_scanner()
        Generate Gradients Scanner

    get_jk (mol=None, dm=None, hermi=0)
        J = ((-nabla i) j| kl) D_lk K = ((-nabla i) j| kl) D_jk

    grad (*args, **kwargs)

    optimizer (solver='geometric')
        Geometry optimization solver
        Kwargs: solver (string) : geometry optimization solver, can be “geomeTRIC” (default) or “berny”.

    symmetrize (de, atmlst=None)
        Symmetrize the gradients wrt the point group symmetry of the molecule.
```

```
pyscf.grad.rhf.as_scanner (mf_grad)
    Generating a nuclear gradients scanner/solver (for geometry optimizer).
    The returned solver is a function. This function requires one argument “mol” as input and returns energy and first order nuclear derivatives.
    The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters assigned in the nuc-grad object and 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:

```python
>>> from pyscf import gto, scf, grad
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1')
>>> hf_scanner = scf.RHF(mol).apply(grad.RHF).as_scanner()
>>> e_tot, grad = hf_scanner(gto.M(atom='H 0 0 0; F 0 0 1.1'))
>>> e_tot, grad = hf_scanner(gto.M(atom='H 0 0 0; F 0 0 1.5'))
```

```python
pyscf.grad.rhf.get_hcore(mol)
Part of the nuclear gradients of core Hamiltonian
```  
```python
pyscf.grad.rhf.get_jk(mol, dm)
J = ((-nabla i) j| kl) D_lk K = ((-nabla i) jl kl) D_jk
```  
```python
pyscf.grad.rhf.get_veff(mf_grad, mol, dm)
NR Hartree-Fock Coulomb repulsion
```  
```python
pyscf.grad.rhf.make_rdm1e(mo_energy, mo_coeff, mo_occ)
Energy weighted density matrix
```  
```python
pyscf.grad.rhf.symmetrize(mol, de, atmlst=None)
Symmetrize the gradients wrt the point group symmetry of the molecule.
```

rks

Non-relativistic RKS analytical nuclear gradients

```python
pyscf.grad.rks.get_veff(ks_grad, mol=None, dm=None)
Coulomb + XC functional
```  
```python
pyscf.grad.rks.get_vxc_full_response(ni, mol, grids, xc_code, dms, relativity=0, hermi=1, max_memory=2000, verbose=None)
Full response including the response of the grids
```

rohf

Non-relativistic ROHF analytical nuclear gradients

```python
class pyscf.grad.rohf.Gradients(mf)
Non-relativistic ROHF gradients
```  
```python
pyscf.grad.roks.Grads
alias of Gradients
```  
```python
pyscf.grad.roks.Grad
alias of Gradients
```  
```python
class pyscf.grad.roks.Gradients(mf)
Non-relativistic ROHF gradients
```
tdrhf

pyscf.grad.tdrhf.as_scanner(td_grad, state=1)
Generates a nuclear gradients scanner/solver (for geometry optimizer).

The returned solver is a function. This function requires one argument “mol” as input and returns energy and first order nuclear derivatives.

The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters assigned in the nuc-grad object and 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:

```python
>>> from pyscf import gto, scf, tdsf, grad
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1')
>>> td_grad_scanner = scf.RHF(mol).apply(tdsf.TDA).nuc_grad_method().as_scanner()
>>> e_tot, grad = td_grad_scanner(gto.M(atom='H 0 0 0; F 0 0 1.1'))
>>> e_tot, grad = td_grad_scanner(gto.M(atom='H 0 0 0; F 0 0 1.5'))
```

drdhf_slow
drks
tduhf
tduks

uccsd

UCCSD analytical nuclear gradients

uccsd_t

ucisd

UCISD analytical nuclear gradients

uhf

Non-relativistic unrestricted Hartree-Fock analytical nuclear gradients

pyscf.grad.uhf.Grad
alias of Gradients

class pyscf.grad.uhf.Grad(method)
Non-relativistic unrestricted Hartree-Fock gradients

pyscf.grad.uhf.make_rdm1e(mo_energy, mo_coeff, mo_occ)
Energy weighted density matrix
1.15 hessian — Analytical nuclear Hessian

The hessian module implements the analytical nuclear Hessian for mean-field methods. This module also provides functions for thermo-chemical corrections (entropic corrections) using the harmonic model obtained from the Hessian frequencies.

For example:

```python
from pyscf import gto
mol = gto.M(atom =[['O', 0., 0., 0.],
                  ['H', 0., -0.757, 0.587],
                  ['H', 0., 0.757, 0.587],
                  basis = '631g')
mf = mol.RHF().run()
h = mf.Hessian().kernel()
```

The resulting structure of the Hessian is :\( h[\text{Atom}_1, \text{Atom}_2, \text{Atom}_1_{\text{XYZ}}, \text{Atom}_1_{\text{XYZ}}] \): in this case :\((3,3,3,3)\):

1.15.1 Examples

Relevant examples:
- examples/hessian/01-scf_hessian.py
- examples/hessian/10-thermochemistry.py

1.15.2 Program reference

Spin-restricted Hartree-Fock

Non-relativistic RHF analytical Hessian

```python
class pyscf.hessian.rhf.Hessian(scf_method)
    Non-relativistic restricted Hartree-Fock hessian
    partial_hess_elec(hessobj, mo_energy=None, mo_coeff=None, mo_occ=None, atmlst=None,
                      max_memory=4000, verbose=None)
```

Partial derivative

```python
pyscf.hessian.rhf.get_hcore(mol)
    Part of the second derivatives of core Hamiltonian

pyscf.hessian.rhf.partial_hess_elec(hessobj, mo_energy=None, mo_coeff=None, mo_occ=None, atmlst=None,
                                    max_memory=4000, verbose=None)
```

Partial derivative

```python
pyscf.hessian.rhf.solve_mol(mf, mo_energy, mo_coeff, mo_occ, h1ao_or_chkfile, fx=None,
                            atmlst=None, max_memory=4000, verbose=None)
```

Solve the first order equation

Kwargs:
fx [function(dm_mo) => v1_mo] A function to generate the induced potential. See also the function gen_vind.

Spin-unrestricted Hartree-Fock

Non-relativistic UHF analytical Hessian

```python
class pyscf.hessian.uhf.Hessian(scf_method)
    Non-relativistic UHF hessian
```

Spin-restricted DFT

Non-relativistic RKS analytical Hessian

```python
class pyscf.hessian.rks.Hessian(mf)
    Non-relativistic RKS hessian
```

Spin-unrestricted DFT

Non-relativistic UKS analytical Hessian

```python
class pyscf.hessian.uks.Hessian(mf)
    Non-relativistic UKS hessian
```

Thermo-chemistry analysis

Thermochemistry analysis.

**Ref:** psi4/psi4/driver/qcdb/vib.py

```python
pyscf.hessian.thermo.harmonic_analysis(mol, hess, exclude_trans=True, exclude_rot=True, imaginary_freq=True)
```

Each column is one mode

- imaginary_freq (boolean): save imaginary_freq as complex number (if True) or negative real number (if False)

```python
pyscf.hessian.thermo.rotation_const(mass, atom_coords, unit='GHz')
```

Rotational constants to characterize rotational spectra

**Kwargs:**
- unit (string): One of GHz, wavenumber

```python
pyscf.hessian.thermo.rotational_symmetry_number(mol)
```

Number of unique orientations of the rigid molecule that only interchange identical atoms.

**Source** [http://cccbdb.nist.gov/thermo.asp](http://cccbdb.nist.gov/thermo.asp) (search "symmetry number")

### 1.16 soscf — Second order Hartree-Fock solver

The soscf module implements second-order SCF optimization.
1.16.1 Program reference

main class

Co-iterative augmented hessian second order SCF solver (CIAH-SOSCF)

```
pyscf.soscf.newton_ah.newton(mf)
```

Co-iterative augmented hessian (CIAH) second order SCF solver

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1', basis='cc-pvdz')
>>> mf = scf.RHF(mol).run(conv_tol=.5)
>>> mf = scf.newton(mf).set(conv_tol=1e-9)
>>> mf.kernel()
-1.0811707843774987
```

soscf.ciah

1.17 sgx — Pseudo-spectral methods (COSX, PS, SN-K)

The `sgx` module implements pseudo-spectral methods for Hartree-Fock exchange.

Here is a simple example of how to use pseudo spectral integrals in SCF calculation:

```
from pyscf import gto
from pyscf import scf
from pyscf import sgx
mol = gto.M(
    atom='''O 0. 0. 0.
    H 0. -0.757 0.587
    H 0. 0.757 0.587''',
    basis = 'cc-pvdz',
)
mf = sgx.sgx_fit(scf.RHF(mol))
mf.kernel()
# Using RI for Coulomb matrix while K-matrix is constructed with COS-X method
mf.with_df[dfj] = True
mf.kernel()
```

1.17.1 Examples

Relevant examples `examples/sgx/00-simple_sgx.py`

1.17.2 Program reference

Main class

Pseudo-spectral methods (COSX, PS, SN-K)

```
pyscf.sgx.sgx.sgx_fit(mf, auxbasis=None, with_df=None)
```

For the given SCF object, update the J, K matrix constructor with corresponding SGX or density fitting integrals.
Args: mf: an SCF object

Kwargrs:

auxbasis [str or basis dict] Same format to the input attribute mol.basis. If auxbasis is None, optimal auxiliary basis based on AO basis (if possible) or even-tempered Gaussian basis will be used.

Returns: An SCF object with a modified J, K matrix constructor which uses density fitting integrals to compute J and K

Examples:

```python
cf = scf.RHF(mol)
cf.scf()  
scf = scf.UHF(mol)
cf.scf()
```

Get JK

semi-grid Coulomb and eXchange without differencial density matrix

To lower the scaling of coulomb and exchange matrix construction for large system, one coordinate is analitical and the other is grid. The traditional two electron integrals turn to analytical one electron integrals and numerical integration based on grid.(see Friesner, R. A. Chem. Phys. Lett. 1985, 116, 39)


Grid screening for weighted AO value and DktXkg. Two SCF steps: coarse grid then fine grid. There are 5 parameters can be changed: # threshold for Xg and Fg screening gthrd = 1e-10 # initial and final grids level grdlvl_i = 0 grdlvl_f = 1 # norm_ddm threshold for grids change thrd_nddm = 0.03 # set block size to adapt memory sblk = 200

Set mf.direct_scf = False because no traditional 2e integrals

1.18 tddft — Time dependent density functional theory

The tddft module implements TDSCF methods.

An example to run a TDDFT calculation:: from pyscf import gto, scf, dft, tddft mol = gto.Mol(atom='H 0 0 0; F 0 0 1', basis='ccpvdz', verbose=0) mf = dft.RKS(mol) mf.xc = 'b3lyp' mf.kernel() mytd = tddft.TDDFT(mf) mytd.kernel() mytd.analyze()

One can perform NTO analysis for TDDFT as:: weights_1, nto_1 = mytd.get_nlo(state=1, verbose=4) weights_2, nto_2 = mytd.get_nlo(state=2, verbose=4) weights_3, nto_3 = mytd.get_nlo(state=3, verbose=4)

1.18.1 Examples

Relevant examples examples/tddft/00-simple_tddft.py examples/tddft/01-nto_analysis.py examples/tddft/21-matrix_A_B.py examples/tddft/30-change_xc_grids.py examples/tddft/31-energy_transfer_coupling_matrix.py
1.18.2 Program reference

1.18.3 TDHF

pyscf.tdscf.rhf.CIS
   alias of TDA

pyscf.tdscf.rhf.RPA
   alias of TDHF

class pyscf.tdscf.rhf.TDA(mf)
   Tamm-Dancoff approximation

   Attributes:
   conv_tol [float] Diagonalization convergence tolerance. Default is 1e-9.
   nstates [int] Number of TD states to be computed. Default is 3.

   Saved results:
   converged [bool] Diagonalization converged or not
   e [1D array] excitation energy for each excited state.
   xy [A list of two 2D arrays] The two 2D arrays are Excitation coefficients X (shape [nocc,nvir])
      and de-excitation coefficients Y (shape [nocc,nvir]) for each excited state. (X,Y) are normal-
      ized to 1/2 in RHF/RKS methods and normalized to 1 for UHF/UKS methods. In the TDA
      calculation, Y = 0.

   as_scanner (td)
   Generating a scanner/solver for TDA/TDHF/TDDFT PES.

   The returned solver is a function. This function requires one argument “mol” as input and returns total
   TDA/TDHF/TDDFT energy.

   The solver will automatically use the results of last calculation as the initial guess of the new calculation.
   All parameters assigned in the TDA/TDDFT 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, tdscf
   >>> mol = gto.M(atom='H 0 0 0; F 0 0 1')
   >>> td_scanner = tdscf.TDHF(scf.RHF(mol)).as_scanner()
   >>> de = td_scanner(gto.M(atom='H 0 0 0; F 0 0 1.1'))
   [ 0.34460866 0.34460866 0.7131453 ]
   >>> de = td_scanner(gto.M(atom='H 0 0 0; F 0 0 1.5'))
   [ 0.14844013 0.14844013 0.47641829]

   e_tot
   Excited state energies

   gen_vind (mf)
   Compute Ax

   get_ab (mf=None)
   A and B matrices for TDDFT response function.
   A[i,a,j,b] = delta_{ab}delta_{ij}(E_a - E_i) + (ia||bj) B[i,a,j,b] = (ial|jb)
get_nlo (tdobj, state=1, threshold=0.3, verbose=None)
Natural transition orbital analysis.

The natural transition density matrix between ground state and excited state \( T_{ia} = \langle \Psi_e | i a^\dagger | \Psi_0 \rangle \) can be transformed to diagonal form through SVD \( T = O \sqrt{\Lambda} V^\dagger \). \( O \) and \( V \) are occupied and virtual natural transition orbitals. The diagonal elements \( \lambda \) are the weights of the occupied-virtual orbital pair in the excitation.

Ref: Martin, R. L., JCP, 118, 4775-4777

Note in the TDHF/TDDFT calculations, the excitation part (X) is interpreted as the CIS coefficients and normalized to 1. The de-excitation part (Y) is ignored.

Args:

state [int] Excited state ID. state = 1 means the first excited state. If state < 0, state ID is counted from the last excited state.

Kwargs:

threshold [float] Above which the NTO coefficients will be printed in the output.

Returns: A list (weights, NTOs). NTOs are natural orbitals represented in AO basis. The first \( N_{\text{occ}} \) NTOs are occupied NTOs and the rest are virtual NTOs.
xy [A list of two 2D arrays] The two 2D arrays are Excitation coefficients X (shape [nocc,nvir]) and de-excitation coefficients Y (shape [nocc,nvir]) for each excited state. (X,Y) are normalized to 1/2 in RHF/RKS methods and normalized to 1 for UHF/UKS methods. In the TDA calculation, Y = 0.

**gen_vind** *(mf)*
Generate function to compute

\[ \begin{bmatrix} A & B \\ X & Y \end{bmatrix} \]

**kernel** *(x0=None, nstates=None)*
TDHF diagonalization with non-Hermitian eigenvalue solver

```
pyscf.tdscf.rhf.TDRHF
alias of TDHF
```

```
pyscf.tdscf.rhf.as_scanner(id)
Generating a scanner/solver for TDA/TDHF/TDDFT PES.
The returned solver is a function. This function requires one argument "mol" as input and returns total TDA/TDHF/TDDFT energy.
The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters assigned in the TDA/TDDFT 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, tdscf
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1')
>>> td_scanner = tdscf.TDHF(scf.RHF(mol)).as_scanner()
>>> de = td_scanner(gto.M(atom='H 0 0 0; F 0 0 1.1'))
[ 0.34460866 0.34460866 0.7131453 ]
```

```
pyscf.tdscf.rhf.gen_tda_hop(mf, fock_ao=None, singlet=True, wfnsym=None)
Generate function to compute (A+B)x
Kwargs:

  wfnsym [int or str] Point group symmetry irrep symbol or ID for excited CIS wavefunction.
```

```
pyscf.tdscf.rhf.gen_tda_operation(mf, fock_ao=None, singlet=True, wfnsym=None)
Generate function to compute (A+B)x
Kwargs:

  wfnsym [int or str] Point group symmetry irrep symbol or ID for excited CIS wavefunction.
```

```
pyscf.tdscf.rhf.gen_tdhf_operation(mf, fock_ao=None, singlet=True, wfnsym=None)
Generate function to compute
\[ \begin{bmatrix} A & B \\ -B & -A \end{bmatrix} \]
```

```
pyscf.tdscf.rhf.get_ab(mf, mo_energy=None, mo_coeff=None, mo_occ=None)
A and B matrices for TDDFT response function.
\[ A[i,a,j,b] = \delta_{ab} \delta_{ij} (E_a - E_i) + (iallb) B[i,a,j,b] = (iallj) \]
```

---

**1.18. tddft — Time dependent density functional theory**
pyscf.tdscf.rhf.get_nto(tdobj, state=1, threshold=0.3, verbose=None)
Natural transition orbital analysis.

The natural transition density matrix between ground state and excited state $T_{ia} = \langle \Psi_{ex} | i | \Psi_0 \rangle$ can be transformed to diagonal form through SVD $T = O \sqrt{\Lambda} V^\dagger$. $O$ and $V$ are occupied and virtual natural transition orbitals. The diagonal elements $\lambda$ are the weights of the occupied-virtual orbital pair in the excitation.

Ref: Martin, R. L., JCP, 118, 4775-4777

Note in the TDHF/TDDFT calculations, the excitation part (X) is interpreted as the CIS coefficients and normalized to 1. The de-excitation part (Y) is ignored.

Args:

state [int] Excited state ID. state = 1 means the first excited state. If state < 0, state ID is counted from the last excited state.

Kwargs:

threshold [float] Above which the NTO coefficients will be printed in the output.

Returns: A list (weights, NTOs). NTOs are natural orbitals represented in AO basis. The first N_occ NTOs are occupied NTOs and the rest are virtual NTOs.

pyscf.tdscf.rhf.transition_dipole(tdobj, xy=None)
Transition dipole moments in the length gauge

pyscf.tdscf.rhf.transition_magnetic_dipole(tdobj, xy=None)
Transition magnetic dipole moments (imaginary part only)

pyscf.tdscf.rhf.transition_magnetic_quadrupole(tdobj, xy=None)
Transition magnetic quadrupole moments (imaginary part only)

pyscf.tdscf.rhf.transition_octupole(tdobj, xy=None)
Transition octupole moments in the length gauge

pyscf.tdscf.rhf.transition_quadrupole(tdobj, xy=None)
Transition quadrupole moments in the length gauge

pyscf.tdscf.rhf.transition_velocity_dipole(tdobj, xy=None)
Transition dipole moments in the velocity gauge (imaginary part only)

pyscf.tdscf.rhf.transition_velocity_octupole(tdobj, xy=None)
Transition octupole moments in the velocity gauge (imaginary part only)

pyscf.tdscf.rhf.transition_velocity_quadrupole(tdobj, xy=None)
Transition quadrupole moments in the velocity gauge (imaginary part only)

pyscf.tdscf.uhf.gen_tda_hop(mf, fock_ao=None, wfnsym=None)
(A+B)x

Kwargs:

wfnsym [int or str] Point group symmetry irrep symbol or ID for excited CIS wavefunction.

pyscf.tdscf.uhf.gen_tda_operation(mf, fock_ao=None, wfnsym=None)
(A+B)x

Kwargs:

wfnsym [int or str] Point group symmetry irrep symbol or ID for excited CIS wavefunction.

pyscf.tdscf.uhf.gen_tdhf_operation(mf, fock_ao=None, singlet=True, wfnsym=None)
Generate function to compute

$[ A B] [X] [-B -A] [Y]$
pyscf.tdscf.uhf.get_ab (mf, mo_energy=None, mo_coeff=None, mo_occ=None)
A and B matrices for TDDFT response function.

\[ A[i,a,j,b] = \delta_{ab}\delta_{ij}(E_a - E_i) + (ia||bj) \]
\[ B[i,a,j,b] = (ia||jb) \]
Spin symmetry is considered in the returned A, B lists. List A has three items: (A_aaaa, A_aabb, A_bbbb). A_bbaa = A_aabb.transpose(2,3,0,1). B has three items: (B_aaaa, B_aabb, B_bbbb). B_bbaa = B_aabb.transpose(2,3,0,1).

pyscf.tdscf.uhf.get_nto (tdobj, state=1, threshold=0.3, verbose=None)
Natural transition orbital analysis.
The natural transition density matrix between ground state and excited state 
\[ T \psi = \langle \psi_{exc} | i_a^\dagger | \psi_0 \rangle \] can be trans-formed to diagonal form through SVD 
\[ T = O\sqrt{\Lambda}V^\dagger \]. O and V are occupied and virtual natural transition orbitals. The diagonal elements \( \Lambda \) are the weights of the occupied-virtual orbital pair in the excitation.

Ref: Martin, R. L., JCP, 118, 4775-4777
Note in the TDHF/TDDFT calculations, the excitation part (X) is interpreted as the CIS coefficients and normalized to 1. The de-excitation part (Y) is ignored.

Args:

state [int] Excited state ID. state = 1 means the first excited state. If state < 0, state ID is counted from the last excited state.

Kwargs:

threshold [float] Above which the NTO coefficients will be printed in the output.

Returns: A list (weights, NTOs). NTOs are natural orbitals represented in AO basis. The first N_occ NTOs are occupied NTOs and the rest are virtual NTOs.

1.18.4 TDDFT

class pyscf.tdscf.rks.TDDFTNoHybrid (mf)
Solve (A-B)(A+B)(X+Y) = (X+Y)w^2

kernel (x0=None, nstates=None)
TDDFT diagonalization solver

pyscf.tdscf.rks.tddft (mf)
Driver to create TDDFT or TDDFTNoHybrid object

class pyscf.tdscf.uleks.TDDFTNoHybrid (mf)
Solve (A-B)(A+B)(X+Y) = (X+Y)w^2

kernel (x0=None, nstates=None)
TDDFT diagonalization solver

pyscf.tdscf.uleks.tddft (mf)
Driver to create TDDFT or TDDFTNoHybrid object

1.18.5 TDHF_slow

This and other _slow modules implement the time-dependent procedure. The primary performance drawback is that, unlike other ‘fast’ routines with an implicit construction of the eigenvalue problem, these modules construct TDHF matrices explicitly. As a result, regular numpylinalg.eig can be used to retrieve TDHF roots in a reliable fashion without any issues related to the Davidson procedure.

This is a helper module defining basic interfaces.

1.18. tddft — Time dependent density functional theory 257
pyscf.tdscf.common_slow.ab2full\((a, b)\)
Transforms A and B TD matrices into a full matrix. Args:
\(a\) (numpy.ndarray): TD A-matrix; \(b\) (numpy.ndarray): TD B-matrix;

Returns: The full TD matrix.

pyscf.tdscf.common_slow.ab2mkk\((a, b, \text{tolerance}=1e-12)\)
Transforms A and B TD matrices into MK and K matrices. Args:
\(a\) (numpy.ndarray): TD A-matrix; \(b\) (numpy.ndarray): TD B-matrix; \text{tolerance} (float): a tolerance for checking whether the input matrices are real;

Returns: MK and K submatrices.

pyscf.tdscf.common_slow.eig\((m, \text{driver}=\text{None}, \text{nroots}=\text{None}, \text{half}=\text{True})\)
Eigenvalue problem solver. Args:
\(m\) (numpy.ndarray): the matrix to diagonalize; \text{driver} (str): one of the drivers; \text{nroots} (int): the number of roots to calculate (ignored for \text{driver} == ‘eig’); \text{half} (bool): if True, implies spectrum symmetry and takes only a half of eigenvalues;

Returns:

pyscf.tdscf.common_slow.format_frozen_k\((\text{frozen}, \text{nmo}, \text{nk})\)
Formats the argument into a mask array of bools where False values correspond to frozen orbitals for each k-point. Args:
\text{frozen} (int, Iterable): the number of frozen valence orbitals or the list of frozen orbitals for all k-points or multiple lists of frozen orbitals for each k-point; \text{nmo} (int): the total number of molecular orbitals; \text{nk} (int): the total number of k-points;

Returns: The mask array.

pyscf.tdscf.common_slow.format_frozen_mol\((\text{frozen}, \text{nmo})\)
Formats the argument into a mask array of bools where False values correspond to frozen molecular orbitals. Args:
\text{frozen} (int, Iterable): the number of frozen valence orbitals or the list of frozen orbitals; \text{nmo} (int): the total number of molecular orbitals;

Returns: The mask array.

pyscf.tdscf.common_slow.format_mask\((x)\)
Formats a mask into a readable string. Args:
\(x\) (ndarray): an array with the mask;

Returns: A readable string with the mask.

pyscf.tdscf.common_slow.full2ab\((\text{full, tolerance}=1e-12)\)
Transforms a full TD matrix into A and B parts. Args:
\text{full} (numpy.ndarray): the full TD matrix; \text{tolerance} (float): a tolerance for checking whether the full matrix is in the ABBA-form;

Returns: A and B submatrices.
pyscf.tdscf.common_slow.full2mkk (full)
Transforms a full TD matrix into MK and K parts. Args:
    full (numpy.ndarray): the full TD matrix;

Returns: MK and K submatrices.

pyscf.tdscf.common_slow.k_nmo (model)
Retrieves number of AOs per k-point. Args:
    model (RHF): the model;

Returns: Numbers of AOs in the model.

pyscf.tdscf.common_slow.k_nocc (model)
Retrieves occupation numbers. Args:
    model (RHF): the model;

Returns: Numbers of occupied orbitals in the model.

pyscf.tdscf.common_slow.kernel (eri, driver=None, fast=True, nroots=None, **kwargs)
Calculates eigenstates and eigenvalues of the TDHF problem. Args:
    eri (TDDFTMatrixBlocks): ERI; driver (str): one of the eigenvalue problem drivers; fast (bool): whether to run diagonalization on smaller matrices; nroots (int): the number of roots to calculate; **kwargs: arguments to eri.tdhf_matrix;

Returns: Positive eigenvalues and eigenvectors.

pyscf.tdscf.common_slow.mkk2ab (mk, k)
Transforms MK and M TD matrices into A and B matrices. Args:
    mk (numpy.ndarray): TD MK-matrix; k (numpy.ndarray): TD K-matrix;

Returns: A and B submatrices.

pyscf.tdscf.common_slow.mkk2full (mk, k)
Transforms MK and M TD matrices into a full TD matrix. Args:
    mk (numpy.ndarray): TD MK-matrix; k (numpy.ndarray): TD K-matrix;

Returns: The full TD matrix.

pyscf.tdscf.common_slow.mknj2i (item)
Transforms “mknj” notation into tensor index order for the ERI. Args:
    item (str): an arbitrary transpose of “mknj” letters;

Returns: 4 indexes.

pyscf.tdscf.common_slow.msize (m)
Checks whether the matrix is square and returns its size. Args:
    m (numpy.ndarray): the matrix to measure;

Returns: An integer with the size.
This and other _slow modules implement the time-dependent Hartree-Fock procedure. The primary performance drawback is that, unlike other ‘fast’ routines with an implicit construction of the eigenvalue problem, these modules construct TDHF matrices explicitly via an AO-MO transformation, i.e. with a $O(N^5)$ complexity scaling. As a result, regular `numpy.linalg.eig` can be used to retrieve TDHF roots in a reliable fashion without any issues related to the Davidson procedure. Several variants of TDHF are available:

- (this module) `pyscf.tdscf.rhf_slow`: the molecular implementation;
- `pyscf.pbc.tdscf.rhf_slow`: PBC (periodic boundary condition) implementation for RHF objects of `pyscf.pbc.scf` modules;
- `pyscf.pbc.tdscf.krhf_slow_supercell`: PBC implementation for KRHF objects of `pyscf.pbc.scf` modules. Works with an arbitrary number of k-points but has an overhead due to an effective construction of a supercell.
- `pyscf.pbc.tdscf.krhf_slow_gamma`: A Gamma-point calculation resembling the original `pyscf.pbc.tdscf.krhf` module. Despite its name, it accepts KRHF objects with an arbitrary number of k-points but finds only few TDHF roots corresponding to collective oscillations without momentum transfer;
- `pyscf.pbc.tdscf.krhf_slow`: PBC implementation for KRHF objects of `pyscf.pbc.scf` modules. Works with an arbitrary number of k-points and employs k-point conservation (diagonalizes matrix blocks separately).

`pyscf.tdscf.rhf_slow.vector_to_amplitudes` *(vectors, nocc, nmo)*

Transforms (reshapes) and normalizes vectors into amplitudes. Args:

- vectors (numpy.ndarray): raw eigenvectors to transform;
- nocc (int): number of occupied orbitals;
- nmo (int): the total number of orbitals;

**Returns:** Amplitudes with the following shape: (# of roots, 2 (x or y), # of occupied orbitals, # of virtual orbitals).

This and other proxy modules implement the time-dependent mean-field procedure using the existing pyscf implementations as a black box. The main purpose of these modules is to overcome the existing limitations in pyscf (i.e. real-only orbitals, davidson diagonalizer, incomplete Bloch space, etc). The primary performance drawback is that, unlike the original pyscf routines with an implicit construction of the eigenvalue problem, these modules construct TD matrices explicitly by proxying to pyscf density response routines with a $O(N^4)$ complexity scaling. As a result, regular `numpy.linalg.eig` can be used to retrieve TD roots. Several variants of proxy-TD are available:

- (this module) `pyscf.tdscf.proxy`: the molecular implementation;
- `pyscf.pbc.tdscf.proxy`: PBC (periodic boundary condition) Gamma-point-only implementation;
- `pyscf.pbc.tdscf.kproxy_supercell`: PBC implementation constructing supercells. Works with an arbitrary number of k-points but has an overhead due to ignoring the momentum conservation law. In addition, works only with time reversal invariant (TRI) models: i.e. the k-point grid has to be aligned and contain at least one TRI momentum.
- `pyscf.pbc.tdscf.kproxy`: same as the above but respect the momentum conservation and, thus, diagonalizes smaller matrices (the performance gain is the total number of k-points in the model).

`pyscf.tdscf.proxy.mk_make_canonic` *(m, o, v, return_ov=False, space_ov=None)*

Makes the output of pyscf TDDFT matrix (MK form) to be canonic. Args:

- m (ndarray): the TDDFT matrix;
- o (ndarray): occupied orbital energies;
- v (ndarray): virtual orbital energies;
- return_ov (bool): if True, returns the K-matrix as well;
- space_ov (ndarray): an optional ov space;

**Returns:** The rotated matrix as well as an optional K-matrix.

`pyscf.tdscf.proxy.molecular_response` *(vind, space, nocc, nmo, double, log_dest)*

Retrieves a raw response matrix. Args:
vind (Callable): a pyscf matvec routine; space (ndarray): the active orbital space mask: either the same mask for both rows and columns (1D array) or separate orbital masks for rows and columns (2D array); nocc (int): the number of occupied orbitals (frozen and active); nmo (int): the total number of orbitals; double (bool): set to True if vind returns the double-sized (i.e. full) matrix; log_dest (object): pyscf logging;

Returns: The TD matrix.

pyscf.tdscf.proxy.molecular_response_ov(vind, space_ov, nocc, nmo, double, log_dest)
Retrieves a raw response matrix. Args:

vind (Callable): a pyscf matvec routine; space_ov (ndarray): the active ov space mask: either the same mask for both rows and columns (1D array) or separate ov masks for rows and columns (2D array); nocc (int): the number of occupied orbitals (frozen and active); nmo (int): the total number of orbitals; double (bool): set to True if vind returns the double-sized (i.e. full) matrix; log_dest (object): pyscf logging;

Returns: The TD matrix.

Note: The runtime scales with the size of the column mask space_ov[1] but not the row mask space_ov[0].

pyscf.tdscf.proxy.orb2ov(space, nocc)
Converts orbital active space specification into ov-pairs space spec. Args:

space (ndarray): the orbital space; nocc (int): the number of occupied orbitals;

Returns: The ov space specification.

### 1.19 tdscf — Time dependent SCF

This is an alias module of tddft — Time dependent density functional theory

### 1.20 mp — MP2

The mp module implements 2nd order Moller-Plesset perturbation theory from various mean-field references.

#### 1.20.1 Program reference

Moller-Plesset perturbation theory

restricted MP2

RMP2

pyscf.mp.mp2.as_scanner(mp)
Generating a scanner/solver for MP2 PES.

The returned solver is a function. This function requires one argument “mol” as input and returns total MP2 energy.
The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters assigned in the MP2 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:

```python
>>> from pyscf import gto, scf, mp

>>> mol = gto.M(atom='H 0 0 0; F 0 0 1')

>>> mp2_scanner = mp.MP2(scf.RHF(mol)).as_scanner()

>>> e_tot = mp2_scanner(gto.M(atom='H 0 0 0; F 0 0 1.1'))

>>> e_tot = mp2_scanner(gto.M(atom='H 0 0 0; F 0 0 1.5'))
```

`pyscf.mp.mp2.get_frozen_mask(mp)`  
Get boolean mask for the restricted reference orbitals.

In the returned boolean (mask) array of frozen orbital indices, the element is False if it corresponds to the frozen orbital.

`pyscf.mp.mp2.make_rdm1(mp, t2=None, eris=None, verbose=3, ao_repr=False)`  
Spin-traced one-particle density matrix. The occupied-virtual orbital response is not included.

\[
dm1[p,q] = <q_\alpha^dagger p_\alpha> + <q_\beta^dagger p_\beta>
\]

The convention of 1-pdm is based on McWeeney’s book, Eq (5.4.20). The contraction between 1-particle Hamiltonian and rdm1 is \(E = \text{einsum}(pq,qp,h1,rdm1)\)

**Kwargs:**

- `ao_repr` [boolean] Whether to transform 1-particle density matrix to AO representation.

`pyscf.mp.mp2.make_rdm2(mp, t2=None, eris=None, verbose=3)`  
Spin-traced two-particle density matrix in MO basis

\[
dm2[p,q,r,s] = \sum_{\sigma,\tau} <p_\sigma^dagger r_\tau^dagger s_\tau q_\sigma>
\]

Note the contraction between ERIs (in Chemist’s notation) and rdm2 is \(E = \text{einsum}(pqrspqrs , eri , rdm2)\)

unrestricted MP2

UMP2 with spatial integrals

`pyscf.mp.ump2.get_frozen_mask(mp)`  
Get boolean mask for the unrestricted reference orbitals.

In the returned boolean (mask) array of frozen orbital indices, the element is False if it corresponds to the frozen orbital.

`pyscf.mp.ump2.make_rdm1(mp, t2=None, ao_repr=False)`  
One-particle spin density matrices dm1a, dm1b in MO basis (the occupied-virtual blocks due to the orbital response contribution are not included).

\[
dm1a[p,q] = <q_\alpha^dagger p_\alpha> \quad dm1b[p,q] = <q_\beta^dagger p_\beta>
\]

The convention of 1-pdm is based on McWeeney’s book, Eq (5.4.20).

`pyscf.mp.ump2.make_rdm2(mp, t2=None)`  
Two-particle spin density matrices dm2aa, dm2ab, dm2bb in MO basis

\[
dm2aa[p,q,r,s] = <q_\alpha^dagger s_\alpha^dagger r_\alpha^dagger p_\alpha> \quad dm2ab[p,q,r,s] = <q_\alpha^dagger s_\beta^dagger r_\alpha^dagger p_\beta>
\]

\[
dm2bb[p,q,r,s] = <q_\beta^dagger s_\beta^dagger r_\beta^dagger p_\beta>
\]
(p,q correspond to one particle and r,s correspond to another particle) Two-particle density matrix should be contracted to integrals with the pattern below to compute energy

\[
E = \text{numpy.einsum}(\text{'pqrs,pqrs', \text{eri}_{aa}, \text{dm2}_{aa}) E+= \text{numpy.einsum}(\text{'pqrs,pqrs', \text{eri}_{ab}, \text{dm2}_{ab})
E+= \text{numpy.einsum}(\text{'pqrs,rsq', \text{eri}_{ba}, \text{dm2}_{ab}) E+= \text{numpy.einsum}(\text{'pqrs,pqrs', \text{eri}_{bb}, \text{dm2}_{bb})
\]

where \(\text{eri}_{aa}[p,q,r,s] = \langle p_{\alpha} q_{\alpha} | r_{\alpha} s_{\alpha} \rangle\) \(\text{eri}_{ab}[p,q,r,s] = \langle p_{\alpha} q_{\alpha} | r_{\beta} s_{\beta} \rangle\)
\(\text{eri}_{ba}[p,q,r,s] = \langle p_{\beta} q_{\beta} | r_{\alpha} s_{\alpha} \rangle\) \(\text{eri}_{bb}[p,q,r,s] = \langle p_{\beta} q_{\beta} | r_{\beta} s_{\beta} \rangle\)

**MP2 with density fitting**

density fitting MP2, 3-center integrals incore.

**GMP2**

GMP2 in spin-orbital form \(E(\text{MP2}) = 1/4 <ij|ab><ab|i+j-ae-eb)\)

\[
\text{pyscf.mp.gmp2.make_rdm1}(\text{mp}, t2=None, ao_repr=False)
\]

One-particle density matrix in the molecular spin-orbital representation (the occupied-virtual blocks from the orbital response contribution are not included).

\(\text{dm1}[p,q] = \langle q^{\dagger} p \rangle\) (p,q are spin-orbitals)

The convention of 1-pdm is based on McWeeney’s book, Eq (5.4.20). The contraction between 1-particle Hamiltonian and rdm1 is \(E = \text{einsum}(\text{'pq,qp', h1, rdm1})\)

\[
\text{pyscf.mp.gmp2.make_rdm2}(\text{mp}, t2=None)
\]

Two-particle density matrix in the molecular spin-orbital representation

\(\text{dm2}[p,q,r,s] = \langle p^{\dagger} r^{\dagger} s q \rangle\)

where p,q,r,s are spin-orbitals. p,q correspond to one particle and r,s correspond to another particle. The contraction between ERIs (in Chemist’s notation) and rdm2 is \(E = \text{einsum}(\text{'pqrs,pqrs', eri, rdm2})\)

**MP2 F12**

MP2-F12 (In testing)

Refs: * JCC 32 2492 * JCP 139 084112

With strong orthogonalization ansatz 2

### 1.21 ci — Configuration interaction

The `ci` module implements the truncated CI model. A simple example for cisd:

```python
import pyscf
mol = pyscf.M(atom = 'H 0 0 0; F 0 0 1.1', basis = 'ccpvdz')
mf = mol.HF().run()
mycc = mf.CISD().run()
```

**1.21.1 Examples**

00-simple_cisd.py 01-density_matrix.py 20-from_fci.py 32-wfn_overlap.py
1.21.2 Program reference

**ci.cisd**

Solve CISD equation \( H \cdot \mathbf{C} = \mathbf{C} \cdot e \) where \( e = E_{\text{HF}} + E_{\text{CORR}} \)

**Class** *pyscf.ci.cisd.CISD*(\(mf\), \(frozen=0\), \(mo\_coeff=None\), \(mo\_occ=None\))

restricted CISD

**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-9.
- **max_cycle** [int] max number of iterations. Default is 50.
- **max_space** [int] Davidson diagonalization space size. Default is 12.
- **direct** [bool] AO-direct CISD. Default is False.
- **async_io** [bool] Allow for asynchronous function execution. Default is True.
- **frozen** [int or list] If integer is given, the inner-most orbitals are frozen from CI amplitudes. Given the orbital indices (0-based) in a list, both occupied and virtual orbitals can be frozen in CI calculation.

``` python
>>> mol = gto.M(atom = 'H 0 0 0; F 0 0 1.1', basis = 'ccpvdz')
>>> mf = scf.RHF(mol).run()
>>> # freeze 2 core orbitals
>>> myci = ci.CISD(mf).set(frozen = 2).run()
>>> # freeze 2 core orbitals and 3 high lying unoccupied orbitals
>>> myci.set(frozen = [0,1,16,17,18]).run()
```

**Saved results**

- **converged** [bool] CISD converged or not
- **e_corr** [float] CISD correlation correction
- **e_tot** [float] Total CCSD energy (HF + correlation)
- **ci** : CI wavefunction coefficients

**as_scanner** *(ci)*

Generating a scanner/solver for CISD PES.

The returned solver is a function. This function requires one argument “mol” as input and returns total CISD energy.

The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters assigned in the CISD 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:**

``` python
>>> from pyscf import gto, scf, ci
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1')
>>> ci_scanner = ci.CISD(scf.RHF(mol)).as_scanner()
```
get_frozen_mask (mp)
   Get boolean mask for the restricted reference orbitals.
   In the returned boolean (mask) array of frozen orbital indices, the element is False if it corresponds to the
   frozen orbital.

make_rdm1 (myci, civec=None, nmo=None, nocc=None, ao_repr=False)
   Spin-traced one-particle density matrix in MO basis (the occupied-virtual blocks from the orbital response
   contribution are not included).
   \[ dm1[p,q] = \langle q_\alpha^\dagger p_\alpha \rangle + \langle q_\beta^\dagger p_\beta \rangle \]
   The convention of 1-pdm is based on McWeeney's book, Eq (5.4.20). The contraction between 1-particle
   Hamiltonian and rdm1 is \( E = \text{einsum}(\'pqqp\', h1, rdm1) \)

make_rdm2 (myci, civec=None, nmo=None, nocc=None)
   Spin-traced two-particle density matrix in MO basis
   \[ dm2[p,q,r,s] = \sum_{\sigma,\tau} \langle p_\sigma^\dagger r_\tau^\dagger s_\tau q_\sigma \rangle \]
   Note the contraction between ERIs (in Chemist's notation) and rdm2 is \( E = \text{einsum}(\'pqrs,pqrs\', \text{eri}, rdm2) \)

trans_rdm1 (myci, cibra, ciket, nmo=None, nocc=None)
   Spin-traced one-particle transition density matrix in MO basis.
   \[ dm1[p,q] = \langle q_\alpha^\dagger p_\alpha \rangle + \langle q_\beta^\dagger p_\beta \rangle \]
   The convention of 1-pdm is based on McWeeney's book, Eq (5.4.20). The contraction between 1-particle
   Hamiltonian and rdm1 is \( E = \text{einsum}(\'pqqp\', h1, rdm1) \)

vector_size ()
   The size of the vector which was returned from amplitudes_to_cisdvec()

pyscf.ci.cisd.as_scanner (ci)
   Generating a scanner/solver for CISD PES.
   The returned solver is a function. This function requires one argument “mol” as input and returns total CISD
   energy.
   The solver will automatically use the results of last calculation as the initial guess of the new calculation. All
   parameters assigned in the CISD 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, ci
   >>> mol = gto.M(atom='H 0 0 0; F 0 0 1.1')
   >>> ci_scanner = ci.CISD(scf.RHF(mol)).as_scanner()
   >>> e_tot = ci_scanner(gto.M(atom='H 0 0 0; F 0 0 1.1'))
   >>> e_tot = ci_scanner(gto.M(atom='H 0 0 0; F 0 0 1.5'))

pyscf.ci.cisd.from_fci_vec (ci0, norb, nelec, frozen=0)
   Extract CISD coefficients from FCI coefficients

pyscf.ci.cisd.make_rdm1 (myci, civec=None, nmo=None, nocc=None, ao_repr=False)
   Spin-traced one-particle density matrix in MO basis (the occupied-virtual blocks from the orbital response
   contribution are not included).
dm1[p,q] = \langle q^\text{dagger} p \rangle + \langle q^\beta \text{dagger} p^\beta \rangle

The convention of 1-pdm is based on McWeeney’s book, Eq (5.4.20). The contraction between 1-particle Hamiltonian and rdm1 is $E = \text{einsum}^{('pq,qp', h1, rdm1)}$

pyscf.ci.cisd.make_rdm2 (myci, civec=None, nmo=None, nocc=None)
Spin-traced two-particle density matrix in MO basis
$dm2[p,q,r,s] = \text{sum}_{\{\sigma,\tau\}} \langle p^\sigma \text{dagger} r^\tau \text{dagger} s^\tau q^\sigma \rangle$

Note the contraction between ERIs (in Chemist’s notation) and rdm2 is $E = \text{einsum}^{('pqrs,pqrs', eri, rdm2)}$

pyscf.ci.cisd.overlap (cibra, ciket, nmo, nocc, s=None)
Overlap between two CISD wavefunctions.

Args:
- s [2D array] The overlap matrix of non-orthogonal one-particle basis

pyscf.ci.cisd.t1strs (norb, nelec)
Compute the FCI strings (address) for CIS single-excitation amplitudes and the signs of the coefficients when transferring the reference from physics vacuum to HF vacuum.

pyscf.ci.cisd.tn_addrss_signs (norb, nelec, n_excite)
Compute the FCI strings (address) for CIS n-excitation amplitudes and the signs of the coefficients when transferring the reference from physics vacuum to HF vacuum.

pyscf.ci.cisd.to_fcivec (cisivec, norb, nelec, frozen=0)
Convert CISD coefficients to FCI coefficients

pyscf.ci.cisd.trans_rdm1 (myci, cibra, ciket, nmo=None, nocc=None)
Spin-traced one-particle transition density matrix in MO basis.
$dm1[p,q] = \langle q^\text{dagger} p \rangle + \langle q^\beta \text{dagger} p^\beta \rangle$

The convention of 1-pdm is based on McWeeney’s book, Eq (5.4.20). The contraction between 1-particle Hamiltonian and rdm1 is $E = \text{einsum}^{('pq,qp', h1, rdm1)}$

**ci.addons**

**ci.gcisd**

General spin-orbital CISD

pyscf.ci.gcisd.from_rcisdvec (civec, nocc, orbspin)
Convert the (spin-separated) CISD coefficient vector to GCISD coefficient vector

pyscf.ci.gcisd.from_ucisdvec (civec, nocc, orbspin)
Convert the (spin-separated) CISD coefficient vector to GCISD coefficient vector

pyscf.ci.gcisd.make_rdm1 (myci, civec=None, nmo=None, nocc=None, ao_repr=False)
One-particle density matrix in the molecular spin-orbital representation (the occupied-virtual blocks from the orbital response contribution are not included).
$dm1[p,q] = \langle p^\text{dagger} \rangle (p,q \text{ are spin-orbitals})$

The convention of 1-pdm is based on McWeeney’s book, Eq (5.4.20). The contraction between 1-particle Hamiltonian and rdm1 is $E = \text{einsum}^{('pq,qp', h1, rdm1)}$

pyscf.ci.gcisd.make_rdm2 (myci, civec=None, nmo=None, nocc=None)
Two-particle density matrix in the molecular spin-orbital representation
$dm2[p,q,r,s] = \langle p^\text{dagger} r^\text{dagger} s q \rangle$
where p,q,r,s are spin-orbitals. p,q correspond to one particle and r,s correspond to another particle. The contraction between ERIs (in Chemist’s notation) and rdm2 is $E = \text{einsum('pqrs,pqrs', eri, rdm2)}$

```python
def to_ucisdvec(civec, nmo, noec, orbspin):
    # Convert the GCISD coefficient vector to UCISD coefficient vector

def trans_rdm1(myci, cibra, ciket, nmo=None, nocc=None):
    # One-particle transition density matrix in the molecular spin-orbital representation.
    # dm1[p,q] = <$q^dagger p> (p,q are spin-orbitals)
    # The convention of 1-pdm is based on McWeeney’s book, Eq (5.4.20). The contraction between 1-particle Hamiltonian and rdm1 is $E = \text{einsum('pq,qp', h1, rdm1)}$
```

ci.ucisd

Unrestricted CISD

```python
def from_fcivec(ci0, norb, nelec, frozen=0):
    # Extract CISD coefficients from FCI coefficients

def make_rdm1(myci, civec=None, nmo=None, nocc=None, ao_repr=True):
    # One-particle spin density matrices dm1a, dm1b in MO basis (the occupied-virtual blocks due to the orbital response contribution are not included).
    # dm1a[p,q] = <$q_{\alpha}^dagger p_{\alpha}> dm1b[p,q] = <$q_{\beta}^dagger p_{\beta}>
    # The convention of 1-pdm is based on McWeeney’s book, Eq (5.4.20).

def make_rdm2(myci, civec=None, nmo=None, nocc=None):
    # Two-particle spin density matrices dm2aa, dm2ab, dm2bb in MO basis
    # dm2aa[p,q,r,s] = <$q_{\alpha}^dagger r_{\alpha}^dagger p_{\alpha}> dm2ab[p,q,r,s] = <$q_{\alpha}^dagger r_{\beta}^dagger p_{\alpha}>
    # dm2bb[p,q,r,s] = <$q_{\beta}^dagger r_{\beta}^dagger p_{\beta}>
    # (p,q correspond to one particle and r,s correspond to another particle) Two-particle density matrix should be contracted to integrals with the pattern below to compute energy
    # $E = \text{numpy.einsum('pqrs,pqrs', eri_aa, dm2_aa)} E+= \text{numpy.einsum('pqrs,pqrs', eri_ab, dm2_ab)} E+= \text{numpy.einsum('pqrs,pqrs', eri_ba, dm2_ab)} E+= \text{numpy.einsum('pqrs,pqrs', eri_bb, dm2_bb)}$
    # where eri_aa[p,q,r,s] = (p_{\alpha} q_{\alpha} | r_{\alpha} s_{\alpha}) eri_ab[p,q,r,s] = (p_{\alpha} q_{\alpha} | r_{\beta} s_{\beta}) eri_ba[p,q,r,s] = (p_{\beta} q_{\beta} | r_{\alpha} s_{\alpha}) eri_bb[p,q,r,s] = (p_{\beta} q_{\beta} | r_{\beta} s_{\beta})$

def overlap(cibra, ciket, nmo, nocc, s=None):
    # Overlap between two CISD wavefunctions.
    # Args:
    # s [a list of 2D arrays] The overlap matrix of non-orthogonal one-particle basis


def to_fcivec(cisdvec, norb, nelec, frozen=0):
    # Convert CISD coefficients to FCI coefficients

def trans_rdm1(myci, cibra, ciket, nmo=None, nocc=None):
    # One-particle spin density matrices dm1a, dm1b in MO basis (the occupied-virtual blocks due to the orbital response contribution are not included).
    # dm1a[p,q] = <$q_{\alpha}^dagger p_{\alpha}> dm1b[p,q] = <$q_{\beta}^dagger p_{\beta}>
    # The convention of 1-pdm is based on McWeeney’s book, Eq (5.4.20).
1.22 doci — Doubly occupied configuration interaction

The doci implements doubly-occupied configuration interaction. A minimal example on N2 calculation:

```python
from pyscf import gto, doci
g = gto.M(atom='N 0 0; N 0 0 2.', basis='6-31g')
mf = g.RHF().run()
m = doci.CASSCF(mf, 18, 14)
m.kernel()
```

1.22.1 Examples

Relevant examples examples/doci/00-simple_doci_casscf.py

1.22.2 Program reference

DOCI and DOCI based MCSCF

doci_mcsf

```python
class pyscf.doci.doci_mcsf.CASCI(mf_or_mol, ncas, nelecas, ncore=None)
    DOCI-CASCI

class pyscf.doci.doci_mcsf.CASSCF(mf_or_mol, ncas, nelecas, ncore=None, frozen=None)
    DOCI-CASSCF
```

doci_slow

Doubly occupied configuration interaction (DOCI)

```python
from pyscf.doci.doci_slow.contract_2e(eri, civec, norb, nelec, link_index=None)
    Compute E_[pq]E_[rs]|CI>
```

1.23 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:

```python
from pyscf import gto, scf, cc
g = gto.M(atom='H 0 0 0; F 0 0 1', basis='ccpvdz')
mf = scf.RHF(g).run()
cc = cc.CCSD(mf)
cc.kernel()
```

Unrelaxed density matrices are evaluated in the MO basis:

```python
dm1 = cc.make_rdm1()
dm2 = cc.make_rdm2()
```
The CCSD(T) energy can be obtained by:

```python
from pyscf.cc import ccsd_t
print(ccsd_t.kernel(mycc, mycc.ao2mo())[0])
```

Gradients are available:

```python
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:

```python
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:

```python
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.

1.23. cc — Coupled cluster
1.23.1 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.

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.

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.

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.

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.

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.
Interfering CCSD-DIIS

Restart CCSD

1.23.2 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, \(T_1\) and \(T_2\) amplitudes are stored in the CCSD object. This class supports the calculation of CCSD 1- and 2-particle density matrices.

```python
class pyscf.cc.ccsd.CCSD(mf, frozen=0, mo_coeff=None, mo_occ=None)
```

This class is used to perform restricted CCSD calculations. It takes a `Molecule` object or a `SCF` object as input and allows for the specification of various parameters to control the calculation, such as:

- `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.
- `iterative_damping` [float] The self consistent damping parameter.
- `direct` [bool] AO-direct CCSD. Default is False.
- `incore_complete` [bool] Avoid all I/O (also for DIIS). Default is False.
- `level_shift` [float] A shift on virtual orbital energies to stabilize the CCSD iteration
- `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.

```python
>>> 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)
- \(t_1, t_2\) : T amplitudes \(t_1[i,a], t_2[i,j,a,b]\) (i,j in occ, a,b in virt)
- \(l_1, l_2\) : Lambda amplitudes \(l_1[i,a], l_2[i,j,a,b]\) (i,j in occ, a,b in virt)
RCCSD for real integrals 8-fold permutation symmetry has been used \( (ijkl) = (jikl) = (klij) = \ldots \)

```
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.
- **iterative_damping** [float] The self consistent damping parameter.
- **direct** [bool] AO-direct CCSD. Default is False.
- **async_io** [bool] Allow for asynchronous function execution. Default is True.
- **incore_complete** [bool] Avoid all I/O (also for DIIS). Default is False.
- **level_shift** [float] A shift on virtual orbital energies to stablize the CCSD iteration
- **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 \( t_1[i,a], t_2[i,j,a,b] \) (i,j in occ, a,b in virt)
- **l1, l2** : Lambda amplitudes \( l_1[i,a], l_2[i,j,a,b] \) (i,j in occ, a,b in virt)

```
EOMEA_Ta (obj, *args, **kwargs)
Class for EOM EACCSD(T)*(a) method by Matthews and Stanton.
```

```
EOMIP_Ta (obj, *args, **kwargs)
Class for EOM IPCCSD(T)*(a) method by Matthews and Stanton.
```

```
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:

```python
>>> 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 = cc_scanner(gto.M(atom='H 0 0 0; F 0 0 1.1'))
>>> e_tot = cc_scanner(gto.M(atom='H 0 0 0; F 0 0 1.5'))
```

energy (mycc, t1=None, t2=None, eris=None)
CCSD correlation energy

get_frozen_mask (mp)
Get boolean mask for the restricted reference orbitals.
In the returned boolean (mask) array of frozen orbital indices, the element is False if it corresonds to the frozen orbital.

make_rdm1 (t1=None, t2=None, l1=None, l2=None, ao_repr=False)
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
\[ dm2[p,r,q,s] = \langle p^+ q^+ s r \rangle \]

restore_from_diis_ (mycc, diis_file, inplace=True)
Reuse an existed DIIS object in the CCSD calculation.
The CCSD amplitudes will be restored from the DIIS object to generate t1 and t2 amplitudes. The t1/t2 amplitudes of the CCSD object will be overwritten by the generated t1 and t2 amplitudes. The amplitudes vector and error vector will be reused in the CCSD calculation.
pyscf.cc.ccsd.energy (mycc, t1=None, t2=None, eris=None)
CCSD correlation energy

pyscf.cc.ccsd.get_d1_diagnostic (t1)
D1 diagnostic given in

pyscf.cc.ccsd.get_d2_diagnostic (t2)
D2 diagnostic given in
Note: This is currently only defined in the literature for restricted closed-shell systems.

pyscf.cc.ccsd.get_t1_diagnostic (t1)
Returns the t1 amplitude norm, normalized by number of correlated electrons.

pyscf.cc.ccsd.restore_from_diis_(mycc, diis_file, inplace=True)
Reuse an existed DIIS object in the CCSD calculation.
The CCSD amplitudes will be restored from the DIIS object to generate t1 and t2 amplitudes. The t1/t2 amplitudes of the CCSD object will be overwritten by the generated t1 and t2 amplitudes. The amplitudes vector and error vector will be reused in the CCSD calculation.

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 implementation 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.

Restricted CCSD implementation which supports both real and complex integrals. The 4-index integrals are saved on disk entirely (without using any symmetry). This code is slower than the pyscf.cc.ccsd implementation.

Note MO integrals are treated in chemist’s notation

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.
energy (cc, t1=None, t2=None, eris=None)
CCSD correlation energy

class pyscf.cc.rccsd.energy (cc, t1=None, t2=None, eris=None)
CCSD correlation energy
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.

```python
class pyscf.cc.uccsd.UCCSD(mf, frozen=0, mo_coeff=None, mo_occ=None)
```

UCCSD with spatial integrals

```python
pyscf.cc.uccsd.energy(cc, t1=None, t2=None, eris=None)
```

UCCSD correlation energy

cc.addons

Helper functions for CCSD, RCCSD and UCCSD modules are implemented in `cc.addons`

```python
pyscf.cc.addons.spatial2spin(tx, orbspin=None)
```

Convert T1/T2 of spatial orbital representation to T1/T2 of spin-orbital representation

```python
pyscf.cc.addons.spatial2spinorb(tx, orbspin=None)
```

Convert T1/T2 of spatial orbital representation to T1/T2 of spin-orbital representation

CCSD(T)

RHF-CCSD(T) for real integrals

CCSD gradients

```python
pyscf.cc.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:

```python
>>> 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'))
```

1.24 gw — Molecular G0W0

The `gw` module provides an exact $N^6$ scaling GW implementation for eigenvalues.
1.24.1 Program reference

G0W0 approximation

G0W0 approximation

```python
class pyscf.gw.gw.GW (mf, tdmf, frozen=0)
    non-relativistic restricted GW

Saved results
    mo_energy : Orbital energies
    mo_coeff : Orbital coefficients
```

```python
get_frozen_mask (mp)
    Get boolean mask for the restricted reference orbitals.
    In the returned boolean (mask) array of frozen orbital indices, the element is False if it corresponds to the
    frozen orbital.
```

```python
pyscf.gw.gw.kernel (gw, mo_energy, mo_coeff, td_e, td_xy, eris=None, orbs=None, verbose=3)
    GW-corrected quasiparticle orbital energies
    Returns: A list : converged, mo_energy, mo_coeff
```

Slow version

This module implements the G0W0 approximation on top of `pyscf.tdscf.rhf_slow` and `pyscf.tdscf.proxy` TD implementations. Unlike `gw.py`, all integrals are stored in memory. Several variants of GW are available:

- (this module) `pyscf.gw_slow`: the molecular implementation;
- `pyscf.pbc.gw.gw_slow`: single-kpoint PBC (periodic boundary condition) implementation;
- `pyscf.pbc.gw.kgw_slow_supercell`: a supercell approach to PBC implementation with multiple k-points. Runs the molecular code for a model with several k-points for the cost of discarding momentum conservation and using dense instead of sparse matrices;
- `pyscf.pbc.gw.kgw_slow`: a PBC implementation with multiple k-points;

```python
pyscf.gw.gw_slow.corrected_moe (eri, p)
    Calculates the corrected orbital energy. Args:
    eri (PhysERI): a container with electron repulsion integrals; p (int): orbital;
    Returns: The corrected orbital energy.
```

```python
pyscf.gw.gw_slow.kernel (imds, orbs=None, linearized=False, eta=0.001, tol=1e-09, method='fallback')
    Calculates GW energies. Args:
    imds (AbstractIMDS): GW intermediates; orbs (Iterable): indexes of MO orbitals to correct; linearized (bool): whether to apply a single-step linearized correction to energies instead of iterative procedure; eta (float): imaginary energy for the Green’s function; tol (float): tolerance for the search of zero; method (str): ‘bisect’ finds roots no matter what but, potentially, wrong ones, ‘newton’ finding roots close to the correct one but, potentially, failing during iterations, or ‘fallback’ using ‘newton’ and proceeding to ‘bisect’ in case of failure;
    Returns: Corrected orbital energies.
```
1.25 fci — Full configuration interaction

Different FCI solvers are implemented to support different type of 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 (ijkl) = (jilk) = (ijlk) = (jilk)

** Hamiltonian is real but not hermitian, (ijkl) != (jilk) ...

The fci module implements Full Configuration Interaction. Different implementations are available for different Hamiltonian and spin symmetries. One example for FCI calculation on HF molecule:

```python
from pyscf import gto, scf, fci
mol = gto.Mole(atom = 'H 0 0 0; F 0 0 1.1', basis= '6-31g', symmetry=True)
myhf = scf.RHF(mol)
myhf.kernel()
cisolver = fci.FCI(mol, myhf.mo_coeff)
cisolver.kernel()
```

1.25.1 Examples

For a complete list of FCI examples, see pyscf/examples/fci.

1.25.2 Program reference

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.

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 (ijkl) = (jilk) = (ijlk) = (jilk)

** Hamiltonian is real but not hermitian, (ijkl) != (jilk) ...

pyscf.fci.direct_spin1.FC
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 diagonalization.

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

eci [float or a list of float] FCI energy(ies)

Ci [nparray] FCI wfn vector(s)

converged [bool (or a list of bool for multiple roots)] Whether davidson iteration is converged

Examples:

```python
>>> 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 (fle, 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 4-index tensor eri[pqrs] with a FCI vector

\[
|\text{output}\rangle = E_{pq} E_{rs} e r_{pq,rs} |C\rangle \\
E_{pq} E_{rs} = E_{pr,qs} + \delta_{qr} E_{ps} \\
E_{pq} = p^+ q + \bar{p}^+ \bar{q} \\
E_{pr,qs} = p^+ r^+ s q + \bar{p}^+ r^+ s \bar{q} + \ldots
\]

p, q, ... means spin-up orbitals and \( \bar{p}, \bar{q} \) means spin-down orbitals.
Note the input argument eri is NOT the 2e hamiltonian tensor. 2e hamiltonian is

\[ h_{2e} = (pq|rs)E_{pq,rs} \]
\[ = (pq|rs)(E_{pq}E_{rs} - \delta_{qr}E_{ps}) \]
\[ = \text{eri}_{pq,rs}E_{pq}E_{rs} \]

So the relation between eri and hamiltonian (the 2e-integral tensor) is

\[ \text{eri}_{pq,rs} = (pq|rs) - (1/N_{elec}) \sum_q (pq|qs) \]

to restore the symmetry between pq and rs,

\[ \text{eri}_{pq,rs} = (pq|rs) - (5/6N_{elec})[\sum_q (pq|qs) + \sum_p (pq|rp)] \]

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 one-particle density matrix

\[ \text{dm1}_{pq} = <q_{\alpha}^\dagger p_{\alpha}> + <q_{\beta}^\dagger p_{\beta}> \]

The convention is based on McWeeney’s book, Eq (5.4.20) The contraction between 1-particle Hamiltonian and rdm1 is \( E = \text{einsum}(’pq,qp’, \text{h1}, \text{rdm1}) \)

**make_rdm12** *(fcivec, norb, nelec, link_index=None, reorder=True)*

Spin traced 1- and 2-particle density matrices.

\[ \text{1pdm}_{pq} = <q_{\alpha}^\dagger p_{\alpha}> + <q_{\beta}^\dagger p_{\beta}> \]
\[ \text{2pdm}_{pqrs} = <p_{\alpha}^\dagger r_{\alpha}^\dagger s q> + <p_{\beta}^\dagger r_{\beta}^\dagger s q> + <p_{\beta}^\dagger r_{\alpha}^\dagger s q> + <p_{\alpha}^\dagger r_{\beta}^\dagger s q> \]

Energy should be computed as \( E = \text{einsum}(’pq,qp’, \text{h1}, 1\text{pdm}) + 1/2 \times \text{einsum}(’pqrs,pqrs’, \text{eri}, 2\text{pdm}) \)

where \( h1[p,q] = <p|h|q> \) and \( \text{eri}[p,q,r,s] = (pq|rs) \)

**make_rdm12s** *(fcivec, norb, nelec, link_index=None, reorder=True)*

Spin separated 1- and 2-particle density matrices. The return values include two density matrices: (alpha,alpha), (beta,beta)

\[ \text{dm1}_{pq} = <q^\dagger p> \]
\[ \text{dm2}_{pqrs} = <p_{\alpha}^\dagger r_{\alpha}^\dagger s q> + <p_{\beta}^\dagger r_{\beta}^\dagger s q> + <p_{\beta}^\dagger r_{\alpha}^\dagger s q> + <p_{\alpha}^\dagger r_{\beta}^\dagger s q> \]

Energy should be computed as \( E = \text{einsum}(’pq,qp’, \text{h1}, 1\text{pdm}) + 1/2 \times \text{einsum}(’pqrs,pqrs’, \text{eri}, 2\text{pdm}) \)

where \( h1[p,q] = <p|h|q> \) and \( \text{eri}[p,q,r,s] = (pq|rs) \)

**make_rdm1s** *(fcivec, norb, nelec, link_index=None)*

Spin separated 1-particle density matrices. The return values include two density matrices: (alpha,alpha), (beta,beta)

\[ \text{dm1}_{pq} = <q^\dagger p> \]

1.25. **fci** — Full configuration interaction
The convention is based on McWeeny’s book, Eq (5.4.20). The contraction between 1-particle Hamiltonian and rdm1 is \( E = \text{einsum}('pq,qp', h1, rdm1) \)

\[
\text{make_rdm2} \left( \text{fcivec}, \text{norb}, \text{nelec}, \text{link_index=None}, \text{reorder=True} \right)
\]

Spin traced 2-particle density matrix

NOTE the 2pdm is \( \langle p^\dagger q^\dagger sr \rangle \) but stored as [p,r,q,s]

\[
\text{pspace} \left( \text{h1e}, \text{eri}, \text{norb}, \text{nelec}, \text{hdiag=None}, \text{np=400} \right)
\]

pspace Hamiltonian to improve Davidson preconditioner. See, CPL, 169, 463

\[
\text{spin_square} \left( \text{fcivec}, \text{norb}, \text{nelec} \right)
\]

Spin square for RHF-FCI CI wfn only (obtained from spin-degenerated Hamiltonian)

\[
\text{trans_rdm1} \left( \text{cibra}, \text{ciket}, \text{norb}, \text{nelec}, \text{link_index=None} \right)
\]

Spin traced transition 1-particle transition density matrices.

\[
1\text{pdm}[p,q] = \langle q^{\alpha^\dagger} p^{\alpha} \rangle
\]

\[
\text{trans_rdm12} \left( \text{cibra}, \text{ciket}, \text{norb}, \text{nelec}, \text{link_index=None}, \text{reorder=True} \right)
\]

Spin traced transition 1- and 2-particle transition density matrices.

\[
1\text{pdm}[p,q] = \langle q^{\dagger p} \rangle; \quad 2\text{pdm}[p,q,r,s] = \langle p^{\dagger r^{\dagger} s q} \rangle.
\]

\[
\text{trans_rdm12s} \left( \text{cibra}, \text{ciket}, \text{norb}, \text{nelec}, \text{link_index=None}, \text{reorder=True} \right)
\]

Spin separated transition 1-particle density matrices. The return values include two density matrices: (alpha,alpha), (beta,beta). See also function \text{make_rdmis}()

\[
1\text{pdm}[p,q] = \langle q^{\dagger p} \rangle
\]

\[
\text{psycf.fci.direct_spin1.absorb_h1e} \left( \text{h1e}, \text{eri}, \text{norb}, \text{nelec}, \text{fac=1} \right)
\]

Modify 2e Hamiltonian to include 1e Hamiltonian contribution.

\[
\text{psycf.fci.direct_spin1.contract_1e} \left( \text{fle}, \text{fcivec}, \text{norb}, \text{nelec}, \text{link_index=None} \right)
\]

Contract the 1-electron Hamiltonian with a FCI vector to get a new FCI vector.

\[
\text{psycf.fci.direct_spin1.contract_2e} \left( \text{eri}, \text{fcivec}, \text{norb}, \text{nelec}, \text{link_index=None} \right)
\]

Contract the 4-index tensor eri[pqrs] with a FCI vector

\[
|\text{output} \rangle = E_{pq} E_{rs} \text{eri}_{pq,rs} |CI\rangle
\]

\[
E_{pq} E_{rs} = E_{pr,qs} + \delta_{qr} E_{ps}
\]

\[
E_{pq} = p^{\dagger} q + p \bar{q}
\]

\[
E_{pr,qs} = p^{\dagger} r^{\dagger} s q + p^{\dagger} r^{\dagger} s \bar{q} + \ldots
\]

\( p, q, \ldots \) means spin-up orbitals and \( \bar{p}, \bar{q} \) means spin-down orbitals.
Note the input argument eri is NOT the 2e hamiltonian tensor. 2e hamiltonian is

\[ h_{2e} = (pq|rs)E_{pq,qs} = (pq|rs)(E_{pq}E_{rs} - \delta_{qr}E_{ps}) = eri_{pq,rs}E_{pq}E_{rs} \]

So the relation between eri and hamiltonian (the 2e-integral tensor) is

\[ eri_{pq,rs} = (pq|rs) - (1/N_{elc}) \sum_{q} (pq|qs) \]

to restore the symmetry between pq and rs,

\[ eri_{pq,rs} = (pq|rs) - (0.5/N_{elc}) \left[ \sum_{q} (pq|qs) + \sum_{p} (pq|rp) \right] \]

See also \texttt{direct\_spin1.absorb\_h1e()} 
\texttt{pyscf.fci.direct\_spin1.energy(h1e, eri, fcivec, norb, nelec, link\_index=None)}
Compute the FCI electronic energy for given Hamiltonian and FCI vector. 
\texttt{pyscf.fci.direct\_spin1.get\_init\_guess(norb, nelec, nroots, hdiag)}
Initial guess is the single Slater determinant
\texttt{pyscf.fci.direct\_spin1.make\_hdiag(h1e, eri, norb, nelec)}
Diagonal Hamiltonian for Davidson preconditioner
\texttt{pyscf.fci.direct\_spin1.make\_rdm1(fcivec, norb, nelec, link\_index=None)}
Spin-traced one-particle density matrix
\[ dm1[p,q] = \langle q_{\alpha}^dagger p_{\alpha} \rangle + \langle q_{\beta}^dagger p_{\beta} \rangle \]
The convention is based on McWeeney’s book, Eq (5.4.20) The contraction between 1-particle Hamiltonian and rdm1 is E = einsum(’pq,pq’, h1, dm1)
\texttt{pyscf.fci.direct\_spin1.make\_rdm12(fcivec, norb, nelec, link\_index=None, reorder=True)}
Spin traced 1- and 2-particle density matrices.
\[ 1pdm[p,q] = :text{math}:langle q_{alpha}^dagger p_{alpha} rangle + :text{math}:langle q_{beta}^dagger p_{beta} rangle; \]
\[ 2pdm[p,q,r,s] = :text{math}:langle p_{alpha}^dagger r_{alpha}^dagger s_{alpha} q_{alpha} rangle + :text{math}:langle p_{beta}^dagger r_{alpha}^dagger s_{alpha} q_{beta} rangle + \]
\[ :text{math}:langle p_{alpha}^dagger r_{beta}^dagger s_{beta} q_{beta} rangle + :text{math}:langle p_{beta}^dagger r_{beta}^dagger s_{beta} q_{beta} rangle. \]
Energy should be computed as E = einsum(’pq,pq’, h1, 1pdm) + 1/2 * einsum(’pqr,pqrs’, eri, 2pdm) where h1[p,q] = <p|q> and eri[p,q,r] = <pq|rs>
\texttt{pyscf.fci.direct\_spin1.make\_rdm1s(fcivec, norb, nelec, link\_index=None, reorder=True)}
Spin separated 1-particle density matrices. The return values include two density matrices: (alpha,alpha), (beta,beta)
\[ dm1[p,q] = \langle q_{alpha}^dagger p_{alpha} \rangle + \langle q_{beta}^dagger p_{beta} \rangle \]
\texttt{pyscf.fci.direct\_spin1.make\_rdm1s(fcivec, norb, nelec, link\_index=None)}
Spin separated 1-particle density matrices. The return values include two density matrices: (alpha,alpha), (beta,beta)
The convention is based on McWeeney’s book, Eq (5.4.20). The contraction between 1-particle Hamiltonian and rdm1 is 
\[ E = \text{einsum}('pq,qp', h1, \text{rdm1}) \]

```python
pyscf.fci.direct_spin1.pspace(h1e, eri, norb, nelec, hdiag=\text{None}, np=400)
pspace Hamiltonian to improve Davidson preconditioner. See, CPL, 169, 463
```

```python
pyscf.fci.direct_spin1.trans_rdm1(cibra, ciket, norb, nelec, link_index=\text{None})
Spin traced transition 1-particle transition density matrices.
\[ \text{1pdm}[p,q] = \langle q_\alpha^dagger p_\alpha \rangle \]
```

```python
pyscf.fci.direct_spin1.trans_rdm12(cibra, ciket, norb, nelec, link_index=\text{None}, reorder=True)
Spin traced transition 1- and 2-particle transition density matrices.
\[ \text{1pdm}[p,q] = \langle q^p \rangle; \text{2pdm}[p,q,r,s] = \langle p^r s q \rangle. \]
```

```python
pyscf.fci.direct_spin1.trans_rdm1s(cibra, ciket, norb, nelec, link_index=\text{None})
Spin separated transition 1-particle density matrices. The return values include two density matrices: (alpha,alpha), (beta,beta).
```

**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

direct_nosym No No No** Yes

* Real hermitian Hamiltonian implies (ijkl) = (jikl) = (ijlk) = (jilk)

** Hamiltonian is real but not hermitian, (ijkl) != (jikl) ...

```python
pyscf.fci.direct_spin0.contract_1e(fle, fcivec, norb, nelec, link_index=\text{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 4-index tensor eri[pqrs] with a FCI vector

\[
|\text{output}\rangle = E_{pq}E_{rs}\text{eri}_{pq,rs}|\text{CI}\rangle
\]

\[
E_{pq}E_{rs} = E_{pr,qs} + \delta_{qr}E_{ps}
\]

\[
E_{pq} = p^+q + \bar{p}^+\bar{q}
\]

\[
E_{pr,qs} = p^+r^+sq + \bar{p}^+r^+\bar{s}q + \ldots
\]

\(p, q, \ldots\) means spin-up orbitals and \(\bar{p}, \bar{q}\) means spin-down orbitals.

Note the input argument eri is NOT the 2e hamiltonian tensor. 2e hamiltonian is

\[
h_{2e} = (pq|rs)E_{pr,qs}
\]

\[
= (pq|rs)(E_{pq}E_{rs} - \delta_{qr}E_{ps})
\]

\[
= \text{eri}_{pq,rs}E_{pq}E_{rs}
\]

So the relation between eri and hamiltonian (the 2e-integral tensor) is

\[
\text{eri}_{pq,rs} = (pq|rs) - (1/Nelec)\sum_q (pq|qs)
\]

to restore the symmetry between \(pq\) and \(rs\),

\[
\text{eri}_{pq,rs} = (pq|rs) - (5/Nelec)[\sum_q (pq|qs) + \sum_p (pq|rp)]
\]

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 one-particle density matrix

\[
dm1[p,q] = q_\alpha^\dagger p_\alpha + q_\beta^\dagger p_\beta
\]

The convention is based on McWeeney’s book, Eq (5.4.20) The contraction between 1-particle Hamiltonian and rdm1 is \(E = \text{einsum}(‘pqqp’, h1, dm1)\)

pyscf.fci.direct_spin0.make_rdm12(fcivec, norb, nelec, link_index=None, reorder=True)

Spin traced 1- and 2-particle density matrices.

\[
1\text{pdm}[p,q] = :\text{math:‘langle q_\alpha^\dagger p_\alpha rangle} + \text{langle q_\beta^\dagger p_\beta rangle}:
\]

\[
2\text{pdm}[p,q,r,s] = :\text{math:‘langle p_\alpha^\dagger r_\alpha^\dagger s_\alpha q_\alpha rangle} + \text{langle p_\beta^\dagger r_\alpha^\dagger s_\alpha q_\beta rangle} + \text{langle p_\alpha^\dagger r_\beta^\dagger s_\beta q_\alpha rangle} + \text{langle p_\beta^\dagger r_\beta^\dagger s_\beta q_\beta rangle}:
\]

Energy should be computed as \(E = \text{einsum}(‘pqqp’, h1, 1\text{pdm}) + 1/2 \times \text{einsum}(‘pqrs,pqrs’, \text{eri}, 2\text{pdm})\) where \(h1[p,q] = <\text{plh_ql}>\) and \(\text{eri}[p,q,r,s] = (pqrs)\)

pyscf.fci.direct_spin0.make_rdm1s(fcivec, norb, nelec, link_index=None)

Spin separated 1-particle density matrices. The return values include two density matrices: (alpha, alpha), (beta, beta)

\[
dm1[p,q] = q^\dagger p
\]

The convention is based on McWeeney’s book, Eq (5.4.20). The contraction between 1-particle Hamiltonian and rdm1 is \(E = \text{einsum}(‘pqqp’, h1, dm1)\)

1.25. fci — Full configuration interaction 283
PySCF Documentation, Release 1.7.0

pyscf.fci.direct_spin0.trans_rdm1(cibra, ciket, norb, nelec, link_index=None)
Spin traced transition 1-particle transition density matrices.

\[ 1pdm[p,q] = \langle q_\alpha^\dagger p_\alpha \rangle \]

pyscf.fci.direct_spin0.trans_rdm12(cibra, ciket, norb, nelec, link_index=None, reorder=True)
Spin traced transition 1- and 2-particle transition density matrices.

\[ 1pdm[p,q] = \langle q^p \rangle; 2pdm[p,q,r,s] = \langle p^r q^s \rangle. \]

pyscf.fci.direct_spin0.trans_rdm1s(cibra, ciket, norb, nelec, link_index=None)
Spin separated transition 1-particle density matrices. The return values include two density matrices: (alpha,alpha), (beta,beta). See also function make_rdm1s()

\[ 1pdm[p,q] = \langle q^p \rangle \]

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_spin0_symm Yes No Yes Yes direct_spin0 No Yes Yes Yes direct_spin1 No No Yes Yes direct_uhf No No Yes Yes direct_nosym No No No** Yes

• Real hermitian Hamiltonian implies (ijkl) = (jilk) = (ijlk) = (jilk)

** Hamiltonian is real but not hermitian, (ijkl) != (jilk) ...

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_spin0_symm Yes No Yes Yes direct_spin0 No Yes Yes Yes direct_spin1 No No Yes Yes direct_uhf No No Yes Yes direct_nosym No No No** Yes

• Real hermitian Hamiltonian implies (ijkl) = (jilk) = (ijlk) = (jilk)

** Hamiltonian is real but not hermitian, (ijkl) != (jilk) ...

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_cre_str_index_o0(orb_list, nelec)
Slow version of gen_cre_str_index function

pyscf.fci.cistring.gen_cre_str_index_o1(orb_list, nelec)
C implementation of gen_cre_str_index function
pyscf.fci.cistring.\texttt{gen\_des\_str\_index}(\texttt{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.\texttt{gen\_des\_str\_index\_o0}(\texttt{orb\_list, nelec})

Slow version of \texttt{gen\_des\_str\_index} function

pyscf.fci.cistring.\texttt{gen\_des\_str\_index\_o1}(\texttt{orb\_list, nelec})

C implementation of \texttt{gen\_des\_str\_index} function

pyscf.fci.cistring.\texttt{gen\_linkstr\_index}(\texttt{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 excitations, starting from str0, annihilating i, creating a, to get str1.

pyscf.fci.cistring.\texttt{gen\_linkstr\_index\_trilidx}(\texttt{orb\_list, nocc, strs=None})

Generate linkstr\_index with the assumption that $p > q$. So the resultant link\_index has the structure [pq,*,str1,sign]. It is identical to a call to \texttt{reform\_linkstr\_index}\texttt{(gen\_linkstr\_index(...))}.

pyscf.fci.cistring.\texttt{gen\_strings\_4\_orblist}(\texttt{orb\_list, nelec})

Generate string from the given orbital list.

\textbf{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.

\textbf{Examples:}

\begin{verbatim}
>>> [bin(x) for x in make_strings((0,1,2,3),2)]
[0b11, 0b101, 0b110, 0b1001, 0b1010, 0b1100]
>>> [bin(x) for x in make_strings((3,1,0,2),2)]
[0b1010, 0b1001, 0b11, 0b1100, 0b110, 0b101]
\end{verbatim}

pyscf.fci.cistring.\texttt{make\_strings}(\texttt{orb\_list, nelec})

Generate string from the given orbital list.

\textbf{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.

\textbf{Examples:}

\begin{verbatim}
>>> [bin(x) for x in make_strings((0,1,2,3),2)]
[0b11, 0b101, 0b110, 0b1001, 0b1010, 0b1100]
>>> [bin(x) for x in make_strings((3,1,0,2),2)]
[0b1010, 0b1001, 0b11, 0b1100, 0b110, 0b101]
\end{verbatim}

pyscf.fci.cistring.\texttt{reform\_linkstr\_index}(\texttt{link\_index})

Compress the (a, i) pair index in linkstr\_index to a lower triangular index. The compressed indices can match the 4-fold symmetry of integrals.

pyscf.fci.cistring.\texttt{str2addr}(\texttt{norb, nelec, string})

Convert string to CI determinant address

1.25. fci — Full configuration interaction 285
pyscf.fci.cistring.\texttt{strs2addr}(norb, nelec, strings)  
Convert a list of string to CI determinant address

pyscf.fci.cistring.\texttt{sub_addrs}(norb, nelec, orbital_indices, sub_nelec=0)  
The addresses of the determinants which include the specified orbital indices. The size of the returned addresses  
is equal to the number of determinants of (norb, nelec) system.

pyscf.fci.cistring.\texttt{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.

\textbf{spin operator}

pyscf.fci.spin_op.\texttt{contract_ss}(fcivec, norb, nelec)  
Contract spin square operator with FCI wavefunction \(S^2|CI\rangle\)

pyscf.fci.spin_op.\texttt{local_spin}(fcivec, norb, nelec, mo_coeff=None, ovlp=1, aolst=[])  
Local spin expectation value, which is defined as  
\(\langle CI|\text{local } S^2|CI\rangle\)  
The local \(S^2\) operator only couples the orbitals specified in aolst. The cross term which involves the interaction  
between the local part (in aolst) and non-local part (not in aolst) is not included. As a result, the value of  
local\_spin is not additive. In other words, if local\_spin is computed twice with the complementary aolst in the  
two runs, the summation does not equal to the \(S^2\) of the entire system.  
For a complete list of AOs, the value of local\_spin is equivalent to \(\langle CI|S^2|CI\rangle\)

pyscf.fci.spin_op.\texttt{spin_square}(fcivec, norb, nelec, mo_coeff=None, ovlp=1)  
General spin square operator.

\[
\begin{align*}
\langle CI|S_+^* S_-|CI\rangle &= n_\alpha + \delta_{ik}\delta_{jl} Gamma_{i\alpha k\beta \to j\beta l\alpha} \\
\langle CI|S_-^* S_+|CI\rangle &= n_\beta + \delta_{ik}\delta_{jl} Gamma_{i\beta k\alpha \to j\alpha l\beta} \\
\langle CI|S_z*S_z|CI\rangle &= \delta_{ik}\delta_{jl} (Gamma_{i\alpha k\alpha \to j\alpha l\alpha} - Gamma_{i\alpha k\alpha \to j\beta l\beta} - Gamma_{i\beta k\beta \to j\alpha l\alpha} + Gamma_{i\beta k\beta \to j\beta l\beta}) + (n_\alpha+n_\beta)/4
\end{align*}
\]

Given the overlap between non-degenerate alpha and beta orbitals, this function can compute the expectation  
value spin square operator for UHF-FCI wavefunction

pyscf.fci.spin_op.\texttt{spin_square0}(fcivec, norb, nelec)  
Spin square for RHF-FCI CI wfn only (obtained from spin-degenerated Hamiltonian)

\textbf{rdm}

FCI 1, 2, 3, 4-particle density matrices.

Note the 1-particle density matrix has the same convention as the mean-field 1-particle density matrix (see  
McWeeney’s book Eq 5.4.20), which is  
\[dm[p,q] = \langle q^*+p \rangle\]

The contraction between 1-particle Hamiltonian and 1-pdm is  
\[E = \text{einstein}('pqqp', \text{h1}, \text{1pdm})\]
Different conventions are used in the high order density matrices: \( \text{dm}[p,q,r,s,...] = <p^+ r^+ ... s q> \)

```python
pyscf.fci.rdm.make_dm123(fname, cibra, ciket, nobr, nelec)
```
Spin traced 1, 2 and 3-particle density matrices.

**Note:** In this function, 2pdm\([p,q,r,s]\) is \(\langle p^+ q r^+ s \rangle\); 3pdm\([p,q,r,s,t,u]\) is \(\langle p^+ q r^+ s t^+ u \rangle\).

After calling reorder_dm123, the 2pdm and 3pdm are transformed to the normal density matrices: 2pdm\([p,r,q,s]\) = \(\langle p^+ q^+ s r \rangle\); 3pdm\([p,s,q,t,r,u]\) = \(\langle p^+ q^+ r^+ s t u \rangle\).

```python
pyscf.fci.rdm.make_dm1234(fname, cibra, ciket, nobr, nelec)
```
Spin traced 1, 2, 3 and 4-particle density matrices.

**Note:** In this function, 2pdm\([p,q,r,s]\) is \(\langle p^+ q r^+ s \rangle\); 3pdm\([p,q,r,s,t,u]\) is \(\langle p^+ q r^+ s t^+ u \rangle\); 4pdm\([p,q,r,s,t,u,v,w]\) is \(\langle p^+ q^+ r^+ s t^+ u v^+ w \rangle\).

After calling reorder_dm123, the 2pdm and 3pdm are transformed to the normal density matrices: 2pdm\([p,r,q,s]\) = \(\langle p^+ q^+ s r \rangle\); 3pdm\([p,s,q,t,r,u]\) = \(\langle p^+ q^+ r^+ s t u \rangle\); 4pdm\([p,t,q,u,r,v,s,w]\) = \(\langle p^+ q^+ r^+ s t u v^+ w \rangle\).

### addons

```python
pyscf.fci.addons.cre_a(ci0, nobr, 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.

```python
pyscf.fci.addons.cre_b(ci0, nobr, 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.cylindrical_init_guess(mol, norb, nelec, orbsym, wfn_sym=0, singlet=True, nroots=1)

FCI initial guess for system of cylindrical symmetry. (In testing)

Examples:

```python
>>> mol = gto.M(atom='O; O 1 1.2', spin=2, symmetry=True)
>>> orbsym = [6,7,2,3]
>>> ci0 = fci.addons.cylindrical_init_guess(mol, 4, (3,3), orbsym, wfnsym=10)[0]
>>> print(ci0.reshape(4,4))
>>> ci0 = fci.addons.cylindrical_init_guess(mol, 4, (3,3), orbsym, wfnsym=10, singlet=False)[0]
>>> print(ci0.reshape(4,4))
```

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.fix_spin_(fciobj, shift=0.2, ss=None, **kwargs)

If FCI solver cannot stay on spin eigenfunction, this function can add a shift to the states which have wrong spin.

\[ (H + \text{shift} \cdot S^2) |\Psi\rangle = E |\Psi\rangle \]

Args: fciobj : An instance of FCISolver

Kwarg:

- shift [float] Level shift for states which have different spin
ss [number] S^2 expectation 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:
- 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.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:
- 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

1.26 hci — Interface to Heat-Bath selective CI

The HCI module is a native implementation of the Heat-bath selected CI algorithm, where selection is done based on strings rather than determinants. For example:

See also cornell_shci.

1.26.1 Examples

Relevant examples examples/hci/00-simple_hci.py examples/hci/10-spin.py

1.26.2 Program reference

Interface

Selected CI using Heat-Bath CI algorithm (JCTC 2016, 12, 3674-3680)

Simple usage:

```python
.. py:function:: fix_spin(myci, shift=0.2, ss=None, **kwargs)

   module pyscf.hci

   If Selected CI 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: myci : An instance of SelectedCI
   Kwarg:
   shift [float] Level shift for states which have different spin
   ss [number] S^2 expection value == s*(s+1)

   Returns A modified Selected CI object based on myci.
```

pyscf.hci.hci.make_rdm12s(civec, norb, nelec)

Spin orbital 1- and 2-particle reduced density matrices (aa, bb, aaaa, aabb, bbbb)
1.27 cornell_shci — Interface to Fast Semistochastic Heat Bath Configuration Interaction (SHCI)


1.27.1 Examples

Relevant examples

1.27.2 Program reference

1.27.3 shci

1.28 mcscf — Multi-configurational self-consistent field

The mcscf implements orbital optimization for MCSCF and CASSCF. 1-step (combined orbital and wavefunction optimization) and 2-step algorithms (alternating orbital and wavefunction optimization) are available. Different kinds of active space solvers can be used with this module.

For example, a simple CASCI calculation can be run as:

```python
import pyscf

mol = pyscf.M(
    atom = 'O 0 0 0; O 0 0 1.2',
    basis = 'ccpvdz',
    spin = 2)

myhf = mol.RHF().run()

# 6 orbitals, 8 electrons
mycas = myhf.CASCI(6, 8).run()
```

and a simple CASSCF can be run as:

```python
import pyscf

mol = pyscf.M(
    atom = 'O 0 0 0; O 0 0 1.2',
    basis = 'ccpvdz',
    spin = 2)

myhf = mol.RHF().run()

# 6 orbitals, 8 electrons
mycas = myhf.CASSCF(6, 8).run()
```

The CASSCF orbital optimization is general and can be combined with many different solvers, such as DMRG and selected CI solvers.
1.28.1 Examples

Relevant examples

- examples/mcscf/00-simple_casci.py
- examples/mcscf/00-simple_casscf.py
- examples/mcscf/01-for_expensive_fci.py
- examples/mcscf/03-natural_orbital.py
- examples/mcscf/04-density_matrix.py
- examples/mcscf/10-define_cas_space.py
- examples/mcscf/11-casscf_with_uhf_uk.pdf
- examples/mcscf/12-c2_triplet_from_single_hf.py
- examples/mcscf/13-load_chkfile.py
- examples/mcscf/14-project_init_guess.py
- examples/mcscf/15-state_specific.py
- examples/mcscf/15-state_specific.py
- examples/mcscf/16-density_fitting.py
- examples/mcscf/17-approx_orbital_hessian.py
- examples/mcscf/18-02_spatial_spin_symmetry.py
- examples/mcscf/18-spatial_spin_symmetry.py
- examples/mcscf/19-frozen_core.py
- examples/mcscf/20-change_symmetry.py
- examples/mcscf/21-active_space_symmetry.py
- examples/mcscf/21-nosymhf_then_symcasscf.py
- examples/mcscf/22-x2c.py
- examples/mcscf/23-local_spin.py
- examples/mcscf/33-make_init_guess.py
- examples/mcscf/34-init_guess_localization.py
- examples/mcscf/40-customizing_hamiltonian.py
- examples/mcscf/41-mcscf_custom_df_hamiltonian.py
- examples/mcscf/41-state_average.py
- examples/mcscf/42-compare_cas_space.py
- examples/mcscf/43-dmet_cas.py
- examples/mcscf/44-mcscf_active_space_hamiltonian.py
- examples/mcscf/50-casscf_with_selected_ci.py
- examples/mcscf/50-cornell_shci_casscf.py
- examples/mcscf/50-dmrgscf_with_block.py
- examples/mcscf/51-o2_triplet_by_various_fci.py
- examples/mcscf/60-uhf_based_ucasscf.py
- examples/mcscf/61-rcas_vs_ucas
- examples/mcscf/70-casscf_hot_tuning.py
- examples/mcscf/70-casscf_optimize_scheduler.py

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
>>> 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 electron 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 assign the FCISolver to this attribute, e.g.

```python
>>> 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.:

```python
>>> 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_stepsizes** [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 optimizer crosses the barrier of local minimum and converge to the neighbour solution, e.g. the CAS(4,4) for C2H4 in the test files. In these systems, adjusting max_stepsizes, max_ci_stepsizes and max_cycle_micro, max_cycle_micro_inner and ah_start_tol may be helpful)

```python
>>> mc = mcscf.CASSCF(mf, 6, 6)
>>> mc.max_stepsizes = .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_stepsizes** [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 preferred. 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_stepsizes, 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 might 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.

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
1.28.2 CASSCF active space solver

DMRG solver

FCIQMC solver

State-average FCI solver

State-average with mixed solver

1.28.3 Symmetry broken

1.28.4 Initial guess

1.28.5 Canonical orbitals

There are two relevant parameters for orbital canonicalization. They are `mc.canonicalization` and `mc.natorb` (assuming the MCSCF object is `mc`). In the CASCI/CASSCF calculations, the resultant orbitals are stored in the attribute `mc.mo_coeff`. These orbitals may be identical or partially identical to the initial orbitals, depending on the values of `mc.canonicalization` and `mc.natorb`.

`mc.canonicalization` controls whether the resultant CASCI/CASSCF orbitals are canonicalized with respect to the general Fock matrix. General Fock matrix is defined as

\[
F = h_{\text{core}} + J - K
\]

\[
J_{pq} = \sum_{rs} \langle pq|rs \rangle \gamma_{sr}
\]

\[
K_{pq} = \sum_{qr} \langle pq|qr \rangle \gamma_{qr}
\]

\(\gamma\) is the total density matrix which is the summation of doubly occupied core density matrix and correlated density matrix in active space. If `mc.canonicalization` is enabled, the CASCI/CASSCF program will call the `mc.canonicalize()` function to diagonalize the core and external space wrt the general Fock matrix. The eigenvalues of the core and external subspace are stored in attribute `mc.mo_energy`. By default, `mc.canonicalization` is enabled because the canonicalized MCSCF orbitals can simplify the algorithm of MRPT methods.

`mc.natorb` controls whether the CASCI/CASSCF active space orbitals are transformed to natural orbitals of the correlated density matrix. When this parameter is enabled, the natural orbitals will be stored in the active part of the attribute `mc.mo_coeff` and the CI coefficients `mc.ci` (if applicable) will be transformed accordingly. By default `mc.natorb` is disabled and it is important for the MCSCF solver. Generally, the value of `mc.natorb` does not affect (the default) FCI solver because an independent CASCI calculation following a previous MCSCF calculation should give the same solutions no matter `mc.natorb` is enabled or not. But this is not true for some external large active space solvers such as DMRG, selected CI methods. The CASCI calculation may produce different answers depending on the value of `mc.natorb`. Therefore, it is recommended to disable `mc.natorb` in your calculation.

Following presents what the `mc.mo_coeff` would be like for different combinations of `mc.canonicalization` and `mc.natorb` in a CASCI calculation:

- **mc.canonicalization = False and mc.natorb = False:**
  
  The resultant orbitals `mc.mo_coeff` are identical to the input orbitals. If the CASCI was initialized with a RHF calculation, `mc.mo_coeff` points to RHF orbitals.

- **mc.canonicalization = True and mc.natorb = False:**
Core part and external part of `mc.mo_coeff` are canonicalized orbitals, which diagonalize the core and external blocks of general Fock matrix. The orbitals in active space are identical to the active orbitals in the input.

- `mc.canonicalization = False and mc.natorb = True`

Core and external part of `mc.mo_coeff` are identical to the core and external part of the input orbitals. Active space orbitals are transformed to the natural orbitals of the correlated density matrix.

- `mc.canonicalization = True and mc.natorb = True`

`mc.mo_coeff` are completely different to the input orbitals.

There is another parameter `mc.sorting_mo_energy` which may affect the ordering of MCSCF orbitals when `mc.canonicalization` or `mc.natorb` is enabled. Generally, the canonical orbitals in the core and external space are sorted by the orbital energies (from low to high) and the natural orbitals in the active space are sorted by natural occupations (from large to small). This ordering may not be held if point group symmetry is enabled in the calculation. When a system has high spatial symmetry and point group symmetry is enabled, each SCF orbital will be assigned to an irreducible representation label. In the MCSCF calculation and the canonicalization, the irreducible representation label of the orbitals will not be changed. They are always the same to the symmetry labels of the input orbitals. Although the orbitals are still ordered within each irreducible representation, the orbital energies (or occupancies) for all orbitals are not strictly sorted. Setting `mc.sorting_mo_energy = True` (though not recommended) can force the orbitals to be sorted regardless whether the point group symmetry is enabled. In certain scenario, you may want to enable `mc.natorb` and `mc.sorting_mo_energy`. examples/dmrg/31-cr2_scan/cr2-scan.py provides one example that you need to enable the two parameters. In that example, the dissociation curve of Cr dimer was scanned by heat-bath selected-CI method in which the active space of selected-CI-CASSCF was gradually enlarged in a series of CASSCF calculations. Since the selected-CI algorithm depends on the initial single determinant, the orbital ordering do have matters to the final CASSCF results. Thus `mc.natorb` and `mc.sorting_mo_energy` have to be enabled to make sure that the each selected-CI starts from the similar initial reference for each point on the dissociation curve. At some critical points, the difference in the orbital ordering in the active space can lead to discontinuous potential energy curve.

### 1.28.6 Program reference

#### CASCI

class `pyscf.mcscf.casci.CASCI` (mf_or_mol, ncas, nelecas, ncore=None)

**Args:**

- `mf_or_mol` [SCF object or Mole object] SCF or Mole to define the problem size.
- `ncas` [int] Number of active orbitals.
- `nelecas` [int or a pair of int] Number of electrons in active space.

**Kwargs:**

- `ncore` [int] Number of doubly occupied core orbitals. If not presented, this parameter can be automatically determined.

**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 electron numbers.

natorb [bool] Whether to transform natural orbital in active space. Be cautious of this parameter when CASCI/CASSCF are combined with DMRG solver or selected CI solver because DMRG and selected CI are not invariant to the rotation in active space. False by default.

canonicalization [bool] Whether to canonicalize orbitals. Note that canonicalization does not change the orbitals in active space by default. It only diagonalizes core and external space of the general Fock matrix. To get the natural orbitals in active space, attribute natorb need to be enabled. True by default.

sorting_mo_energy [bool] Whether to sort the orbitals based on the diagonal elements of the general Fock matrix. Default is False.

cfisolver [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 assign the FCISolver to this attribute, e.g.

```python
>>> 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.:

```python
>>> 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)
- **e_cas** [float] CAS space FCI energy
- **ci** [ndarray] CAS space FCI coefficients
- **mo_coeff** [ndarray] When canonicalization is specified, the orbitals are canonical orbitals which make the general Fock matrix (Fock operator on top of MCSCF 1-particle density matrix) diagonalized within each subspace (core, active, external). If natorb (natural orbitals in active space) is specified, the active segment of the mo_coeff is natural orbitals.
- **mo_energy** [ndarray] Diagonal elements of general Fock matrix (in mo_coeff representation).

Examples:

```python
>>> 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
```

**Gradients** *(obj, *args, **kwargs)*
Non-relativistic restricted Hartree-Fock gradients
as_scanner \((mc)\)
Generating a scanner for CASCI PES.

The returned solver is a function. This function requires one argument “mol” as input and returns total CASCI energy.

The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters of MCSCF object 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:

```python
>>> from pyscf import gto, scf, mcscf
>>> mf = scf.RHF(gto.Mole().set(verbose=0))
>>> mc_scanner = mcscf.CASCI(mf, 4, 4).as_scanner()
>>> mc_scanner(gto.M(atom='N 0 0 0; N 0 0 1.1'))
>>> mc_scanner(gto.M(atom='N 0 0 0; N 0 0 1.5'))
```

canonicalize \((mc, \text{mo coeff}=None, \text{ci}=None, \text{eris}=None, \text{sort}=False, \text{cas natorb}=False, \text{casdm1}=None, \text{verbose}=3, \text{with meta lowdin}=True)\)
Canonicalized CASCI/CASSCF orbitals of effective Fock matrix and update CI coefficients accordingly.

Effective Fock matrix is built with one-particle density matrix (see also mcscf.casci.get_fock()).
For state-average CASCI/CASSCF object, the canonicalized orbitals are based on the state-average density matrix. To obtain canonicalized orbitals for an individual state, you need to pass “casdm1” of the specific state to this function.

**Args:** mc: a CASSCF/CASCI object or RHF object

**Kwargs:**

- **mo_coeff (ndarray):** orbitals that span the core, active and external space.
- **ci (ndarray):** CI coefficients (or objects to represent the CI wavefunctions in DMRG/QMC-MCSCF calculations).
- **eris:** Integrals for the MCSCF object. Input this object to reduce the overhead of computing integrals. It can be generated by mc.ao2mo() method.
- **sort (bool):** Whether the canonicalized orbitals are sorted based on the orbital energy (diagonal part of the effective Fock matrix) within each subspace (core, active, external). If point group symmetry is not available in the system, orbitals are always sorted. When point group symmetry is available, sort=False will preserve the symmetry label of input orbitals and only sort the orbitals in each symmetry sector. sort=True will reorder all orbitals over all symmetry sectors in each subspace and the symmetry labels may be changed.
- **cas_natorb (bool):** Whether to transform active orbitals to natural orbitals. If enabled, the output orbitals in active space are transformed to natural orbitals and CI coefficients are updated accordingly.
- **casdm1 (ndarray):** 1-particle density matrix in active space. This density matrix is used to build effective fock matrix. Without input casdm1, the density matrix is computed with the input ci coefficients/object. If neither ci nor casdm1 were given, density matrix is computed by mc.fcisolver.make_rdm1() method. For state-average CASCI/CASSCF calculation, this results in a set of canonicalized orbitals of state-average effective Fock matrix. To canonicalize the orbitals for one particular state, you can assign the density matrix of that state to the kwarg casdm1.

**Returns:** A tuple, (natural orbitals, CI coefficients, orbital energies) The orbital energies are the diagonal terms of effective Fock matrix.
canonicalize_ (mo_coeff=None, ci=None, eris=None, sort=False, cas_natorb=False, casdm1=None, verbose=None, with_meta_lowdin=True)

Canonicalized CASCI/CASSCF orbitals of effective Fock matrix and update CI coefficients accordingly.

Effective Fock matrix is built with one-particle density matrix (see also mcscf.casci.get_fock()).

For state-average CASCI/CASSCF object, the canonicalized orbitals are based on the state-average density matrix. To obtain canonicalized orbitals for an individual state, you need to pass "casdm1" of the specific state to this function.

Args:  
mc: a CASSCF/CASCI object or RHF object

Kwargs:

- mo_coeff (ndarray): orbitals that span the core, active and external space.
- ci (ndarray): CI coefficients (or objects to represent the CI wavefunctions in DMRG/QMC-MCSCF calculations).
- eris: Integrals for the MCSCF object. Input this object to reduce the overhead of computing integrals. It can be generated by mc.ao2mo() method.
- sort (bool): Whether the canonicalized orbitals are sorted based on the orbital energy (diagonal part of the effective Fock matrix) within each subspace (core, active, external). If point group symmetry is not available in the system, orbitals are always sorted. When point group symmetry is available, sort=False will preserve the symmetry label of input orbitals and only sort the orbitals in each symmetry sector. sort=True will reorder all orbitals over all symmetry sectors in each subspace and the symmetry labels may be changed.
- cas_natorb (bool): Whether to transform active orbitals to natural orbitals. If enabled, the output orbitals in active space are transformed to natural orbitals and CI coefficients are updated accordingly.
- casdm1 (ndarray): 1-particle density matrix in active space. This density matrix is used to build effective fock matrix. Without input casdm1, the density matrix is computed with the input ci coefficients/object. If neither ci nor casdm1 were given, density matrix is computed by mc.fcisolver.make_rdm1() method. For state-average CASCI/CASCF calculation, this results in a set of canonicalized orbitals of state-average effective Fock matrix. To canonicalize the orbitals for one particular state, you can assign the density matrix of that state to the kwarg casdm1.

Returns:  
A tuple, (natural orbitals, CI coefficients, orbital energies) The orbital energies are the diagonal terms of effective Fock matrix.

cas_natorb (mo_coeff=None, ci=None, eris=None, sort=False, casdm1=None, verbose=None, with_meta_lowdin=True)

Transform active orbitals to natural orbitals, and update the CI wfn accordingly

Args:  
mc : a CASSCF/CASCI object or RHF object

Kwargs:

- sort [bool] Sort natural orbitals wrt the occupancy.

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, with_meta_lowdin=True)

Transform active orbitals to natural orbitals, and update the CI wfn accordingly

Args:  
mc : a CASSCF/CASCI object or RHF object

Kwargs:
sort [bool] Sort natural orbitals wrt the occupancy.

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.

\[ \text{fix\_spin}(\text{shift}=0.2, \text{ss}=\text{None}) \]
Use level shift to control FCI solver spin.

\[ (H + \text{shift} \cdot S^2)|\Psi\rangle = E|\Psi\rangle \]

Kwargs:
- \text{shift} [float] Energy penalty for states which have wrong spin
- \text{ss} [number] \( S^2 \) expectation value \( s(s+1) \)

\[ \text{fix\_spin}\_a(\text{shift}=0.2, \text{ss}=\text{None}) \]
Use level shift to control FCI solver spin.

\[ (H + \text{shift} \cdot S^2)|\Psi\rangle = E|\Psi\rangle \]

Kwargs:
- \text{shift} [float] Energy penalty for states which have wrong spin
- \text{ss} [number] \( S^2 \) expectation value \( s(s+1) \)

\[ \text{get\_h1cas}(\text{casci}, \text{mo\_coeff}=\text{None}, \text{ncas}=\text{None}, \text{ncore}=\text{None}) \]
CAS space one-electron hamiltonian

Args: \text{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.

\[ \text{get\_h1eff}(\text{mo\_coeff}=\text{None}, \text{ncas}=\text{None}, \text{ncore}=\text{None}) \]
CAS space one-electron hamiltonian

Args: \text{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.

\[ \text{get\_h2cas}(\text{mo\_coeff}=\text{None}) \]
Computing active space two-particle Hamiltonian.

Note It is different to get\_h2eff when df.approx_hessian is applied, in which get\_h2eff function returns the DF integrals while get\_h2cas returns the regular 2-electron integrals.

\[ \text{get\_h2eff}(\text{mo\_coeff}=\text{None}) \]
Computing active space two-particle Hamiltonian.

Note It is different to get\_h2cas when df.approx_hessian is applied, in which get\_h2eff function returns the DF integrals while get\_h2cas returns the regular 2-electron integrals.

\[ \text{h1e\_for\_cas}(\text{casci}, \text{mo\_coeff}=\text{None}, \text{ncas}=\text{None}, \text{ncore}=\text{None}) \]
CAS space one-electron hamiltonian

Args: \text{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.

\[ \text{kernel}(\text{mo\_coeff}=\text{None}, \text{ci0}=\text{None}, \text{verbose}=\text{None}) \]
**Returns:** Five elements, they are total energy, active space CI energy, the active space FCI wavefunction coefficients or DMRG wavefunction ID, the MCSCF canonical orbital coefficients, the MCSCF canonical orbital coefficients.

They are attributes of mcscf object, which can be accessed by .e_tot, .e_cas, .ci, .mo_coeff, .mo_energy

```python
def make_rdm1(
    mo_coeff=None,
    ci=None,
    ncas=None,
    nelecas=None,
    ncore=None,
    **kwargs
):
    # One-particle density matrix in AO representation
```

```python
def make_rdm1s(
    mo_coeff=None,
    ci=None,
    ncas=None,
    nelecas=None,
    ncore=None,
    **kwargs
):
    # One-particle density matrices for alpha and beta spin on AO basis
```

```python
def sort_mo(caslst, mo_coeff=None, 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:
```

```python
>>> 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
```

```python
def 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

    MRH, 04/08/2019: Instead of turning casscf_finalize into an instance attribute that points to the previous casscf object, I’m going to make a whole new child class. This will have the added benefit of making state_average and state_average_ actually behave differently for the first time (until now they both modified the casscf object inplace). I’m also going to assign the weights argument as a member of the mc child class because an accurate second-order CASSCF algorithm for state-averaged calculations requires that the gradient and Hessian be computed for CI vectors of each root individually and then multiplied by that root’s weight. The second derivatives computed by newton_casscf.py need to be extended to state-averaged calculations in order to be used as intermediates for calculations of the gradient of a single root in the context of the SA-CASSCF method; see: Mol. Phys. 99, 103 (2001).

    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

```python
mc.fcisolver = fci.solver(mol, False)
```

before calling state_average_(mc), to mix the singlet and triplet states

MRH, 04/08/2019: Instead of turning casscf._finalize into an instance attribute that points to the previous casscf object, I’m going to make a whole new child class. This will have the added benefit of making state_average and state_average_ actually behave differently for the first time (until now they both modified the casscf object in place). I’m also going to assign the weights argument as a member of the mc child class because an accurate second-order CASSCF algorithm for state-averaged calculations requires that the gradient and Hessian be computed for CI vectors of each root individually and then multiplied by that root’s weight. The second derivatives computed by newton_casscf.py need to be extended to state-averaged calculations in order to be used as intermediates for calculations of the gradient of a single root in the context of the SA-CASSCF method; see: Mol. Phys. 99, 103 (2001).

```python
state_specific_(state=1)
```  
For excited state

**Kwargs:** state : int 0 for ground state; 1 for first excited state.

```python
pyscf.mcscf.casci.as_scanner(mc)
```

Generating a scanner for CASCI PES.

The returned solver is a function. This function requires one argument “mol” as input and returns total CASCI energy.

The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters of MCSCF object 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:

```python
>>> from pyscf import gto, scf, mcscf

>>> mf = scf.RHF(gto.Mole().set(verbose=0))

>>> mc_scanner = mcscf.CASCI(mf, 4, 4).as_scanner()

>>> mc_scanner(gto.M(atom='N 0 0 0; N 0 0 1.1'))

>>> mc_scanner(gto.M(atom='N 0 0 0; N 0 0 1.5'))
```

```python
pyscf.mcscf.casci.canonicalize(mc, mo_coeff=None, ci=None, eris=None, sort=False, cas_natorb=False, casdm1=None, verbose=3, with_meta_lowdin=True)
```

Canonicalized CASCI/CASSCF orbitals of effective Fock matrix and update CI coefficients accordingly.

Effective Fock matrix is built with one-particle density matrix (see also mcscf.casci.get_fock()). For state-average CASCI/CASSCF object, the canonicalized orbitals are based on the state-average density matrix.

To obtain canonicalized orbitals for an individual state, you need to pass “casdm1” of the specific state to this function.

**Args:** mc: a CASSCF/CASCI object or RHF object

**Kwargs:**

- **mo_coeff (ndarray):** orbitals that span the core, active and external space.
- **ci (ndarray):** CI coefficients (or objects to represent the CI wavefunctions in DMRG/QMC-MCSCF calculations).
- **eris: Integrals for the MCSCF object. Input this object to reduce the overhead of computing integrals. It can be generated by mc.ao2mo() method.
sort (bool): Whether the canonicalized orbitals are sorted based on the orbital energy (diagonal part of the effective Fock matrix) within each subspace (core, active, external). If point group symmetry is not available in the system, orbitals are always sorted. When point group symmetry is available, sort=False will preserve the symmetry label of input orbitals and only sort the orbitals in each symmetry sector. sort=True will reorder all orbitals over all symmetry sectors in each subspace and the symmetry labels may be changed.

cas_natorb (bool): Whether to transform active orbitals to natural orbitals. If enabled, the output orbitals in active space are transformed to natural orbitals and CI coefficients are updated accordingly.

casdm1 (ndarray): 1-particle density matrix in active space. This density matrix is used to build effective fock matrix. Without input casdm1, the density matrix is computed with the input ci coefficients/object. If neither ci nor casdm1 were given, density matrix is computed by mc.fcisolver.make_rdm1() method. For state-average CASCI/CASCF calculation, this results in a set of canonicalized orbitals of state-average effective Fock matrix. To canonicalize the orbitals for one particular state, you can assign the density matrix of that state to the kwarg casdm1.

Returns: A tuple, (natural orbitals, CI coefficients, orbital energies) The orbital energies are the diagonal terms of effective Fock matrix.

pyscf.mcscf.casci.cas_natorb (mc, mo_coeff=None, ci=None, eris=None, sort=False, casdm1=None, verbose=None, with_meta_lowdin=True)
Transform active orbitals to natural orbitals, and update the CI wavefunction accordingly

Args: mc : a CASSCF/CASCI object or RHF object

Kwarg:

sort [bool] Sort natural orbitals wrt the occupancy.

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)
Effective one-electron Fock matrix in AO representation f = \sum_{pq} E_{pq} F_{pq} = h_{pq} + \sum_{rs} [(pq|rs)-(ps|rq)] DM_{sr}
For state-average CASCI/CASSCF object, the effective fock matrix is based on the state-average density matrix. To obtain Fock matrix of a specific state in the state-average calculations, you can pass “casdm1” of the specific state to this function.

Args: mc: a CASSCF/CASCI object or RHF object

Kwarg:

mo_coeff (ndarray): orbitals that span the core, active and external space.

ci (ndarray): CI coefficients (or objects to represent the CI wavefunctions in DMRG/QMC-MCSCF calculations).

eris: Integrals for the MCSCF object. Input this object to reduce the overhead of computing integrals. It can be generated by mc.ao2mo() method.

casdm1 (ndarray): 1-particle density matrix in active space. Without input casdm1, the density matrix is computed with the input ci coefficients/object. If neither ci nor casdm1 were given, density matrix is computed by mc.fcisolver.make_rdm1() method. For state-average CASCI/CASCF calculation, this results in the effective Fock matrix based on the state-average density matrix. To obtain the effective Fock matrix for one particular state, you can assign the density matrix of that state to the kwarg casdm1.
Returns: Fock matrix

```python
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.

```python
pyscf.mcscf.casci.kernel(casci, mo_coeff=None, ci0=None, verbose=3)
```
CASSCF

UCASCI (CASSCI with non-degenerated alpha and beta orbitals, typically UHF orbitals)

```python
pyscf.mcscf.ucasci.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

```python
pyscf.mcscf.ucasci.kernel(casci, mo_coeff=None, ci0=None, verbose=3)
```
UHF-CASCI solver

CASSCF

class pyscf.mcscf.mclstep.CASSCF(mf_or_mol, ncas, nelecas, ncore=None, frozen=None)

Args:
mf_or_mol [SCF object or Mole object] SCF or Mole to define the problem size.
ncas [int] Number of active orbitals.
nelecas [int or a pair of int] Number of electrons in active space.

Kwargs:
ncore [int] Number of doubly occupied core orbitals. If not presented, this parameter can be automatically determined.

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 element numbers.
natorb [bool] Whether to transform natural orbital in active space. Be cautious of this parameter when CASSCI/CASSCF are combined with DMRG solver or selected CI solver because DMRG and selected CI are not invariant to the rotation in active space. False by default.
canonicalization [bool] Whether to canonicalize orbitals. Note that canonicalization does not change the orbitals in active space by default. It only diagonalizes core and external space of the general Fock matrix. To get the natural orbitals in active space, attribute natorb need to be enabled. True by default.
sorting_mo_energy [bool] Whether to sort the orbitals based on the diagonal elements of the general Fock matrix. Default is False.
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 assign the FCISolver to this attribute, e.g.

```python
>>> 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.:

```python
>>> 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)
- **e_cas** [float] CAS space FCI energy
- **ci** [ndarray] CAS space FCI coefficients
- **mo_coeff** [ndarray] When canonicalization is specified, the orbitals are canonical orbitals which make the general Fock matrix (Fock operator on top of MCSCF 1-particle density matrix) diagonalized within each subspace (core, active, external). If natorb (natural orbitals in active space) is specified, the active segment of the mo_coeff is natural orbitls.
- **mo_energy** [ndarray] Diagonal elements of general Fock matrix (in mo_coeff representation).

Examples:

```python
>>> 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
```

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. 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.

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 environment.

scale_restoration  [float] When a step of orbital rotation moves out of trust region, the orbital optimization will be restored to previous state and the step size of the orbital rotation needs to be reduced. scale_restoration controls how much to scale down the step size.

Saved results

e_tot  [float] Total MCSCF energy (electronic energy plus nuclear repulsion)

e_cas  [float] CAS space FCI energy

ci  [ndarray] CAS space FCI coefficients

mo_coeff  [ndarray] Optimized CASSCF orbitals coefficients. When canonicalization is specified, the returned orbitals make the general Fock matrix (Fock operator on top of MCSCF 1-particle density matrix) diagonalized within each subspace (core, active, external). If natorb (natural orbitals in active space) is specified, the active segment of the mo_coeff is natural orbitals.

mo_energy  [ndarray] Diagonal elements of general Fock matrix (in mo_coeff representation).

Examples:
Gradients *(obj, *args, **kwargs)*
Non-relativistic restricted Hartree-Fock gradients

**as_scanner** *(mc)*
Generating a scanner for CASSCF PES.

The returned solver is a function. This function requires one argument “mol” as input and returns total CASSCF energy.

The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters of MCSCF object (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:

```python
>>> 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)
>>> mc = mcscf.CASSCF(mf, 6, 6)
>>> mc.kernel()[0]
-109.044401882238134
```

get_grad *(mo_coeff=None, casdm1_casdm2=None, eris=None)*
Orbital gradients

get_h2cas *(mo_coeff=None)*
Computing active space two-particle Hamiltonian.

Note It is different to get_h2eff when df.approx_hessian is applied, in which get_h2eff function returns the DF integrals while get_h2cas returns the regular 2-electron integrals.

get_h2eff *(mo_coeff=None)*
Computing active space two-particle Hamiltonian.

Note It is different to get_h2cas when df.approx_hessian is applied, in which get_h2eff function returns the DF integrals while get_h2cas returns the regular 2-electron integrals.

kernel *(mo_coeff=None, ci0=None, callback=None, _kern=<function kernel>)*
Returns: Five elements, they are total energy, active space CI energy, the active space FCI wavefunction coefficients or DMRG wavefunction ID, the MCSCF canonical orbital coefficients, the MCSCF canonical orbital coefficients.

They are attributes of mcscf object, which can be accessed by .e_tot, .e_cas, .ci, .mo_coeff, .mo_energy

rotate_mo *(mo, u, log=None)*
Rotate orbitals with the given unitary matrix

solve_approx_ci *(h1, h2, ci0, ecore, e_cas, envs)*
Solve CI eigenvalue/response problem approximately

**as_scanner** *(mc)*
Generating a scanner for CASSCF PES.
The returned solver is a function. This function requires one argument “mol” as input and returns total CASSCF energy.

The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters of MCSCF object (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:

```python
>>> from pyscf import gto, scf, mcscf

>>> mol = gto.M(atom='N 0 0 0; N 0 0 1.2', verbose=0)

>>> mc_scanner = mcscf.CASSCF(scf.RHF(mol), 4, 4).as_scanner()

>>> e = mc_scanner(gto.M(atom='N 0 0 0; N 0 0 1.1'))

>>> e = mc_scanner(gto.M(atom='N 0 0 0; N 0 0 1.5'))
```

```
pyscf.mcscf.mc1step.kernel(casscf, mo_coeff, tol=1e-07, conv_tol_grad=None, call-back=None, verbose=3, dump_chk=True)

quasi-newton CASSCF optimization driver
```

```
pyscf.mcscf.mc1step_symm.CASSCF

alias of SymAdaptedCASSCF
```

```
class pyscf.mcscf.mc1step_symm.SymAdaptedCASSCF

Args:

  mf_or_mol [SCF object or Mole object] SCF or Mole to define the problem size.

  ncas [int] Number of active orbitals.

  nelecas [int or a pair of int] Number of electrons in active space.

Kwargs:

  ncore [int] Number of doubly occupied core orbitals. If not presented, this parameter can be automatically determined.

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 electron numbers.

  natorb [bool] Whether to transform natural orbital in active space. Be cautious of this parameter when CASCI/CASSCF are combined with DMRG solver or selected CI solver because DMRG and selected CI are not invariant to the rotation in active space. False by default.

  canonicalization [bool] Whether to canonicalize orbitals. Note that canonicalization does not change the orbitals in active space by default. It only diagonalizes core and external space of the general Fock matrix. To get the natural orbitals in active space, attribute natorb need to be enabled. True by default.

  sorting_mo_energy [bool] Whether to sort the orbitals based on the diagonal elements of the general Fock matrix. Default is False.
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 assign the FCISolver to this attribute, e.g.

```python
>>> 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.:  

```python
>>> 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)
- **e_cas** [float] CAS space FCI energy
- **ci** [ndarray] CAS space FCI coefficients
- **mo_coeff** [ndarray] When canonicalization is specified, the orbitals are canonical orbitals which make the general Fock matrix (Fock operator on top of MCSCF 1-particle density matrix) diagonalized within each subspace (core, active, external). If natorb (natural orbitals in active space) is specified, the active segment of the mo_coeff is natural orbitals.
- **mo_energy** [ndarray] Diagonal elements of general Fock matrix (in mo_coeff representation).

Examples:

```python
>>> 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
```

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. 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 allowed 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 [float, 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.

```python
>>> 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()[0]
-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 environment.

scale_restoration [float] When a step of orbital rotation moves out of trust region, the orbital optimization will be restored to previous state and the step size of the orbital rotation needs to be reduced. scale_restoration controls how much to scale down the step size.

Saved results

e_tot [float] Total MCSCF energy (electronic energy plus nuclear repulsion)

e_cas [float] CAS space FCI energy

ci [ndarray] CAS space FCI coefficients

mo_coeff [ndarray] Optimized CASSCF orbitals coefficients. When canonicalization is specified, the returned orbitals make the general Fock matrix (Fock operator on top of MCSCF 1-particle density matrix) diagonalized within each subspace (core, active, external). If natorb (natural orbitals in active space) is specified, the active segment of the mo_coeff is natural orbitals.

mo_energy [ndarray] Diagonal elements of general Fock matrix (in mo_coeff representation).

Examples:
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
sort_mo_by_irrep (cas_irrep_nocc, cas_irrep_ncore=None, mo_coeff=None, s=None)
  Select active space based on symmetry information. See also
  pyscf.mcscf.addons.sort_mo_by_irrep()

UCASSCF (CASSCF without spin-degeneracy between alpha and beta orbitals) 1-step optimization algorithm

MO integrals for UCASSCF methods

addons

pyscf.mcscf.addons.cas_natorb (casscf, mo_coeff=None, ci=None, sort=False)
  Natural 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.make_rdm1 (casscf, mo_coeff=None, ci=None, **kwargs)
One-particle density 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:
```python
def 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)
```

pyscf.mcscf.addons.make_rdm1s (casscf, mo_coeff=None, ci=None, **kwargs)
Alpha and beta one-particle density matrices in AO representation

pyscf.mcscf.addons.map2hf (casscf, mf_mo=None, base=1, tol=0.4)
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 initial 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:
```python
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 reordered mo_coeff, which put the orbitals given by caslst in the CAS space

**Examples:**

```python
>>> 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 functional 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

MRH, 04/08/2019: Instead of turning casscf._finalize into an instance attribute that points to the previous casscf
object, I’m going to make a whole new child class. This will have the added benefit of making state_average
and state_average_ actually behave differently for the first time (until now they both modified the casscf object
inplace). I’m also going to assign the weights argument as a member of the mc child class because an accurate
second-order CASSCF algorithm for state-averaged calculations requires that the gradient and Hessian be computed
for CI vectors of each root individually and then multiplied by that root’s weight. The second derivatives
computed by newton_casscf.py need to be extended to state-averaged calculations in order to be used as inter-
mediates for calculations of the gradient of a single root in the context of the SA-CASSCF method; see: Mol.

pyscf.mcscf.addons.state_average_(casscf, weights=(0.5, 0.5))
Inplace version of state_average

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))
Inplace version of state_average

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.

## 1.29 dmrgrscf

An interface to DMRG and DMRG-SCF/CASSCF. The DMRG calculation must be carried out with an external solver, such as Block (https://sanshar.github.io/Block) or StackBlock (https://github.com/sanshar/StackBlock) or CheMPS2 (https://github.com/SebWouters/CheMPS2). See also mcscf.

To perform a DMRG CASSCF calculation, first modify the pyscf/dmrgscf/settings.py and set the correct path for DMRG solver, and then run e.g. examples/dmrg/01-dmrg_casscf_with_block.py

### 1.29.1 Examples

- examples/dmrg/01-dmrg_casscf_with_block.py
- examples/dmrg/01-dmrg_casscf_with_stackblock.py
- examples/dmrg/02-dmrg_nevpt2.py
- examples/dmrg/03-density_matrix.py
- examples/dmrg/10-state_average.py
- examples/dmrg/11-excited_states.py
- examples/dmrg/30-dmrg_casscf_nevpt2_for_Cr2.py
- examples/dmrg/31-dmrg_casscf_for_feporph.py
- examples/dmrg/32-dmrg_casscf_nevpt2_for_FeS.py

### 1.29.2 Program Reference

```python
pyscf.dmrgscf.dmrg_sym.convert_lzsym(gpname, orbsym)
Convert orbital symmetry irrep_id to Block internal irrep_id
```

```python
pyscf.dmrgscf.dmrg_sym.convert_orbsym(gpname, orbsym)
Convert orbital symmetry irrep_id to Block internal irrep_id
```

DMRG solver for CASCI and CASSCF.

```python
class pyscf.dmrgscf.dmrgci.DMRGCI(mol=None, maxM=None, tol=None, num_thrds=1, memory=None)
```

Block program interface and the object to hold Block program input parameters.

**Attributes:**

- `outputlevel` [int] Noise level for Block program output.
- `maxItr` : int
  - `hf_occ` : str
    - The initial HF wave function occupancies, in spin orbital.
- `approx_maxItr` [int] To control the DMRG-CASSCF approximate DMRG solver accuracy.
- `twodot_to_onedot` [int] When to switch from two-dot algorithm to one-dot algorithm.
- `nroots` : int
- `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 tolerance
- `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.

**Examples:**

```python
>>> mol = gto.M(atom='C 0 0 0; C 0 0 1')
>>> mf = scf.RHF(mol).run()
>>> mc = mcscf.CASCI(mf, 4, 4)
>>> mc.fcisolver = DMRGCI(mol)
>>> mc.kernel()
-74.379770619390698
```

**pyscf.dmrgscf.dmrgci.DMRGSCF**

Shortcut function to setup CASSCF using the DMRG solver. The DMRG solver is properly initialized in this function so that the 1-step algorithm can be applied with DMRG-CASSCF.

**Examples:**

```python
>>> mol = gto.M(atom='C 0 0 0; C 0 0 1')
>>> mf = scf.RHF(mol).run()
>>> mc = DMRGSCF(mf, 4, 4)
>>> mc.kernel()
-74.414908818611522
```

**pyscf.dmrgscf.dmrgci.dryrun**

Generate FCIDUMP and dmrg config file

### 1.30 fciqmcscf

An interface to the NECI (https://github.com/ghb24/NECI_STABLE) FCIQMC solver. See also mcscf.

### 1.31 mrpt — Multi-reference perturbation theory

The mrpt module implements the N-electron valence (multi-reference) perturbation theory.

#### 1.31.1 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:**
```python
>>> 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, nevptsolver=None, tol=1e-07, stored_integral=False)
SC-NEVPT2 with compressed perturber

Kwargs:

- maxM [int] DMRG bond dimension

Examples:

```python
>>> 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

### 1.32 icmpspt — Internal-contracted MPS perturbation method

ic-MPSPT is an interface to a multi-reference perturbation method that uses DMRG (MPS) wavefunctions as the zeroth order wavefunction. The DMRG wavefunction needs to be generated using the `dmrgscf` solver. The ic-MPSPT solver can be obtained from Sandeep Sharma (sanshar@gmail.com).

#### 1.32.1 Program reference

interface

#### 1.33 x2c — exact-two-component approach

An example to apply scalar relativistic effects by decorating the scf object with module `x2c` is shown as following:

```python
from pyscf import gto
from pyscf import scf
mol = gto.M(
    verbose = 0,
    atom = '''8 0 0. 0
1 0 -0.757 0.587
1 0 0.757 0.587''',
    basis = 'ccpvdz',
)
mf = scf.RHF(mol).x2c().run()
mf.spin = 1
mol.charge = 1
mol.build(0, 0)
mf = scf.UKS(mol).x2c1e()
energy = mf.kernel()
```

More examples can be find here:
## 1.33.1 Examples

Relevant examples examples/x2c/01-spin_free_x2c.py examples/x2c/02-basis_for_x.py

## 1.33.2 Program reference

### X2C

```python
class pyscf.x2c.x2c.X2C(mol=None)
    2-component X2c (including spin-free and spin-dependent terms) in the j-adapted spinor basis.

get_hcore(mol=None)
    2-component X2c Foldy-Wouthuysen (FW) Hamiltonian (including spin-free and spin-dependent terms)
    in the j-adapted spinor basis.

reset(mol)
    Reset mol and clean up relevant attributes for scanner mode
```

```python
pyscf.x2c.x2c.get_hcore(mol)
    2-component X2c hcore Hamiltonian (including spin-free and spin-dependent terms) in the j-adapted spinor basis.
```

```python
pyscf.x2c.x2c.get_jk(mol, dm, hermi=1, mf_opt=None, with_j=True, with_k=True, omega=None)
    non-relativistic J/K matrices (without SSO,SOO etc) in the j-adapted spinor basis.
```

```python
pyscf.x2c.x2c.init_guess_by_1e(mol)
    Initial guess from one electron system.
```

```python
pyscf.x2c.x2c.init_guess_by_atom(mol)
    Initial guess from atom calculation.
```

```python
pyscf.x2c.x2c.init_guess_by_minao(mol)
    Initial guess in terms of the overlap to minimal basis.
```

### 1e spin-free x2c

1-electron Spin-free X2C approximation

```python
class pyscf.x2c.sfx2c1e.SpinFreeX2C(mol=None)
    1-component X2c (spin-free part only)

get_hcore(mol=None)
    1-component X2c Foldy-Wouthuysen (FW Hamiltonian (spin-free part only)

pyscf.x2c.sfx2c1e.sfx2c(mf)
    Spin-free X2C. For the given SCF object, it updates the hcore constructor. All integrals are computed in the real spherical GTO basis.

Args:  mf : an SCF object

Returns: An SCF object
```

Examples:

```python
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1', basis='ccpvdz', verbose=0)
>>> mf = scf.RHF(mol).sfx2c1e()
>>> mf.scf()
```
>>> import pyscf.x2c.sfx2c1e
>>> mol.symmetry = 1
>>> mol.build(0, 0)
>>> mf = pyscf.x2c.sfx2c1e.sfx2c1e(scf.UHF(mol))
>>> mf.scf()

**pyscf.x2c.sfx2c1e.sfx2c1e** *(mf)*  
Spin-free X2C. For the given SCF object, it updates the hcore constructor. All integrals are computed in the real spherical GTO basis.

**Args:** mf : an SCF object  
**Returns:** An SCF object  
**Examples:**

```python
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1', basis='ccpvdz', verbose=0)
>>> mf = scf.RHF(mol).sfx2c1e()
>>> mf.scf()
```

### 1e spin-free x2c gradient

Analytical nuclear gradients for 1-electron spin-free x2c method  
Ref. JCP 135 084114

**pyscf.x2c.sfx2c1e_grad.hcore_grad_generator** *(x2obj, mol=None)*  
nuclear gradients of 1-component X2c hcore Hamiltonian (spin-free part only)

### 1e spin-free x2c hessian

Analytical nuclear hessian for 1-electron spin-free x2c method  
Ref. JCP 135 244104 JCTC 8 2617

**pyscf.x2c.sfx2c1e_hess.hcore_hess_generator** *(x2obj, mol=None)*  
nuclear gradients of 1-component X2c hcore Hamiltonian (spin-free part only)

### 1.34 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 closely related to the molecular implementation. The module names, function names, and layout 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 across the materials science and quantum chemistry communities, 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 module, we implement minor changes to the Gamma-point data structures. The associated classes
and methods are prefixed by “K”; for example, the mean-field k-point restricted Hartree-Fock and Kohn-Sham modules
are KRHF and KRKS. On top of these KSCF methods, one can find many correlation methods, such as k-point CCSD
and k-point EOM-CCSD methods. Other post-mean-field methods can be also be developed to explicitly enforce
translational symmetry through k-point sampling.

When using results of this code for publication, please cite the following papers:


The list of modules described in this chapter is:

1.34.1 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. Amongst 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 of 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.

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
`mesh` (a length-3 list or 1x3 array).

In the `Cell` class, many parameters are determined automatically according to the attribute `precision`, which
likewise can be set manually or left to its default value (1e-8). The parameters determined by `precision` include
`ke_cutoff`, `mesh`, `rcut`, `ew_cut` and `ew_eta`. These parameters can also be set manually by the user.

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 `mesh`
is None, the `Cell` initialization function will convert the `ke_cutoff` to the equivalent FFT-mesh according to the
relation $g = \frac{\sqrt{2E_{\text{cut}}}}{2\pi} a^T$ and will overwrite the `mesh` attribute.

Many PBC calculations are best performed using pseudopotentials, which are set via the `pseudo` attribute. Pseu-
dopotentials 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 $r_{\text{cut}}$, the Gaussian model charge $\text{ew}_{\text{eta}}$, and the energy cutoff $\text{ew}_{\text{cut}}$.

Besides the methods and parameters provided by Mole class (see Chapter \textit{gto — Molecular structure and GTO basis}), there are some parameters frequently used in the code to access the information of the crystal.

\textbf{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:

```python
# 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)
```

\textbf{Gv} The (N x 3) array of plane waves associated to mesh. mesh 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 $\text{mesh} = [nx,ny,nz]$, the number of vectors in Gv is $nx*ny*nz$.

\textbf{vol} Cell.vol gives the volume of the unit cell (in atomic unit).

\textbf{reciprocal_vectors} A 3x3 array. Each row is a reciprocal space primitive vector.

\textbf{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.

\textbf{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 \textit{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 \textit{Connection to Mole class}).

\textbf{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 $b1,b2,b3 = 2 \pi \text{inv}(a).T$
        mesh [(3,) list of ints] The number 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 \(0.5 * G^2 < \text{ke_cutoff}\) The default value is estimated based on precision.

dimension [int] Default is 3

** Following attributes (for experts) are automatically generated. **

ew_\eta, ew_\text{cut} [float] The Ewald ‘\eta’ and ‘cut’ parameters. See get_ewald_params()

(See other attributes in Mole)

Examples:

```python
>>> 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', 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 \(\sim \int g(r-0) g(r-R)\)

build (dump_input=True, parse_arg=True, a=None, mesh=None, ke_cutoff=None, precision=None, nimgs=None, ew_\eta=None, ew_\text{cut}=None, pseudo=None, basis=None, h=None, dimension=None, rcut=None, ecp=None, low_dim_ft_type=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:

- \text{a} [(3,3) \text{ndarray}] The real-space unit cell lattice vectors. Each row represents a lattice vector.
- \text{mesh} [(3) \text{ndarray of ints}] The number of \textit{positive} G-vectors along each direction.
- \text{pseudo} [\text{dict or str}] To define pseudopotential. If given, overwrite \text{Cell.pseudo}

.dumps (\text{cell})

Serialize Cell object to a JSON formatted str.

.energy_nuc (cell, ew_\eta=none, ew_\text{cut}=none)

Perform real (R) and reciprocal (G) space Ewald sum for the energy.

Formulation of Martin, App. F2.

Returns:

- \text{float} The Ewald energy consisting of overlap, self, and G-space sum.

See Also: pyscf.pbc.gto.get_ewald_params

eval_ao (eval_name, coords, comp=None, kpts=None, kpt=None, shls_slice=None, non0tab=None, ao_loc=None, out=None)

Evaluate PBC-AO function value on the given grids.

Args: eval_name : str
<table>
<thead>
<tr>
<th>Function</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>“GTOval_sph”</td>
<td>(\sum_T \exp(i\mathbf{k}\cdot\mathbf{T}) \mid \text{AO}\rangle)</td>
</tr>
<tr>
<td>“GTOval_ip_sph”</td>
<td>(\nabla \sum_T \exp(i\mathbf{k}\cdot\mathbf{T}) \mid \text{AO}\rangle)</td>
</tr>
<tr>
<td>“GTOval_cart”</td>
<td>(\sum_T \exp(i\mathbf{k}\cdot\mathbf{T}) \mid \text{AO}\rangle)</td>
</tr>
<tr>
<td>“GTOval_ip_cart”</td>
<td>(\nabla \sum_T \exp(i\mathbf{k}\cdot\mathbf{T}) \mid \text{AO}\rangle)</td>
</tr>
</tbody>
</table>

**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 cell 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 `dft.gen_grid.make_mask()`.

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

**Returns:** A list of 2D (or 3D) arrays to hold the AO values on grids. Each element of the list corresponds to a k-point and it has the shape \((N,\text{nao})\) or shape \((*,N,\text{nao})\).

**Examples:**

```python
cell = pbc.gto.M(a=numpy.eye(3)*4, atom='He 1 1 1', basis='6-31g')
coords = cell.get_uniform_grids([20,20,20])
kpts = cell.make_kpts([3,3,3])

ao_value = cell.pbc_eval_gto("GTOval_sph", coords, kpts)
len(ao_value)  # 27
ao_value[0].shape  # (100, 2)

ao_value = cell.pbc_eval_gto("GTOval_ip_sph", coords, kpts, comp=3)
print(ao_value.shape)
len(ao_value)  # 27
ao_value[0].shape  # (3, 100, 2)
```

**eval_gto** (eval_name, coords, comp=None, kpts=None, kpt=None, shls_slice=None, non0tab=None, ao_loc=None, out=None)
Evaluate PBC-AO function value on the given grids,

**Args:** eval_name : str

<table>
<thead>
<tr>
<th>Function</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>“GTOval_sph”</td>
<td>(\sum_T \exp(i\mathbf{k}\cdot\mathbf{T}) \mid \text{AO}\rangle)</td>
</tr>
<tr>
<td>“GTOval_ip_sph”</td>
<td>(\nabla \sum_T \exp(i\mathbf{k}\cdot\mathbf{T}) \mid \text{AO}\rangle)</td>
</tr>
<tr>
<td>“GTOval_cart”</td>
<td>(\sum_T \exp(i\mathbf{k}\cdot\mathbf{T}) \mid \text{AO}\rangle)</td>
</tr>
<tr>
<td>“GTOval_ip_cart”</td>
<td>(\nabla \sum_T \exp(i\mathbf{k}\cdot\mathbf{T}) \mid \text{AO}\rangle)</td>
</tr>
</tbody>
</table>

**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 cell 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 dft.gen_grid.make_mask()
- **out** [ndarray] If provided, results are written into this array.

**Returns:** A list of 2D (or 3D) arrays to hold the AO values on grids. Each element of the list corresponds to a k-point and it has the shape (N,nao) or shape (*,N,nao).

**Examples:**

```python
cell = pbc.gto.M(a=numpy.eye(3)*4, atom='He 1 1 1', basis='6-31g')
coords = cell.get_uniform_grids([20,20,20])
kpts = cell.make_kpts([3,3,3])
ao_value = cell.pbc_eval_gto("GTOval_sph", coords, kpts)
len(ao_value)
27
>>> ao_value[0].shape
(100, 2)
>>> ao_value = cell.pbc_eval_gto("GTOval_ig_sph", coords, kpts, comp=3)
>>> print(ao_value.shape)
27
>>> ao_value[0].shape
(3, 100, 2)
```

**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:

```python
{ 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:**

```python
>>> pbc.format_basis({'H': 'gth-szv'})
{'H': [[0,
(8.3744350009, -0.0283380461),
(1.8058681460, -0.1333810052),
... ]
```
**format_pseudo** *(pseudo_tab)*

Convert the input `Cell.pseudo (dict)` to the internal data format:

```python
{ 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[2,1], hproj1[2,2], ...),
    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:**

```python
>>> 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:**

```python
>>> from ase.lattice import bulk
>>> cell.from_ase(bulk('C', 'diamond', a=LATTICE_CONST))
```

**gen_uniform_grids** *(cell, mesh=None, **kwargs)*

Generate a uniform real-space grid consistent w/ samp thm; see MH (3.19).

**Args:** `cell : instance of `Cell`

**Returns:**

**get_Gv** *(cell, mesh=None, **kwargs)*

Calculate three-dimensional G-vectors for the cell; see MH (3.8).

Indices along each direction go as [0...N-1, -N...-1] to follow FFT convention.

**Args:** `cell : instance of `Cell`

**Returns:**
- `Gv` [[(ngrid, 3) ndarray of floats] The array of G-vectors.]
get_Gv_weights (cell, mesh=None, **kwargs)
Calculate G-vectors and weights.

Returns:

Gv [(ngris, 3) ndarray of floats] The array of G-vectors.

get_SI (cell, Gv=None)
Calculate the structure factor (0D, 1D, 2D, 3D) for all atoms; see MH (3.34).

Args:
cell : instance of Cell
Gv [(N,3) array] G vectors

Returns:

SI [(natm, ngrids) 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, mesh=None)
Choose a reasonable value of Ewald ‘eta’ and ‘cut’ parameters. \( \eta^2 \) is the exponent coefficient of the
model Gaussian charge for nucleus at \( R: \frac{1}{\pi^{1.5}} e^{-\eta^2 (r-R)^2} \)
Choice is based on largest G vector and desired relative precision.
The relative error in the G-space sum is given by

\[
\text{precision} \sim 4\pi G_{\text{max}}^2 e^{-G_{\text{max}}^2/(4 \eta^2)}
\]
which determines \( \eta \). Then, real-space cutoff is determined by (exp. factors only)

\[
\text{precision} \sim \text{erfc}(\eta \times \text{rcut}) / \text{rcut} \sim e^{(-\eta^2 \times \text{rcut}^2)}
\]

Returns:

ew_\eta, ew_\text{cut} [float] The Ewald ‘eta’ and ‘cut’ parameters.

get_kpts (cell, nks, wrap_around=False, with_gamma_point=True, scaled_center=None)
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.
**scaled_center** [(3,) array] Shift all points in the Monkhorst-pack grid to be centered on scaled_center, given as the zeroth index of the returned kpts. Scaled meaning that the k-points are scaled to a grid from [-1,1] x [-1,1] x [-1,1]

**Returns:** kpts in absolute value (unit 1/Bohr). Gamma point is placed at the first place in the k-points list

**Examples:**

```python
cell.make_kpts((4, 4, 4))
```

**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

\[
\text{precision} = \int r^l e^{-\alpha r^2} (r-rcut)^l e^{-\alpha (r-rcut)^2} = (rcut^2/(2\alpha))^l e^{\alpha/2 rcut^2}
\]

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

**get_uniform_grids** *(cell, mesh=None, **kwargs)*

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.

**has_ecp** ()

Whether pseudo potential is used in the system.

**kernel** *(dump_input=True, parse_arg=True, a=None, mesh=None, ke_cutoff=None, precision=None, nimgs=None, ew_etas=None, ew_cuts=None, pseudo=None, basis=None, h=None, dimension=None, rcut=None, ecp=None, low_dim_ft_type=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.

- **mesh** [(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, scaled_center=None)*

Given number of kpoints along x,y,z. generate kpoints
Args: nks : (3,) ndarray

Kwarg:

- **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.
- **scaled_center** [(3,) array] Shift all points in the Monkhorst-pack grid to be centered on scaled_center, given as the zeroth index of the returned kpts. Scaled meaning that the k-points are scaled to a grid from [-1,1] x [-1,1] x [-1,1]

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_eval_ao(eval_name, coords, comp=None, kpts=None, kpt=None, shls_slice=None, non0tab=None, ao_loc=None, out=None)
Evaluate PBC-AO function value on the given grids.

Args: eval_name : str

<table>
<thead>
<tr>
<th>Function</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>“GTOval_sph”</td>
<td>sum_T exp(ik*T)</td>
</tr>
<tr>
<td>“GTOval_ip_sph”</td>
<td>nabla sum_T exp(ik*T)</td>
</tr>
<tr>
<td>“GTOval_cart”</td>
<td>sum_T exp(ik*T)</td>
</tr>
<tr>
<td>“GTOval_ip_cart”</td>
<td>nabla sum_T exp(ik*T)</td>
</tr>
</tbody>
</table>

- **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.

Kwarg:

- **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 cell 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 dft.gen_grid.make_mask()
- **out** [ndarray] If provided, results are written into this array.

Returns: A list of 2D (or 3D) arrays to hold the AO values on grids. Each element of the list corresponds to a k-point and it has the shape (N,nao) Or shape (*,N,nao).

Examples:
>>> cell = pbc.gto.M(a=numpy.eye(3)*4, atom='He 1 1 1', basis='6-31g')
>>> coords = cell.get_uniform_grids([20,20,20])
>>> kpts = cell.make_kpts([3,3,3])
>>> ao_value = cell.pbc_eval_gto("GTOval_sph", coords, kpts)
>>> len(ao_value)
27
>>> ao_value[0].shape
```python
>>> ao_value = cell.pbc_eval_gto("GTOval_ig_sph", coords, kpts, comp=3)
>>> print(ao_value.shape)
27
>>> len(ao_value)
27
>>> ao_value[0].shape
(3, 100, 2)
```

### pbc_eval_gto(eval_name, coords, comp=None, kpts=None, kpt=None, shls_slice=None, non0tab=None, ao_loc=None, out=None)
Evaluate PBC-AO function value on the given grids,

**Args:**
- `eval_name` : str
  - Function Expression
  - "GTOval_sph" \( \sum_T \exp(ik*T) |AO> \)
  - "GTOval_ip_sph" \( \nabla \sum_T \exp(ik*T) |AO> \)
  - "GTOval_cart" \( \sum_T \exp(ik*T) |AO> \)
  - "GTOval_ip_cart" \( \nabla \sum_T \exp(ik*T) |AO> \)

- `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 cell 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 dft.gen_grid.make_mask()
- `out` [ndarray] If provided, results are written into this array.

**Returns:** A list of 2D (or 3D) arrays to hold the AO values on grids. Each element of the list corresponds to a k-point and it has the shape (N,nao) Or shape (*,N,nao).

**Examples:**
```python
>>> cell = pbc.gto.M(a=numpy.eye(3)*4, atom='He 1 1 1', basis='6-31g')
>>> coords = cell.get_uniform_grids([20,20,20])
>>> kpts = cell.make_kpts([3,3,3])
>>> ao_value = cell.pbc_eval_gto("GTOval_sph", coords, kpts)
>>> len(ao_value)
27
>>> ao_value[0].shape
(100, 2)
>>> ao_value = cell.pbc_eval_gto("GTOval_ig_sph", coords, kpts, comp=3)
>>> print(ao_value.shape)
27
>>> len(ao_value)
27
>>> ao_value[0].shape
(3, 100, 2)
```
**pbc_intor** *(inter, comp=None, hermi=0, kpts=None, kpt=None, shls_slice=None, **kwargs)*

One-electron integrals with PBC.

\[ \sum_i \int \frac{\mu(r) \ast [\text{intor}] \ast \nu(r - T)}{dr} \]

See also Mole.intor

**reciprocal_vectors** *(norm_to=6.283185307179586)*

- \[ b_1 = 2\pi \frac{a_2 \times a_3}{a_1 \cdot (a_2 \times a_3)} \]
- \[ b_2 = 2\pi \frac{a_3 \times a_1}{a_2 \cdot (a_3 \times a_1)} \]
- \[ b_3 = 2\pi \frac{a_1 \times a_2}{a_3 \cdot (a_1 \times a_2)} \]

**to_mol()**

Return a Mole object using the same atoms and basis functions as the Cell object.

**tot_electrons** *(cell, nkpts=1)*

Total number of electrons

**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.
**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 density 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.

**1.34.2 pbc.ao2mo — PBC integral transformation**

**Examples**

Relevant examples: examples/pbc/30-mo_integrals.py examples/pbc/41-pbc1d_real_space_sum.py

**Program reference**

eris

**1.34.3 pbc.scf — Mean-field with periodic boundary condition**

This module is constructed to be analogous to the molecular pyscf.scf module to handle mean-field calculations with periodic boundary conditions.

**Gamma point and single k-point calculation**

The usage of the Gamma point Hartree-Fock program is very close to that of the molecular program. In a PBC Gamma point calculation, one needs to initialize the Cell object and the corresponding pyscf.pbc.scf.hf.RHF class:

```python
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()
```

Compared to the pyscf.scf.hf.RHF object for molecular calculations, the PBC-HF calculation with pyscf.pbc.scf.hf.RHF or pyscf.pbc.scf.uhf.UHF has three differences
• pyscf.pbc.scf.hf.RHF is the single k-point PBC HF class. By default, it creates a Gamma point instance. You can change to other (single) k-points by setting the kpt attribute:

```python
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 show a slow convergence with respect to the number of k-points. A proper treatment for the divergent part of the exchange integral can improve the convergence. The attribute `exxdiv` is used to control the method to handle the divergent exchange term. The default `exxdiv='ewald'` can be used for most scenarios. However, if molecular post-HF methods are mixed with the Gamma point mean-field methods (see Mixing with molecular program for post-HF methods), you will need to explicitly use `exxdiv=None` to obtain a consistent total energy (see Exchange divergence treatment). If PBC post-HF methods are used, the `exxdiv` attribute is always treated as `None` in the correlated calculation to obtain a consistent total energy.

• In PBC calculations there are different choices for how to evaluate 2-electron integrals. The default integral scheme (FFTDF) is accurate for pseudo-potentials and fast for DFT calculations but slow in other scenarios. A second integral scheme which is a good balance of speed and accuracy for Hartree-Fock, all-electron, and post-Hartree-Fock calculations, is Gaussian density fitting. This can be used by setting the `with_df` attribute (see pbc.df — PBC density fitting) or, conveniently, by using the `density_fit()` function (see pbc_density_fit). For higher accuracy, you may wish to use mixed density fitting (MDF) (see with_df). Here is an example to update with_df

### Mixing with molecular program for post-HF methods

The Gamma point HF code adopts the same code structure, function and method names and argument’ conventions as the molecular SCF code. This design allows one to mix PBC HF objects with the existing molecular post-HF code for PBC electron correlation treatments. A typical molecular post-HF calculation starts from the finite-size HF method with the Mole object:

```python
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:

```python
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 mol or cell object to create and the scf module to import. With the system-specific mean-field object, one can carry 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**

An example demonstrating the use of k-points for a KRHF, KRKS calculation, and using the Newton (second-order SCF) solver.

**Smearing**

In many periodic systems, the HOMO-LUMO gap may become very small. In such a case, one needs to use occupation number smearing to converge the SCF calculation, as illustrated here:

```python
import numpy
from pyscf.pbc import gto, scf

cell = gto.Cell()
cell.atom = '''
He 0 0 1
He 1 0 1
'''
cell.basis = 'ccpvdz'
cell.a = numpy.eye(3) * 4
cell.verbose = 4
cell.build()

# Use scf.addons.smearing_ function to modify the PBC (gamma-point or k-points) SCF object
nks = [2,1,1]
mf = scf.KRHF(cell, cell.make_kpts(nks))
mf = scf.addons.smearing_(mf, sigma=.1, method='fermi')
mf.kernel()
```

**Exchange divergence treatment**

The attribute `exxdiv` controls the handling of the slow convergence of the HF exchange integrals. Generally, `exxdiv` leads to a shift in the total energy that can be thought of as a HF finite size-correction. It also modifies the spectrum of the occupied orbitals. When a Gamma-point PBC mean-field calculation is mixed with a molecular post-HF implementation, setting this attribute to anything other than `None` will lead to an inconsistency in the total energy.

Possible values are `None`, `vcut_sph` (spherical cutoff), `vcut_ws` (Wigner-Seitz cutoff), `ewald` (probe-charge Ewald correction). `vcut_sph` and `vcut_ws` are only available when using FFTDF. When using hybrid functionals, the exchange treatment will use the `exxdiv` attribute.

Note that when calling `get_bands()`, the choice of `exxdiv` affects the band structure. In particular, `vcut_sph` (spherical cutoff), `vcut_ws` (Wigner-Seitz cutoff) should be used to obtain smooth bands, otherwise, the band structure should be computed by twisting the SCF calculation (i.e. redoing self-consistency at each k point).

**Gaussian density fitting**

This example uses the `density_fit()` function to enable Gaussian density fitting:

```python
import numpy as np
from pyscf import gto as mol_gto
from pyscf.pbc import gto, scf, cc, df
```
cell = gto.Cell()
cell.atom=''
C 0.000000000000 0.000000000000 0.000000000000
C 1.685068664391 1.685068664391 1.685068664391
''
cell.basis = 'gth-dzv'
cell.pseudo = 'gth-pade'
cell.a = '''
0.000000000, 3.370137329, 3.370137329
3.370137329, 0.000000000, 3.370137329
3.370137329, 3.370137329, 0.000000000'''
cell.unit = 'B'
cell.verbose = 5
cell.build()
#
# Default DF auxiliary basis is a set of even-tempered gaussian basis (with
# exponents alpha * beta**i, i = 1,..,N). The even-tempered parameter alpha
# is determined automatically based on the orbital basis. beta is set to 2.0
#
mf = scf.RHF(cell).density_fit()
mf.kernel()

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
class pyscf.pbc.scf.hf.SCF (cell, kpt=array([ 0.,
0., 0.]), exxdiv='ewald')
SCF base 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
'ewald' : Ewald probe charge correction (JCP, 122, 234102)

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.
dip_moment (cell=None, dm=None, unit='Debye', verbose=3, **kwargs)
Dipole moment in the unit cell (is it well defined)?

Args:  cell : an instance of Cell
dm (ndarray) : density matrix

Return:  A list: the dipole moment on x, y and z components
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, omega=None)
Compute J matrix for the given density matrix and k-point (kpt). When kpts_band is given, the J matrices on kpts_band are evaluated.

J_{pq} = \sum_{rs} (pq|rs) \text{dm}[s,r]

where r,s are orbitals on kpt. p and q are orbitals on kpts_band if kpts_band is given otherwise p and q are orbitals on kpt.

get_jk (cell=None, dm=None, hermi=1, kpt=None, kpts_band=None, with_j=True, with_k=True, omega=None)
Get Coulomb (J) and exchange (K) following scf.hf.RHF.get_jk(). for particular k-point (kpt).
When kpts_band is given, the J, K matrices on kpts_band are evaluated.

J_{pq} = \sum_{rs} (pq|rs) \text{dm}[s,r] K_{pq} = \sum_{rs} (pr|sq) \text{dm}[r,s]

where r,s are orbitals on kpt. p and q are orbitals on kpts_band if kpts_band is given otherwise p and q are orbitals on kpt.

get_jk_incore (cell=None, dm=None, hermi=1, kpt=None, kpts_band=None, omega=None, **kwargs)
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, omega=None)
Compute K matrix for the given density matrix.

get_rho (mf, dm=None, grids=None, kpt=None)
Compute density in real space

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()

to_ghf (mf)
Convert the input mean-field object to a GHF/GKS object

to_rhf (mf)
Convert the input mean-field object to a RHF/ROHF/RKS/ROKS object

to_uhf (mf)
Convert the input mean-field object to a UHF/UKS object

PySCF Documentation, Release 1.7.0

1.34. pbc — Periodic boundary conditions
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 \( \psi_n(k) \)

```python
def get_hcore(cell, kpt=array([ 0., 0., 0.]))
```

Get the core Hamiltonian AO matrix.

```python
def 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

**Kwarg:**

- **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).

```python
def get_jk(mf, cell, dm, hermi=1, vhfopt=None, kpt=array([ 0., 0., 0.]), kpts_band=None, with_j=True, with_k=True, omega=None, **kwargs)
```

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

**Kwarg:**

- **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).

```python
def 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).

```python
def get_ovlp(cell, kpt=array([ 0., 0., 0.]))
```

Get the overlap AO matrix.

```python
def get_rho(mf, dm=None, grids=None, kpt=None)
```

Compute density in real space

```python
def get_t(cell, kpt=array([ 0., 0., 0.]))
```

Get the kinetic energy AO matrix.

```python
def init_guess_by_chkfile(cell, chkfile_name, project=None, kpt=None)
```

Read the HF results from checkpoint file, then project it to the basis defined by `cell`
Returns: Density matrix, (nao,nao) ndarray

pycsf.pbc.scf.hf.makov_payne_correction(mf)
Makov-Payne correction (Phys. Rev. B, 51, 4014)

pycsf.pbc.scf.hf.normalize_dm_(mf, dm)
Scale density matrix to make it produce the correct number of electrons.

Unrestricted Hartree-Fock for periodic systems at a single k-point

See Also: pyscf/pbc/scf/kfh.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.

convert_from_(mf)
Convert given mean-field object to RHF/ROHF

dip_moment (cell=None, dm=None, unit='Debye',verbose=3,**kwargs)
Dipole moment in the unit cell.

Args: cell : an instance of Cell
dm_kpts (a list of ndarrays) : density matrices of k-points

Return: A list: the dipole moment on x, y and z components

energy_tot (mf, dm=None, h1e=None, vhf=None)
Total Hartree-Fock energy, electronic part plus nuclear repulsion See scf.hf.energy_elec() for the electron part

Note this function has side effects which cause mf.scf_summary updated.

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, omega=None)
Compute J matrix for the given density matrix and k-point (kpt). When kpts_band is given, the J matrices on kpts_band are evaluated.

J_[pq] = sum_[rs] (pq|rs) dm[s,r]
where r,s are orbitals on kpt. p and q are orbitals on kpts_band if kpts_band is given otherwise p and q are orbitals on kpt.

get_jk (cell=None, dm=None, hermi=1, kpt=None, kpts_band=None, omega=None, **kwargs)
Get Coulomb (J) and exchange (K) following scf.hf.RHF.get_jk(). for particular k-point (kpt).

When kpts_band is given, the J, K matrices on kpts_band are evaluated.

J_[pq] = sum_[rs] (pq|rs) dm[s,r] K_[pq] = sum_[rs] (pr|sq) dm[r,s]
where r,s are orbitals on kpt. p and q are orbitals on kpts_band if kpts_band is given otherwise p and q are orbitals on kpt.

get_jk_incore (cell=None, dm=None, hermi=1, kpt=None, omega=None, **kwargs)
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, omega=None)
Compute K matrix for the given density matrix.

get_rho (mf, dm=None, grids=None, kpt=None)
Compute density in real space

make_rdm1 (mo_coeff=None, mo_occ=None, **kwargs)
One-particle density matrix
Returns: A list of 2D ndarrays for alpha and beta spins

stability (internal=True, external=False, verbose=None)
Stability analysis for RHF/RKS method.
See also pyscf.scf.stability.uhf_stability function.
Args: mf : UHF or UKS object
Kwargs:
    internal [bool] Internal stability, within the UHF space.
Returns: New orbitals that are more close to the stable condition. The return value includes two set of orbitals. The first corresponds to the internal stability and the second corresponds to the external stability.

pyscf.pbc.scf.uhf.dip_moment (cell, dm, unit='Debye', verbose=3, grids=None, rho=None, kpt=array([ 0., 0., 0.]))
Dipole moment in the unit cell.
Args: cell : an instance of Cell
    dm_kpts (a list of ndarrays) : density matrices of k-points
Return: A list: the dipole moment on x, y and z components

pyscf.pbc.scf.uhf.init_guess_by_chkfile (cell, chkfile_name, project=None, kpt=None)
Read the HF results from checkpoint file and make the density matrix for UHF initial guess.
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

class pyscf.pbc.scf.khf.KSCF (cell, kpts=array([[ 0., 0., 0.]]), exxdiv='ewald')
SCF base 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 [(nkpts,3) ndarray] The sampling k-points in Cartesian coordinates, in units of 1/Bohr.
    dip_moment (cell=None, dm=None, unit='Debye', verbose=3, **kwargs)
    Dipole moment in the unit cell (is it well defined)?
    Args: cell : an instance of Cell
        dm_kpts (a list of ndarrays) : density matrices of k-points
    Return: A list: the dipole moment on x, y and z components

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_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

**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_rho** *(mf, dm=None, grids=None, kpts=None)*
Compute density in real space

**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()

**to_ghf** *(mf)*
Convert the input mean-field object to a KGHF/KGKS object

**to_rhf** *(mf)*
Convert the input mean-field object to a KRHF/KROHF/KRKS/KROKS object

**to_uhf** *(mf)*
Convert the input mean-field object to a KUHF/KUKS object

pyscf.pbc.scf.khf.**analyze** *(mf, verbose=5, with_meta_lowdin=True, **kwargs)*
Analyze the given SCF object: print orbital energies, occupancies; print orbital coefficients; Mulliken population analysis; Dipole moment

pyscf.pbc.scf.khf.**dip_moment** *(cell, dm_kpts, unit='Debye', verbose=3, grids=None, rho=None, kpts=None)*
Dipole moment in the unit cell (is it well defined)?

**Args:**
- **cell** : an instance of Cell
- **dm_kpts** (a list of ndarrays) : density matrices of k-points

**Return:** A list: the dipole moment on x, y and z components
Following `pyscf.scf.hf.energy_elec()`

```
pyscf.pbc.scf.khf.energy_elec(mf, dm_kpts=None, h1e_kpts=None, vhf_kpts=None)
```

Fermi level

```
pyscf.pbc.scf.khf.get_fermi(mf, mo_energy_kpts=None, mo_occ_kpts=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

```
pyscf.pbc.scf.khf.get_grad(mo_coeff_kpts, mo_occ_kpts, fock)
```

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_hcore(mf, cell=None, kpts=None)
```

Get the core Hamiltonian AO matrices at sampled k-points.

```
Args:

Returns:
```

```
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. It needs to be Hermitian.

Kwargs:

kpts_band [(k,3) ndarray] A list of arbitrary “band” k-points at which to evaluate 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, with_j=True, with_k=True, omega=None, **kwargs)
```

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. It needs to be Hermitian.

Kwargs:

kpts_band [(3,) ndarray] A list of arbitrary “band” k-point at which to evaluate the matrix.

Returns: vj : (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.get_rho(mf, dm=None, grids=None, kpts=None)
```

Compute density in real space

```
pyscf.pbc.scf.khf.init_guess_by_chkfile(cell, chkfile_name, project=None, 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, **kwargs)

One particle density matrices for all k-points.

Returns: dm_kpts : (nkpts, nao, nao) ndarray

pyscf.pbc.scf.khf.mulliken_meta(cell, dm_ao_kpts, verbose=5, pre_orth_method='ANO', s=None)

A modified 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 separately (without change occupancy).

convert_from_(mf)

Convert given mean-field object to KUHF

dip_moment(cell=None, dm=None, unit='Debye', verbose=3, **kwargs)

Dipole moment in the unit cell.

Args: cell : an instance of Cell
dm_kpts (two lists of ndarrays) : KUHF density matrices of k-points

Return: A list: the dipole moment on x, y and z components

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)

A pair of Fermi level for spin-up and spin-down orbitals

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

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_rho** *(mf, dm=None, grids=None, kpts=None)*

Compute density in real space

**mulliken_meta** *(cell=None, dm=None, verbose=5, pre_orth_method='ANO', s=None)*

A modified Mulliken population analysis, based on meta-Lowdin AOs.

Note this function only computes the Mulliken population for the gamma point density matrix.

**spin_square** *(mo_coeff=None, s=None)*

Spin square and multiplicity of UHF determinant

\[ S^2 = \frac{1}{2}(S_+S_- + S_-S_+) + S_z^2 \]

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 \( S_+S_- = \sum_i s_{i+}s_{i-} \),
   
   \[ \sum_i \langle UHF|s_{i+}s_{i-}|UHF\rangle = \sum_{pq} \langle p|s_{i+}s_{i-}|q\rangle \gamma_{qp} = n_\alpha \]

   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,

   \[ \langle S_+S_- \rangle = \langle ij|s_{i+}s_{j-}|ij \rangle - \langle ij|s_{i+}s_{j-}|ji \rangle = -(i^\alpha|j^\beta\rangle\langle j^\alpha|i^\beta \rangle) \]

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

   \[
   \frac{1}{2} \sum_{ij} \left( \langle ij|2s_{z1}s_{z2}|ij \rangle - \langle ij|2s_{z1}s_{z2}|ji \rangle \right) \\
   = \frac{1}{4} \left( (i^\alpha|j^\alpha\rangle\langle j^\alpha|i^\alpha \rangle - (i^\alpha|j^\alpha\rangle\langle j^\beta|i^\beta \rangle - (i^\beta|j^\alpha\rangle\langle j^\alpha|i^\beta \rangle + (i^\beta|j^\beta\rangle\langle j^\beta|i^\beta \rangle) \\
   - \frac{1}{4} \left( (i^\alpha|j^\alpha\rangle\langle j^\alpha|i^\alpha \rangle + (i^\beta|j^\beta\rangle\langle j^\beta|i^\beta \rangle \\
   = \frac{1}{4}(n_\alpha^2 - n_\alpha n_\beta - n_\beta n_\alpha + n_\beta^2) - \frac{1}{4}(n_\alpha + n_\beta) \\
   = \frac{1}{4}(n_\alpha - n_\beta)^2 - (n_\alpha + n_\beta) \right) \]

342 Chapter 1. Contents
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:**
```python
>>> 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**

Canonalization diagonalizes the UHF Fock matrix within occupied, virtual subspaces separately (without change occupancy).

**pyscf.pbc.scf.kuhf.dip_moment**

Dipole moment in the unit cell.

**Args:**
- `cell` : an instance of Cell
- `dm_kpts` (two lists of ndarrays) : KUHF density matrices of k-points

**Returns:** A list: the dipole moment on x, y and z components

**pyscf.pbc.scf.kuhf.energy_elec**

Following pyscf.scf.hf.energy_elec()

**pyscf.pbc.scf.kuhf.get_fermi**

A pair of Fermi level for spin-up and spin-down orbitals

**pyscf.pbc.scf.kuhf.get_occ**

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**

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**

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)

A modified Mulliken population analysis, based on meta-Lowdin AOs.

Note this function only computes the Mulliken population for the gamma point density matrix.

1.34.4 *pbc.dft — PBC density functional theory*

**Examples**

Relevant examples:
```
examples/pbc/10-gamma_point_scf.py examples/pbc/11-gamma_point_all_electron_scf.py
examples/pbc/20-k_points_scf.py examples/pbc/21-k_points_all_electron_scf.py
```

**Program reference**

rks
roks
uks
cdft
krks
ekroks
kuks
numint
gen_grid
multigrid

1.34.5 *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.
### FFTDF — FFT-based density fitting

FFTDF represents the method to compute electron repulsion integrals in reciprocal space with the Fourier transformed Coulomb kernel

\[
(ij|kl) = \sum_G \rho_{ij}(G) \frac{4\pi}{G^2} \rho_{kl}(-G)
\]

\(G\) is the plane wave vector. \(\rho_{ij}(G)\) is the Fourier transformed orbital pair

\[
\rho_{ij}(G) = \sum_r e^{-G\cdot r} \psi_i(r) \psi_j(r)
\]

Here are some examples to initialize FFTDF object:

```python
>>> 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:

```python
>>> vnucc = fftdf.get_pp()
>>> print(vnucc.shape)
(2, 2)
>>> kpts = cell.make_kpts([2,2,2])
>>> vnucc = fftdf.get_pp(kpts)
>>> print(vnucc.shape)
(8, 2, 2)
>>> vnucc = fftdf.get_pp(kpts)
```
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:

```python
>>> 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:

```python
>>> 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

\[(k_j - k_i + k_l - k_k) \cdot a = 2n\pi.\]

By default, four \(\Gamma\)-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:

```python
>>> 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:
```python
>>> 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.
Recommended 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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):

```python
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:

```python
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:

```python
>>> 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, \(\text{ke\_cutoff}=10\) 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:

```python
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()
# 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:

```python
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:

```python
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:

```python
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

```python
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

\[ (ij|kl) = \int dr_1 dr_2 i^*(r_1) j(r_1) v(r_12) k^*(r_2) l(r_2) = (ij|G) v(G) (G|kl) \]

\[ i^*(r) j(r) = \frac{1}{N} \sum_G e^{iGr} (G|ij) \]

\[ = \frac{1}{N} \sum_G e^{-iGr} (ij|G) \]

“forward” FFT:

\[ (Gijkl) = \sum_r e^{-iGr} i^*(r) j(r) = \text{fft}[ i^*(r) j(r) ] \]

“inverse” FFT:

\[ (ij|G) = \sum_r e^{iGr} i^*(r) j(r) = N * \text{ifft}[ i^*(r) j(r) ] = \text{conj}[ \sum_r e^{iGr} i^*(r) j(r) ] \]

AFTDF class

```python
class pyscf.pbc.df.aft.AFTDF (cell, kpts=array([[ 0., 0., 0.]]))
    Density expansion on plane waves
```

1.34. pbc — Periodic boundary conditions
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: J. Chem. Phys. 147, 164119 (2017)

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: J. Chem. Phys. 147, 164119 (2017)

1.34.6 pbc.tdscf — PBC time dependent density functional theory

Examples

Relevant examples examples/pbc/22-k_points_tddft.py
The module \texttt{pbc.cc} carries out PBC coupled cluster calculation, with optional usage of k-point symmetry.
Examples

Relevant examples examples/pbc/12-gamma_point_post_hf.py examples/pbc/22-k_points_ccsd.py examples/pbc/24-k_points_vs_gamma.py examples/pbc/25-k_points_mpi_ccsd.py examples/pbc/29-eom_ccsd_Ta.py examples/pbc/36-ccsd_level_shift.py

Program reference

ccsd
kccsd
kccsd_rhf
kccsd_uhf
kccsd_t
kccsd_t_rhf
kccsd_t_rhf_slow
kintermediates
kintermediates_rhf
kintermediates_uhf
eom_kccsd_ghf
eom_kccsd_rhf
eom_kccsd_rhf_ip
eom_kccsd_rhf_ea
eom_kccsd_uhf

1.34.10 pbc.mpicc — PBC coupled cluster for MPI

Examples

Relevant examples examples/pbc/25-k_points_mpi_ccsd.py
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

**1.34.12 Mixing PBC and molecular modules**

Post-HF methods, as standalone numerical solvers, do not require knowledge of the boundary conditions. Calculations on finite-sized systems and extended systems are distinguished by the boundary conditions of the integrals (and basis). The same post-HF solver can thus be used for both finite-size problems and the periodic boundary problems if they have a compatible Hamiltonian structure.

In PySCF, many molecular post-HF solvers have two implementations: an incore and outcore version. These differ by the treatment of the 2-electron integrals. The incore solver takes the \texttt{\_eri} (or \texttt{\_with\_df}, see \texttt{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 conditions) on the fly. To use the molecular post-HF solvers in PBC code, we need to ensure that the incore version solver is called.

Generating \texttt{\_eri} in a mean-field object is the straightforward way to trigger the incore post-HF solver. If the allowed memory is large 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 \texttt{incore\_anyway} in \texttt{cell} which forces the program to generate and hold \texttt{\_eri} in the mean-field object.

\textbf{Note:} If the problem is big, \texttt{incore\_anyway} may overflow the available physical memory.

Holding the full integral array \texttt{\_eri} in memory limits the problem size one can treat. Using the density fitting object \texttt{\_with\_df} to hold the integrals can overcome this problem. This architecture has been bound to PBC and molecular mean-field modules. Not all post-HF methods are available with density fitting.

Aside from the 2-electron integrals, there are some attributes and methods required by the post-HF solver. They are \texttt{\_get\_hcore()}, \texttt{\_get\_ovlp()} for 1-electron integrals, \texttt{\_numint}, \texttt{\_grids} for the numerical integration of DFT exchange-correlation functionals. These are all overloaded in the PBC mean-field object to produce the PBC integrals.
Examples

1.34.13 pbc.prop — PBC property

Electric field gradients

Electric field gradients, nuclear quadrupolar coupling and Mossbauer spectroscopy for non-relativistic (or sf-x2c) mean-field and post-HF methods. See also pyscf/prop/efg/rhf.py

Ref:

1.34.14 pbc.gw — GW with PBC

Program reference

G0W0

This module implements the G0W0 approximation on top of pyscf.tdscf.rhf_slow TDHF implementation. Unlike gw.py, all integrals are stored in memory. Several variants of GW are available:

- pyscf.gw_slow: the molecular implementation;
- pyscf.pbc.gw.gw_slow: single-kpoint PBC (periodic boundary condition) implementation;
- pyscf.pbc.gw.kgw_slow_supercell: a supercell approach to PBC implementation with multiple k-points. Runs the molecular code for a model with several k-points for the cost of discarding momentum conservation and using dense instead of sparse matrices;
- (this module) pyscf.pbc.gw.kgw_slow: a PBC implementation with multiple k-points;

1.35 qmmm — QM/MM interface

To run HF with background charges, one could follow the following example:

```python
import numpyrom pyscf import gto, scf, qmmm
mol = gto.M(atom='''
  C 1.1879 -0.3829 0.0000
  C 0.0000 0.5526 0.0000
  O -1.1867 -0.2472 0.0000
  H -1.9237 0.3850 0.0000
  H 2.0985 0.2306 0.0000
  H 1.1184 -1.0093 0.8869
  H 1.1184 -1.0093 -0.8869
  H -0.0227 1.1812 0.8852
  H -0.0227 1.1812 -0.8852
''',
basis='3-21g',
verbose=4)
numpy.random.seed(1)
coords = numpy.random.random((5,3)) * 10
```
charges = (numpy.arange(5) + 1.) * -.1
mf = scf.UHF(mol)
mf = qmmm.mm_charge(mf, coords, charges)
mf.run()

1.35.1 Examples

Relevant examples:

- examples/qmmm/00-hf.py
- examples/qmmm/01-dft.py
- examples/qmmm/02-mcscf.py
- examples/qmmm/03-ccsd.py
- examples/qmmm/04-cisd.py
- examples/qmmm/05-mp2.py
- examples/qmmm/06-tddft.py
- examples/qmmm/10-x2c.py
- examples/qmmm/11-newton.py
- examples/qmmm/11-soscf.py
- examples/qmmm/20-grad.py
- examples/qmmm/21-geom_opt.py

QM part interface

```python
pyscf.qmmm.itrf.add_mm_charges(scf_method, coords, charges, unit=None)
```

Modify the QM method using the (non-relativistic) potential generated by MM charges. Note the static Coulomb interactions of the background charges are not included in the total energy.

**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 (.x2c method), eg `mf = mm_charge(scf.RHF(mol).x2c()), [(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:**

```python
>>> 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
```

```python
pyscf.qmmm.itrf.mm_charge(scf_method, coords, charges, unit=None)
```

Modify the QM method using the (non-relativistic) potential generated by MM charges. Note the static Coulomb interactions of the background charges are not included in the total energy.

**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 (.x2c method), eg `mf = mm_charge(scf.RHF(mol).x2c()), [(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:**

```python
>>> 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
```
Examples:

```python
>>> 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:

```python
>>> from pyscf import gto, scf, grad
```

```python
>>> 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
```

```python
>>> hfg = mm_charge_grad(grad.hf.RHF(mf), coords, charges)
>>> hfg.kernel()
```

```python
[[-0.25912357 -0.29235976 -0.38245077]
 [-1.70497052 -1.89423883 1.2794798 ]]
```

The qmmm module implements a generic interface for use with MM programs.

### 1.36 solvent — Solvent methods

The solvent implements the domain-decomposition COSMO solvent model for mean-field and correlated methods.

To run a mean-field calculation with the implicit solvent model, one can try the following example:

```python
from pyscf import gto, scf, dft
from pyscf import solvent
mol = gto.M(atom='''
C 0.000000 0.000000 -0.542500
O 0.000000 0.000000 0.677500
H 0.000000 0.9353074360871938 -1.082500
H 0.000000 -0.9353074360871938 -1.082500
''', verbose = 4)
mf = scf.RHF(mol)
solvent.ddCOSMO(mf).run()```
1.36.1 Examples

Relevant examples: examples/solvent/00-scf_with_ddcosmo.py, examples/solvent/01-casscf_with_ddcosmo.py, examples/solvent/02-casci_with_ddcosmo.py, examples/solvent/03-ccsd_with_ddcosmo.py, examples/solvent/20-state_specific_casci.py

1.36.2 Program reference

domain decomposition PCM

domain decomposition PCM (In testing)

See also JCP, 144, 054101 JCP, 144, 160901

```
pyscf.solvent.ddpcm.energy(pcmobj, dm)
```

ddPCM energy $E_s = \frac{1}{2} \int f(\varepsilon) \rho(r) W(r) \, dr$

domain decomposition COSMO

domain decomposition COSMO

See also the code on github

https://github.com/filippolipparini/ddPCM

and the papers


pyscf.solvent.ddcosmo.ddcosmo_for_casci (mc, solvent_obj=None, dm=None)
Patch ddCOSMO to CASCI method.

Kwargs:

- **dm**: [if given, solvent does not response to the change of density] matrix. A frozen ddCOSMO potential is added to the results.

pyscf.solvent.ddcosmo.ddcosmo_for_casscf (mc, solvent_obj=None, dm=None)
Patch ddCOSMO to CASSCF method.

Kwargs:

- **dm**: [if given, solvent does not response to the change of density] matrix. A frozen ddCOSMO potential is added to the results.

pyscf.solvent.ddcosmo.ddcosmo_for_post_scf (method, solvent_obj=None, dm=None)
Default wrapper to patch ddCOSMO to post-SCF methods (CC, CI, MP, TDDFT etc.)

NOTE: this implementation often causes (macro iteration) convergence issue

Kwargs:

- **dm**: [if given, solvent does not response to the change of density] matrix. A frozen ddCOSMO potential is added to the results.

pyscf.solvent.ddcosmo.ddcosmo_for_scf (mf, solvent_obj=None, dm=None)
Patch ddCOSMO to SCF (HF and DFT) method.

Kwargs:

- **dm**: [if given, solvent does not response to the change of density] matrix. A frozen ddCOSMO potential is added to the results.

pyscf.solvent.ddcosmo.energy (pcmobj, dm)
ddCOSMO energy $E_s = 1/2 \int \rho(r) W(r) \, dr$

pyscf.solvent.ddcosmo.gen_ddcosmo_solver (pcmobj, verbose=None)
Generate ddcosmo function to compute energy and potential matrix
Analytical nuclear gradients for domain decomposition COSMO

See also


1.37 semiempirical — Semiempirical methods

The semiempirical module implements several semi-empirical Hamiltonians, including MINDO and AM1.

1.37.1 Program reference

MINDO3

MINDO/3


class pyscf.semiempirical.mindo3.RMINDO3 (mol)
RHF-MINDO/3 for closed-shell systems

def get_jk (mol=None, dm=None, hermi=1, with_j=True, with_k=True)
Compute J, K matrices for all input density matrices

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: not hermitian and not symmetric
  - 1: hermitian or symmetric
  - 2: anti-hermitian
- vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices
- with_j [boolean] Whether to compute J matrices
- with_k [boolean] Whether to compute K matrices
- omega [float] Parameter of range-separated Coulomb operator: erf(omega * r12) / r12. If specified, integration are evaluated based on the long-range part of the range-separated Coulomb operator.

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:
```python
>>> from pyscf import gto, scf
>>> 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.semiempirical.mindo3.UMINDO3(mol)

UHF-MINDO/3 for open-shell systems

```python
def get_jk(mol=None, dm=None, hermi=1, with_j=True, with_k=True):
    Compute J, K matrices for all input density matrices

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 : not hermitian and not symmetric
    1 : hermitian or symmetric
    2 : anti-hermitian
vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices
with_j [boolean] Whether to compute J matrices
with_k [boolean] Whether to compute K matrices
omega [float] Parameter of range-separated Coulomb operator: erf(omega * r12) / r12. If specified, integration are evaluated based on the long-range part of the range-separated Coulomb operator.

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:
```python
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> j, k = scf.hf.get_jk(mol, dms, hermi=0)
>>> print(j.shape)
(3, 2, 2)
```

pyscf.semiempirical.mindo3.get_hcore(mol)

Core Hamiltonian

Examples:
```python
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> scf.hf.get_hcore(mol)
array([[ 0.23655557, 0.75723433],
       [ 0.75723433, 0.23655557]])
```
pyscf.semiempirical.mindo3.get_init_guess(mol)
Average occupation density matrix

pyscf.semiempirical.mindo3.get_jk(mol, dm)
Compute J, K matrices for all input density matrices

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 : not hermitian and not symmetric
1 : hermitian or symmetric
2 : anti-hermitian

vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices

with_j [boolean] Whether to compute J matrices
with_k [boolean] Whether to compute K matrices
omega [float] Parameter of range-separated Coulomb operator: erf( omega * r12 ) / r12. If specified,
integration are evaluated based on the long-range part of the range-separated Coulomb operator.

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:

```python
>>> 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)
```

MINDO3 Gradient

AM1

1.38 geomopt — Geometry optimization

The geomopt module implements geometry optimization via interfaces to geomeTRIC (https://github.com/leeping/geomeTRIC) and PyBerny (https://github.com/azag0/pyberny). For example, optimize the structure of N2 molecule using PyBerny:

```python
from pyscf import gto, scf
from pyscf.geomopt.berny_solver import optimize
mol = gto.M(atom='N 0 0 0; N 0 0 1.2', basis='ccpvdz')
mf = scf.RHF(mol)
mol_eq = optimize(mf)
print(mol_eq.atom_coords())
```
1.38.1 Examples

Relevant examples

- examples/geomopt/01-geomeTRIC.py
- examples/geomopt/01-pyberny.py
- examples/geomopt/02-as_pyscf_method.py
- examples/geomopt/10-with_qmmm.py
- examples/geomopt/11-with_ghost_atom.py
- examples/geomopt/12-excited_states.py
- examples/geomopt/12-mcscf_excited_states.py
- examples/geomopt/13-ccsd_t.py

1.38.2 Program reference

**pyberny**

Interface to geometry optimizer pyberny https://github.com/azag0/pyberny

```python
class pyscf.geomopt.berny_solver.GeometryOptimizer(method):
    Optimize the molecular geometry for the input method.
    Note the method.mol will be changed after calling .kernel() method.

pyscf.geomopt.berny_solver.kernel(method, assert_convergence=True, include_ghost=True, callback=None, **kwargs)
    Optimize geometry with pyberny for the given method.
    To adjust the convergence threshold, parameters can be set in kwargs as below:

pyscf.geomopt.berny_solver.optimize(method, assert_convergence=True, include_ghost=True, callback=None, **kwargs)
    Optimize geometry with pyberny for the given method.
    To adjust the convergence threshold, parameters can be set in kwargs as below:

pyscf.geomopt.berny_solver.to_berny_log(pyscf_log)
    Adapter to allow pyberny to use pyscf.logger
```

**geomeTRIC**

**addons**

Helper functions for geometry optimizer

```python
pyscf.geomopt.addons.as_pyscf_method(mol, scan_function)
    Create an wrapper for the given scan_function, to make it work as a pyscf gradients scanner. The wrapper can be
    passed to optimize().

    Args: scan_function : [mol] => (e_tot, grad)

    Examples::
        >>> mol = gto.M(atom='H; H 1 1.2', basis='ccpvdz')
        >>> scan_fn = scf.RHF(mol).nuc_grad_method().as_scanner()
        >>> m = as_pyscf_method(mol, scan_fn)
        >>> pyscf.geomopt.berny_solver.kernel(m)

pyscf.geomopt.addons.dump_mol_geometry(mol, new_coords, log=None)
    Dump the molecular geometry (new_coords) and the displacement wrt old geometry.

    Args: new_coords (ndarray) : Cartesian coordinates in Angstrom

pyscf.geomopt.addons.symmetrize(mol, coords)
    Symmetrize the structure of a molecule.
```
1.39 lo — Orbital localization and analysis tools

The lo module implements various orbital localizations, such as intrinsic atomic orbitals and natural atomic orbitals. For example, to obtain the natural atomic orbital coefficients (in terms of the original atomic orbitals):

```python
import numpy
from pyscf import gto, scf, lo
x = .63
mol = gto.M(atom=[['C', (0, 0, 0)],
                   ['H', (x, x, x)],
                   ['H', (-x, -x, x)],
                   ['H', (-x, x, -x)],
                   ['H', (x, -x, -x)],
                   ],
           basis='ccpvtz')
mf = scf.RHF(mol).run()

# C matrix stores the AO to localized orbital coefficients
C = lo.orth_ao(mf, 'nao')
```

### 1.39.1 Examples

Relevant examples

- examples/local_orb/01-pop_with_meta_lowdin.py
- examples/local_orb/01-pop_with_nao.py
- examples/local_orb/02-pop_with_iao.py
- examples/local_orb/03-split_localization.py
- examples/local_orb/04-ibo_benzene_cubegen.py
- examples/local_orb/05-ibo_periodic_diamond_cubegen.py
- examples/local_orb/10-modify_valence.py
- examples/local_orb/40-hubbard_model_PM_localization.py
- examples/local_orb/nlocal.py
- examples/local_orb/pmloc.py
- examples/local_orb/ulocal.py

### 1.39.2 Program reference

#### 1.39.3 Foster-Boys, Edmiston-Ruedenberg, Pipek-Mezey localization

Foster-Boys localization  Edmiston-Ruedenberg localization  Pipek-Mezey localization

```python
pyscf.lo.pipek.atomic_pops(mol, mo_coeff, method='meta_lowdin')
```

**Kwargs:**

- **method** [string] one of mulliken, lowdin, meta_lowdin

**Returns:** A 3-index tensor $[A,i,j]$ indicates the population of any orbital-pair density $|i><j|$ for each species (atom in this case). This tensor is used to construct the population and gradients etc. You can customize the PM localization wrt other population metric, such as the charge of a site, the charge of a fragment (a group of atoms) by overwriting this tensor. See also the example pyscf/examples/loc_orb/40-hubbard_model_PM_localization.py for the PM localization of site-based population for hubbard model.

#### 1.39.4 Meta-Lowdin

```python
pyscf.lo.orth.lowdin(s)
```

new basis is $|\mu> c^\dagger{\text{lowdin}}_{\text{mu i}}$
Orthogonalize AOs

Kwargs:

- **method** [str] One of | lowdin : Symmetric orthogonalization | meta-lowdin : Lowdin orth within core, valence, virtual space separately (JCTC, 10, 3784) | NAO

Restore AO characters. Possible methods include the ANO/MINAO projection or fraction-averaged atomic RHF calculation

Projected AO = |bas><bas|ANO

New basis is |mu> c_{mu i}, c = w[(ws)^{-1/2}]

### 1.39.5 Natural atomic orbitals

Natural atomic orbitals Ref:


Change the default atomic core and valence configuration to the one given by “description”. See data/elements.py for the default configuration.

- **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

### 1.39.6 Intrinsic Atomic Orbitals

Intrinsic Atomic Orbitals ref. JCTC, 9, 4834

Intrinsic Atomic Orbitals application

Fast IAO Mullikan Pop: (mol, dm, iaos, verbose=5)
Args: mol: the molecule or cell object
   iaos [2D array] (orthogonal or non-orthogonal) IAO orbitals

Returns: mullikan population analysis in the basis IAO

pyscf.lo.iao.iao(mol, orbocc, minao='minao', kpts=None)
Intrinsic Atomic Orbitals. [Ref. JCTC, 9, 4834]
Args: mol: the molecule or cell object
   orbocc [2D array] occupied orbitals

Returns: non-orthogonal 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, orbocc)
>>> numpy.dot(c, orth.lowdin(reduce(numpy.dot, (c.T,s,c))))

pyscf.lo.iao.reference_mol(mol, minao='minao')
Create a molecule which uses reference minimal basis

1.39.7 Intrinsic Bond Orbitals

Intrinsic Bonding Orbitals ref. JCTC, 9, 4834

Below here is work done by Paul Robinson. much of the code below is adapted from code published freely on the
website of Gerald Knizia Ref: JCTC, 2013, 9, 4834-4843

pyscf.lo.ibo.MakeAtomIbOffsets(Atoms)
calculate offset of first orbital of individual atoms in the valence minimal basis (IB)

pyscf.lo.ibo.PM(mol, orbocc, iaos=None, s=None, exponent=4)
Note this localization is slightly different to Knizia’s implementation. The localization here reserves orthogonormality during optimization. Orbitals are projected to IAO basis first and the Mulliken pop is calculated based on IAO basis (in function atomic_pops). A series of unitary matrices are generated and applied on the input orbitals. The intermediate orbitals in the optimization and the finally localized orbitals are all orthogonormal.

Examples:

```python
>>> from pyscf import gto, scf
>>> from pyscf.lo import ibo
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1', basis='unc-sto3g')
>>> mf = scf.RHF(mol).run()
>>> pm = ibo.PM(mol, mf.mo_coeff[:,mf.mo_occ>0])
>>> loc_orb = pm.kernel()
```

pyscf.lo.ibo.Pipek(mol, orbocc, iaos=None, s=None, exponent=4)
Note this localization is slightly different to Knizia’s implementation. The localization here reserves orthogonormality during optimization. Orbitals are projected to IAO basis first and the Mulliken pop is calculated based on IAO basis (in function atomic_pops). A series of unitary matrices are generated and applied on the input orbitals. The intermediate orbitals in the optimization and the finally localized orbitals are all orthogonormal.

Examples:

```python
>>> from pyscf import gto, scf
>>> from pyscf.lo import ibo
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1', basis='unc-sto3g')
>>> mf = scf.RHF(mol).run()
```
pm = ibo.PM(mol, mf.mo_coeff[:,mf.mo_occ>0])
loc_orb = pm.kernel()

**PySCF Documentation, Release 1.7.0**

```python
>>> pm = ibo.PM(mol, mf.mo_coeff[:,mf.mo_occ>0])
>>> loc_orb = pm.kernel()
```

**pyscf.lo.ibo.PipekMezey** *(mol, orbocc, iaos=None, s=None, exponent=4)*

Note this localization is slightly different to Knizia’s implementation. The localization here reserves orthogonormality during optimization. Orbitals are projected to IAO basis first and the Mulliken pop is calculated based on IAO basis (in function atomic_pops). A series of unitary matrices are generated and applied on the input orbitals. The intermediate orbitals in the optimization and the finally localized orbitals are all orthogonormal.

Examples:

```python
>>> from pyscf import gto, scf
>>> from pyscf.lo import ibo
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1',
              basis='unc-sto3g')
>>> mf = scf.RHF(mol).run()
>>> pm = ibo.PM(mol, mf.mo_coeff[:,mf.mo_occ>0])
>>> loc_orb = pm.kernel()
```

**pyscf.lo.ibo.ibo** *(mol, orbocc, iaos=None, exponent=4, grad_tol=1e-08, max_iter=200, verbose=3)*

Intrinsic Bonding Orbitals. [Ref. JCTC, 9, 4834]

This implementation follows Knizia’s implementation except that the resultant IBOs are symmetrically orthogonalized. Note the IBOs of this implementation do not strictly maximize the IAO Mulliken charges.

IBOs can also be generated by another implementation (see function pyscf.lo.ibo.PM). In that function, PySCF builtin Pipek-Meze localization module was used to maximize the IAO Mulliken charges.

**Args:**
mol : the molecule or cell object

orbocc [2D array or a list of 2D array] occupied molecular orbitals or crystal orbitals for each k-point

**Kwargs:**

iaos [2D array] the array of IAOs
exponent [integer] Localization power in PM scheme
grad_tol [float] convergence tolerance for norm of gradients
max_iter [integer] maximum number of iterations
verbose [integer] print level

**Returns:** IBOs in the big basis (the basis defined in mol object).

### 1.40 tools

The **tools** module contains useful tools to dump out integrals, orbital densities, cube files, and other things to interface with external utilities.

**For example, one can write molecular orbitals in molden format as:**

```python
from pyscf import gto, scf from pyscf import lo from pyscf.tools import molden mol = gto.M(
atom = '''
C 3.2883 3.3891 0.2345 C 1.9047 3.5333 0.2237 C 3.8560 2.1213 0.1612 C 1.0888 2.4099 0.1396
C 3.0401 0.9977 0.0771 C 1.6565 1.1421 0.0663 H 3.9303 4.2734 0.2815 H 0.0000 2.5234 0.1311 H 3.4870 0.0000 0.0197 H 1.0145 0.2578 0.0000
''',
basis = ‘cc-pvdz’, symmetry = 1)
mf = scf.RHF(mol) mf.kernel() with open(‘C6H6mo.molden’, ‘w’) as f1:
```

1.40. tools 369
molden.header(mol, f1) molden.orbital_coeff(mol, f1, mf.mo_coeff, ene=mf.mo_energy, occ=mf.mo_occ)

**Cube files can be generated as:** from pyscf import gto, scf from pyscf.tools import cubegen mol = gto.M(atom='''
O 0.0000000, 0.000000, 0.00000000 H 0.761561 , 0.478993, 0.00000000 H -0.761561, 0.478993,
0.0000000''', basis='6-31g*')
mf = scf.RHF(mol).run() cubegen.density(mol, ‘h2o_den.cube’, mf.make_rdm1())

1.40.1 Examples

Relevant examples examples/tools/01-fcidump.py examples/tools/02-molden.py examples/tools/03-print_mo_and_dm.py examples/tools/04-analyze_local_orbitals.py examples/tools/05-cubegen.py examples/tools/06-chgcar.py examples/tools/11-davidson_eigh.py

1.40.2 Program reference

1.40.3 FCIDUMP

FCIDUMP functions (write, read) for real Hamiltonian

```python
pyscf.tools.fcidump.from_chkfile(filename, 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

```python
pyscf.tools.fcidump.from_integrals(filename, h1e, h2e, nmo, nelec, nuc=0, ms=0, orbsym=None, tol=1e-15, float_format=' %.16g')
```

Convert the given 1-electron and 2-electron integrals to FCIDUMP format

```python
pyscf.tools.fcidump.from_mo(mol, filename, mo_coeff, spin='Alpha', symm=None, ene=None, occ=None, ignore_h=True)
```

Use the given MOs to transform the 1-electron and 2-electron integrals then dump them to FCIDUMP.

```python
pyscf.tools.fcidump.from_scf(mf, filename, tol=1e-15, float_format=' %.16g')
```

Use the given SCF object to transfrom the 1-electron and 2-electron integrals then dump them to FCIDUMP.

```python
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

1.40.4 Molden

```python
pyscf.tools.molden.from_mo(mol, filename, mo_coeff, spin='Alpha', symm=None, ene=None, occ=None, ignore_h=True)
```

Dump the given MOs in Molden format

```python
pyscf.tools.molden.from_scf(mf, filename, ignore_h=True)
```

Dump the given SCF object in Molden format

```python
pyscf.tools.molden.load(moldenfile, verbose=0)
```

Extract mol and orbitals from molden file

```python
pyscf.tools.molden.parse(moldenfile, verbose=0)
```

Extract mol and orbitals from molden file

```python
pyscf.tools.molden.read(moldenfile, verbose=0)
```

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 formatted orbitals may have normalization problem. Some visualization tool will complain about the orbital normalization error.

Examples:

```python
>>> mol1, orbl = remove_high_l(mol, mf.mo_coeff)
>>> molden.from_mo(mol1, outputfile, orbl)
```

1.40.5 GAMESS WFN

GAMESS WFN File format

1.40.6 Cubegen


The output cube file has the following format

Comment line

Comment line N_atom Ox Oy Oz # number of atoms, followed by the coordinates of the origin N1 vx1 vy1 vz1 # number of grids along each axis, followed by the step size in x/y/z direction. N2 vx2 vy2 vz2 # ... N3 vx3 vy3 vz3 # ... Atom1 Z1 x y z # Atomic number, charge, and coordinates of the atom ... # ... AtomN ZN x y z # ... Data on grids # (N1*N2) lines of records, each line has N3 elements

class pyscf.tools.cubegen.Cube(mol, nx=80, ny=80, nz=80, resolution=None, origin=None, extent=None)
Read-write of the Gaussian CUBE files

get_coords()
Result: set of coordinates to compute a field which is to be stored in the file.

write(field, fname, comment=None)
Result: .cube file with the field in the file fname.

pyscf.tools.cubegen.density(mol, outfile, dm, nx=80, ny=80, nz=80, resolution=None)
Calculates electron density and write out in cube format.

Args:

mol [Mole] Molecule to calculate the electron density for.

outfile [str] Name of Cube file to be written.


Kwargs:

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. Conflicts to keyword resolution.

ny [int] Number of grid point divisions in y direction.

nz [int] Number of grid point divisions in z direction.

resolution: float Resolution of the mesh grid in the cube box. If resolution is given in the input, the input nx/ny/nz have no effects. The value of nx/ny/nz will be determined by the resolution and the cube box size.
pyscf.tools.cubegen.mep(mol, outfile, dm, nx=80, ny=80, nz=80, resolution=None)
Calculates the molecular electrostatic potential (MEP) and write out in cube format.

Args:
- **mol** [Mole] Molecule to calculate the electron density for.
- **outfile** [str] Name of Cube file to be written.
- **dm** [ndarray] Density matrix of molecule.

Kwargs:
- **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. Conflicts to keyword resolution.
- **ny** [int] Number of grid point divisions in y direction.
- **nz** [int] Number of grid point divisions in z direction.
- **resolution**: float Resolution of the mesh grid in the cube box. If resolution is given in the input, the input nx/ny/nz have no effects. The value of nx/ny/nz will be determined by the resolution and the cube box size.

pyscf.tools.cubegen.orbital(mol, outfile, coeff, nx=80, ny=80, nz=80, resolution=None)
Calculate orbital value on real space grid and write out in cube format.

Args:
- **mol** [Mole] Molecule to calculate the electron density for.
- **outfile** [str] Name of Cube file to be written.
- **coeff** [1D array] coeff coefficient.

Kwargs:
- **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. Conflicts to keyword resolution.
- **ny** [int] Number of grid point divisions in y direction.
- **nz** [int] Number of grid point divisions in z direction.
- **resolution**: float Resolution of the mesh grid in the cube box. If resolution is given in the input, the input nx/ny/nz have no effects. The value of nx/ny/nz will be determined by the resolution and the cube box size.

1.40.7 Print Matrix

pyscf.tools.dump_mat.dump_mo(mol, c, label=None, ncol=5, digits=5, start=0)
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:

```python
>>> from pyscf import gto
>>> mol = gto.M(atom='C 0 0 0')
>>> mo = numpy.eye(mol.nao_nr())
>>> dump_mo(mol, mo)
```

```plaintext
#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
```

```python
def 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:

```python
>>> 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

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:**

```python
>>> 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
```

### 1.40.8 VASP CHGCAR

Vasp CHGCAR file format

See also https://cms.mpi.univie.ac.at/vasp/vasp/CHGCAR_file.html

**class pyscf.tools.chgcar.CHGCAR**

- **cell**
- **nx=60, ny=60, nz=60**
- **resolution=None, margin=3.0**

**Read-write of the Vasp CHGCAR files**

- **get_coords()**
  - Result: set of coordinates to compute a field which is to be stored in the file.

- **write(field, fname, comment=None)**
  - Result: .vasp file with the field in the file fname.

- **pyscf.tools.chgcar.density(cell, outfile, dm, nx=60, ny=60, nz=60, resolution=None)**
  - Calculates electron density and write out in CHGCAR format.
Args:

- **cell** [Mole or Cell object] Mole or pbc Cell. If Mole object is given, the program will guess a cubic lattice for the molecule.
- **outfile** [str] Name of Cube file to be written.
- **dm** [ndarray] Density matrix of molecule.

Kwargs:

- **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.

Returns: No return value. This function outputs a VASP chgcarlike file (with phase if desired)...it can be opened in VESTA or VMD or many other softwares

Examples:

```python
>>> # generates the first MO from the list of mo_coefficients
>>> from pyscf.pbc import gto, scf
>>> from pyscf.tools import chgcar
>>> cell = gto.M(atom='H 0 0 0; H 0 0 1', a=numpy.eye(3)*3)
>>> mf = scf.RHF(cell).run()
>>> chgcar.density(cell, 'h2.CHGCAR', mf.make_rdm1())
```

`pyscf.tools.chgcar.orbital` (cell, outfile, coeff, nx=60, ny=60, nz=60, resolution=None)

Calculate orbital value on real space grid and write out in CHGCAR format.

Args:

- **cell** [Mole or Cell object] Mole or pbc Cell. If Mole object is given, the program will guess a cubic lattice for the molecule.
- **outfile** [str] Name of Cube file to be written.
- **dm** [ndarray] Density matrix of molecule.

Kwargs:

- **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.

Returns: No return value. This function outputs a VASP chgcarlike file (with phase if desired)...it can be opened in VESTA or VMD or many other softwares

Examples:

```python
>>> # generates the first MO from the list of mo_coefficients
>>> from pyscf.pbc import gto, scf
>>> from pyscf.tools import chgcar
>>> cell = gto.M(atom='H 0 0 0; H 0 0 1', a=numpy.eye(3)*3)
>>> mf = scf.RHF(cell).run()
>>> chgcar.orbital(cell, 'h2_mo1.CHGCAR', mf.mo_coeff[:,0])
```
1.40.9 Molpro to PySCF

1.40.10 chkfile

`pyscf.tools.chkfile_util.dump_mo(filename, key='scf')`
Read scf/mcscf information from chkfile, then dump the orbital coefficients.

`pyscf.tools.chkfile_util.molden(filename, key='scf')`
Read scf/mcscf information from chkfile, then convert the scf/mcscf orbitals to molden format.

`pyscf.tools.chkfile_util.mulliken(filename, key='scf')`
Reading scf/mcscf information from chkfile, then do Mulliken population analysis for the density matrix

1.40.11 MO mapping

`pyscf.tools.mo_mapping.mo_1to1map(s)`
Given <i|j>, search for the 1-to-1 mapping between i and j.

**Returns:** a list [j-close-to-i for i in <bral]

`pyscf.tools.mo_mapping.mo_comps(aolabels_or_baslst, mol, mo_coeff, cart=False, orth_method='meta_lowdin')`
Given AO(s), show how the AO(s) are distributed in MOs.

**Args:**
- `aolabels_or_baslst` [filter function or AO labels or AO index] If it’s a function, the AO indices are the items for which the function return value is true.

**Kwargs:**
- `cart` [bool] whether the orbital coefficients are based on cartesian basis.
- `orth_method` [str] The localization method to generated orthogonal AO upon which the AO contribution are computed. It can be one of ‘meta_lowdin’, ‘lowdin’ or ‘nao’.

**Returns:** A list of float to indicate the total contributions (normalized to 1) of localized AOs

**Examples:**

```python
>>> from pyscf import gto, scf
>>> from pyscf.tools import mo_mapping
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1', basis='6-31g')
>>> mf = scf.RHF(mol).run()
>>> comp = mo_mapping.mo_comps('F 2s', mol, mf.mo_coeff)
>>> print('MO-id F-2s components')
...   for i, c in enumerate(comp):
...     print('%-3d %8.10f % (i, c))
```

<table>
<thead>
<tr>
<th>MO-id</th>
<th>components</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0000066344</td>
</tr>
<tr>
<td>1</td>
<td>0.8796915532</td>
</tr>
<tr>
<td>2</td>
<td>0.0590259826</td>
</tr>
<tr>
<td>3</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>4</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>5</td>
<td>0.0435028851</td>
</tr>
<tr>
<td>6</td>
<td>0.0155889103</td>
</tr>
<tr>
<td>7</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>8</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>9</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>10</td>
<td>0.0021017982</td>
</tr>
</tbody>
</table>
pyscf.tools.mo_mapping.mo_map(mol1, mo1, mol2, mo2, base=0, tol=0.5)

Given two orbitals, based on their overlap \langle i|j \rangle, search all orbital-pairs which have significant overlap.

**Returns:** Two lists. First list is the orbital-pair indices, second is the overlap value.

### 1.40.12 Augmented Hessian Newton-Raphson

### 1.40.13 ring

### 1.40.14 c60struct

### 1.41 Miscellaneous

#### 1.41.1 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

**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

#### 1.41.2 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

and the user-defined Hamiltonian for CASSCF

#### 1.42 ccn — Auto-Generated Coupled-Cluster Equations of Arbitrary Order

The `ccn` module implements arbitrary order coupled-cluster equations. See `cc` for standard coupled-cluster implementations up to perturbative triples.
1.42.1 Examples

Relevant examples

1.42.2 Program reference

itrf

\texttt{pyscf.ccn.cc.kernel\_ea\_d}(cc, t2, nroots=1, tolerance=1e-10)

EOM-EA kernel (doubles only). Args:

\begin{itemize}
  \item cc (object): one of pyscf cc.sd objects; t2 (numpy.ndarray): the t2 amplitudes; nroots (int): the number of roots to find; tolerance (float): tolerance to converge to;
\end{itemize}

\textbf{Returns}: EOM-EA energies and amplitudes.

\texttt{pyscf.ccn.cc.kernel\_ea\_s}(cc, t1, nroots=1, tolerance=1e-10)

EOM-EA kernel (singles). Args:

\begin{itemize}
  \item cc (object): one of pyscf cc.sd objects; t1 (numpy.ndarray): the t1 amplitudes; nroots (int): the number of roots to find; tolerance (float): tolerance to converge to;
\end{itemize}

\textbf{Returns}: EOM-EA energies and amplitudes.

\texttt{pyscf.ccn.cc.kernel\_ea\_sd}(cc, t1, t2, nroots=1, tolerance=1e-10)

EOM-EA kernel (singles and doubles). Args:

\begin{itemize}
  \item cc (object): one of pyscf cc.sd objects; t1 (numpy.ndarray): the t1 amplitudes; t2 (numpy.ndarray): the t2 amplitudes; nroots (int): the number of roots to find; tolerance (float): tolerance to converge to;
\end{itemize}

\textbf{Returns}: EOM-EA energies and amplitudes.

\texttt{pyscf.ccn.cc.kernel\_ea\_sdt}(cc, t1, t2, t3, nroots=1, tolerance=1e-10)

EOM-EA kernel (singles, doubles and triples). Args:

\begin{itemize}
  \item cc (object): one of pyscf cc.sd objects; t1 (numpy.ndarray): the t1 amplitudes; t2 (numpy.ndarray): the t2 amplitudes; t3 (numpy.ndarray): the t3 amplitudes; nroots (int): the number of roots to find; tolerance (float): tolerance to converge to;
\end{itemize}

\textbf{Returns}: EOM-EA energies and amplitudes.

\texttt{pyscf.ccn.cc.kernel\_ground\_state\_d}(cc, tolerance=1e-10, maxiter=50)

A ground-state kernel (doubles only). Args:

\begin{itemize}
  \item cc (object): one of pyscf cc.sd objects; tolerance (float): tolerance to converge to; maxiter (int): the maximal number of iterations;
\end{itemize}

\textbf{Returns}: Ground state amplitudes and energy.

\texttt{pyscf.ccn.cc.kernel\_ground\_state\_s}(cc, tolerance=1e-10, maxiter=50)

A ground-state kernel (singles). Args:

\begin{itemize}
  \item cc (object): one of pyscf cc.sd objects; tolerance (float): tolerance to converge to; maxiter (int): the maximal number of iterations;
\end{itemize}
**Returns:** Ground state amplitudes and energy.

```python
def pyscf.ccn.cc.kernel_ground_state_sd(cc, tolerance=1e-10, maxiter=50):
    A ground-state kernel (singles and doubles). Args:
        cc (object): one of pyscf ccsd objects; tolerance (float): tolerance to converge to; maxiter (int): the
        maximal number of iterations;
    Returns: Ground state amplitudes and energy.
```

**Returns:** Ground state amplitudes and energy.

```python
def pyscf.ccn.cc.kernel_ground_state_sdt(cc, tolerance=1e-10, maxiter=50):
    A ground-state kernel (singles, doubles and triples). Args:
        cc (object): one of pyscf ccsd objects; tolerance (float): tolerance to converge to; maxiter (int): the
        maximal number of iterations;
    Returns: Ground state amplitudes and energy.
```

**Returns:** EOM-IP energies and amplitudes.

```python
def pyscf.ccn.cc.kernel_ip_d(cc, t2, nroots=1, tolerance=1e-10):
    EOM-IP kernel (doubles only). Args:
        cc (object): one of pyscf ccsd objects; t2 (numpy.ndarray): the t2 amplitudes; nroots (int): the
        number of roots to find; tolerance (float): tolerance to converge to;
    Returns: EOM-IP energies and amplitudes.
```

**Returns:** EOM-IP energies and amplitudes.

```python
def pyscf.ccn.cc.kernel_ip_s(cc, t1, nroots=1, tolerance=1e-10):
    EOM-IP kernel (singles). Args:
        cc (object): one of pyscf ccsd objects; t1 (numpy.ndarray): the t1 amplitudes; nroots (int): the
        number of roots to find; tolerance (float): tolerance to converge to;
    Returns: EOM-IP energies and amplitudes.
```

**Returns:** EOM-IP energies and amplitudes.

```python
def pyscf.ccn.cc.kernel_ip_sd(cc, t1, t2, nroots=1, tolerance=1e-10):
    EOM-IP kernel (singles and doubles). Args:
        cc (object): one of pyscf ccsd objects; t1 (numpy.ndarray): the t1 amplitudes; t2 (numpy.ndarray): the
        t2 amplitudes; nroots (int): the number of roots to find; tolerance (float): tolerance to converge to;
    Returns: EOM-IP energies and amplitudes.
```

**Returns:** EOM-IP energies and amplitudes.

```python
def pyscf.ccn.cc.kernel_ip_sdt(cc, t1, t2, t3, nroots=1, tolerance=1e-10):
    EOM-IP kernel (singles, doubles and triples). Args:
        cc (object): one of pyscf ccsd objects; t1 (numpy.ndarray): the t1 amplitudes; t2 (numpy.ndarray): the
        t2 amplitudes; t3 (numpy.ndarray): the t3 amplitudes; nroots (int): the number of roots to find;
        tolerance (float): tolerance to converge to;
    Returns: EOM-IP energies and amplitudes.
```

**Returns:** A ground-state lambda kernel (doubles only). Args:

```python
def pyscf.ccn.cc.kernel_lambda_d(cc, t2, tolerance=1e-10, maxiter=50):
    A ground-state lambda kernel (doubles only). Args:
        cc (object): one of pyscf ccsd objects; t2 (numpy.ndarray): the t2 amplitudes; tolerance (float): the
        tolerance to converge; maxiter (int): the maximal number of iterations;
```
Returns: Ground state lambda amplitudes.

\texttt{pyscf.ccn.cc.kernel\_lambda\_s (cc, t1, tolerance=1e-10, maxiter=50)}

A ground-state lambda kernel (singles). Args:

- \texttt{cc} (object): one of pyscf ccsd objects;
- \texttt{t1} (numpy.ndarray): the t1 amplitudes;
- \texttt{tolerance} (float): tolerance to converge to;
- \texttt{maxiter} (int): the maximal number of iterations;

Returns: Ground state lambda amplitudes.

\texttt{pyscf.ccn.cc.kernel\_lambda\_sd (cc, t1, t2, tolerance=1e-10, maxiter=50)}

A ground-state lambda kernel (singles and doubles). Args:

- \texttt{cc} (object): one of pyscf ccsd objects;
- \texttt{t1} (numpy.ndarray): the t1 amplitudes;
- \texttt{t2} (numpy.ndarray): the t2 amplitudes;
- \texttt{tolerance} (float): tolerance to converge to;
- \texttt{maxiter} (int): the maximal number of iterations;

Returns: Ground state lambda amplitudes.

\texttt{pyscf.ccn.cc.kernel\_lambda\_sdt (cc, t1, t2, t3, tolerance=1e-10, maxiter=50)}

A ground-state lambda kernel (singles, doubles and triples). Args:

- \texttt{cc} (object): one of pyscf ccsd objects;
- \texttt{t1} (numpy.ndarray): the t1 amplitudes;
- \texttt{t2} (numpy.ndarray): the t2 amplitudes;
- \texttt{t3} (numpy.ndarray): the t3 amplitudes;
- \texttt{tolerance} (float): tolerance to converge to;
- \texttt{maxiter} (int): the maximal number of iterations;

Returns: Ground state lambda amplitudes.

### 1.43 \texttt{nao} — Numerical Atomic Orbitals

The \texttt{nao} module is a simple and elegant python library to perform iterative Time Dependent Density Functional Theory (TDDFT) using ground state data from different programs,

- \texttt{Sieta}
- \texttt{PYSCF}
- \texttt{OpenMX}
- \texttt{GPAW}

#### 1.43.1 \texttt{nao.tddft\_iter} — NAO: Iterative Time Dependent Density Functional Theory

#### 1.43.2 \texttt{nao.tddft\_tem} — NAO: Electron Energy Loss Spectroscopy Within TDDFT

#### 1.43.3 \texttt{nao.comp\_spatial\_distributions} — NAO: Spatial Distribution Density change

#### 1.43.4 \texttt{nao.examples} — NAO Examples

Examples Showing use case of the NAO modules.
Interfacing with ASE and Siesta

The Atomic Simulation Environment (ASE) Greatly simplify the scripting of your problem and combine calculations from the ground state with Siesta and excited states with PYSCF-NAO.

Simple Polarizability Calculation

To perform a simple calculation returning the polarizability of the system one just need the following script,

```python
from ase.units import Ry, eV, Ha
from ase.calculators.siesta import Siesta
from ase import Atoms
import numpy as np
import matplotlib.pyplot as plt

# Define the systems
Na8 = Atoms('Na8',
            positions=[[1.90503810, 1.56107288, 0.00000000],
                        [-1.90503810, 1.56107288, 0.00000000],
                        [1.90503810, -1.56107288, 0.00000000],
                        [-1.90503810, -1.56107288, 0.00000000],
                        [0.00000000, 0.00000000, 2.08495836],
                        [0.00000000, 0.00000000, -2.08495836],
                        [0.00000000, 3.22798122, 2.08495836],
                        [0.00000000, 3.22798122, -2.08495836]],
            cell=[20, 20, 20])

# enter siesta input
siesta = Siesta(
    mesh_cutoff=150 * Ry,
    basis_set='DZP',
    pseudo_qualifier='',
    energy_shift=(10 * 10**-3) * eV,
    fdf_arguments={
        'SCFMustConverge': False,
        'COOP.Write': True,
        'WriteDenchar': True,
        'PAO.BasisType': 'split',
        'DM.Tolerance': 1e-4,
        'DM.MixingWeight': 0.01,
        'MaxSCFIterations': 300,
        'DM.NumberPulay': 4,
        'XML.Write': True})

Na8.set_calculator(siesta)
e = Na8.get_potential_energy()

freq, pol = siesta.get_polarizability_pyscf_inter(label="siesta", jcutoff=7, iter_→broadening=0.15/Ha,
    xc_code='LDA,PZ', tol_loc=1e-6, tol_biloc=1e-7, freq = np.arange(0.0, 5.0, 0.1.43. nao — Numerical Atomic Orbitals

---

1.43. nao — Numerical Atomic Orbitals 381
# plot polarizability
plt.plot(freq, pol[:, 0, 0].imag)
plt.show()

You will need first to install Siesta, ASE and to setup the interface between Siesta and ASE. Everything is explain on the Siesta webpage of ASE

## Non-Resonant Raman Spectra

In ASE there is also the possibility to calculate Non-Resonant Raman spectra using the Siesta calculator together with PYSCF,

```
from ase.units import Ry, eV, Ha
from ase.calculators.siesta import Siesta
from ase.calculators.siesta.siesta_raman import SiestaRaman
from ase import Atoms
import numpy as np

# Define the systems
# example of Raman calculation for CO2 molecule, 
# comparison with QE calculation can be done from
# https://github.com/maxhutch/quantum-espresso/blob/master/PHonon/examples/example15/
→README
CO2 = Atoms('CO2',
            positions=[[0.09026, -0.020241, 0.026760],
                        [1.167544, 0.012723, 0.071808],
                        [-1.185592, -0.053316, -0.017945]],
            cell=[20, 20, 20])

# enter siesta input
siesta = Siesta(
                mesh_cutoff=150 * Ry,
                basis_set='DZP',
                pseudo_qualifier='',
                energy_shift=(10 * 10**-3) * eV,
                fdf_arguments={
                        'SCFMustConverge': False,
                        'COOP.Write': True,
                        'WriteDenchar': True,
                        'PAO.BasisType': 'split',
                        'DM.Tolerance': 1e-4,
                        'DM.MixingWeight': 0.01,
                        'MaxSCFIterations': 300,
                        'DM.NumberPulay': 4,
                        'XML.Write': True,
                        'DM.UseSaveDM': True})

CO2.set_calculator(siesta)

ram = SiestaRaman(CO2, siesta, label="siesta", jcutoff=7, iter_broadening=0.15/Ha,
```
Advanced Examples

**nao.examples.qmd_C60 — NAO Examples: QMD C60**

A demonstration of the NAO module used to calculate the polarizability of C60 at room temperature using Quantum Molecular Dynamic (QMD). The relaxation and the ground state calculations are done using the Siesta program. This example will calculate the geometries of 5000 steps of C60 using Siesta and the polarizability will be obtained by calculating 200 steps. This example MUST be done on a server because it will perform 200 calculations. This example has been done for the Torque resources manager and will need to be adapted to your case before to run.

To start the calculations, we first need to generate the 5000 geometries using Molecular Dynamic (MD) with temperature controlled by means of a Nosé thermostat. For this we use the following siesta input file,

```plaintext
SystemLabel  siesta
PAO.EnergyShift  50  meV
PAO.BasisSize    DZP
PAO.SoftDefault  .true.
PAO.BasisType    split

XC.functional    GGA
XC.authors       PBE

WriteIonPlotFiles .false.
WriteCoorXmol    .true.
WriteMDXmol      .true.

@include "C60_org.fdf"

AtomCoorFormatOut Ang

DM.UseSaveDM    .true.
DM.Tolerance    0.0001
DM.MixingWeight 0.25
MaxSCFIterations 400
DM.NumberPulay  4

MD.TypeOfRun    Nose
MD.InitialTemperature 300  K
MD.TargetTemperature 300  K
MD.NumCGsteps  5000
MD.FinalTimeStep 5000
UseSaveData    .true.

MeshCutOff     250  Ry
MD.UseSaveXV   .true.

SaveRho        .false.
COOP.Write     .false.
```
The file containing the position of the atoms can be downloaded with the following link.

**You run this calculation with the normal siesta command (don’t forget the C.psf file)::**

```bash
siesta < siesta_C60_thermal.fdf > siesta.out
```

or with a bash script if you run on a server. Once, this first calculation finish you will be left with a siesta.ANI file containing the geometry at each time step. We will first generate the xyz files corresponding to the geometries for comomdity,

```bash
#!/bin/bash
fname=${1:-siesta.ANI}
suffi=${2:-0}
echo ".ANI file with MM geometries : " $fname
echo "Suffix to choose geometries for QM part : " $suffi
mkdir -p xyz
cp $fname xyz/
echo "cd xyz"
cd xyz
LIT=`head -n1 $fname`
NLINES_PER_FILE=$(expr $LIT + 2)
echo "rm x*"
rm x*
echo "split -l $NLINES_PER_FILE -d -a 6 $fname"
split -l $NLINES_PER_FILE -d -a 6 $fname
echo "xyz2fdf.py $1*$suffi > xy22fdf.py.out"
xyz2fdf.py $1*$suffi > xyz2fdf.py.out
cd ..
ls xyz/*.*
```

This will create a folder xyz containing the 5000 generated geometries. We need then to generate the fdf geometries file for siesta:

```bash
cd cyz
python xyz2fdf.py
```

xyz2fdf.py is a small python script that will generate the geometries for 200 calculations from the 5000 previous geometries because 5000 is a bit too much. The script look like,

```python
from __future__ import division
import numpy as np
import ase.io as io
import subprocess

def write_geofdf(geom, outfile="geo.fdf"):
    from ase.data import chemical_symbols
    f=open(outfile, "w")
    f.write("NumberOfAtoms\n".format(len(geom)))
    f.write("NumberOfSpecies\n".format(len(set(geom.numbers))))
    f.write("\n")
    f.write(’%block ChemicalSpeciesLabel\n’) 
species_label = {}
```
The script will generate the siesta input files in different trajectory, don’t forget to add in the xyz directory the siesta input

1.43. nao — Numerical Atomic Orbitals
DM.NumberPulay 4
MaxSCFIterations 400

MD.TypeOfRun CG
MD.NumCGsteps 0
MD.MaxForceTol 0.02 eV/Ang

SolutionMethod Diagon
MeshCutOff 250 Ry

WriteCoorXmol .True.
WriteDenchar .True.
COOP.Write .True.
XML.Write .True.

After running the xyz2fdf.py script you should have 200 folders containing the 3 input files,
- siesta_C60.fdf
- geo.fdf
- C.psf

I advise you to create a new folder, and to copy all the calculations input into this new folder:

cd ..
mkdir calculations
cp -r xyz/calc_* calculations
cd calculations

Now we will need script to perform the siesta and nao calculations. Here, I will perform in total 50 jobs, each running 4 calculations. Each jobs will be using 6 cpus. I’m assuming that you are using the Torque managing system. We will use two python script to perform the calculations,
- submit_jobs.py
- calc_polarizability.py

The first one is the script that you call in order to submit the 200 jobs and it call the second script calc_polarizability.py. This second script will perform the siesta and nao calculations for each configurations.

submit_jobs.py:

```python
from __future__ import division
import numpy as np
import subprocess

# The bash script for Torque
script = """#!/bin/bash
#PBS -q parallel
#PBS -l nodes=1:ppn=6
#PBS -l mem=5gb
#PBS -l cput=400:00:00
#PBS -N calc_C60
ulimit -s unlimited
export NPROCS=`wc -l $PBS_NODEFILE`
export OMP_NUM_THREADS=$NPROCS"
```
export PATH="/Path/To/Python/binary:$PATH"
export LD_LIBRARY_PATH="/Path/To/Needed/Library:$LD_LIBRARY_PATH" # libxc, libcint, libxcfun
export PYTHONPATH="/Path/To/Pyscf:$PYTHONPATH"

# ASE necessary for using ase.units
#ASE
export ASE_HOME="/Path/To/ASE"
export PYTHONPATH="/($ASE_HOME):$PYTHONPATH"export PATH="/($ASE_HOME)/tools:$PATH"

# you may need to change the job directory
export LSCRATCH_DIR="/scratch/$USER/jobs/$PBS_JOBID"
mkdir -p $LSCRATCH_DIR
cd $PBS_O_WORKDIR

# load the right module depending your pyscf/siesta compilation
# you better to use the same compiler for both program to avoid problems
ml purge
ml load intel/2015b FFTW/3.3.4-intel-2015b

"

# the range of our 200 calculations
xyz_range = np.arange(0, 5000, 25)

start = 0
step = 4
end = 0

while end < xyz_range.size:
    end = start + step
    if end > xyz_range.size:
        end = xyz_range.size + 1

    calcs = xyz_range[start:end]
    include = ["calc_polarizability.py"]
    for i, xyz in enumerate(calcs[0:calcs.shape[0]]):
        if xyz < 10:
            num = "00000".format(xyz)
        elif xyz < 100:
            num = "0000".format(xyz)
        elif xyz < 1000:
            num = "000".format(xyz)
        elif xyz < 10000:
            num = "00".format(xyz)
        else:
            raise ValueError("xyz too large?? (0)".format(xyz))
    include.append("calc_"+num)

    fcalc = calcs[0]
    ecalc = calcs[calcs.shape[0]-1]+25

    lines = script
    for files in include:
        lines += "cp -r " + files + " $LSCRATCH_DIR
    lines += "cd $LSCRATCH_DIR

1.43. nao — Numerical Atomic Orbitals
```python
lines += "./calc_polarizability.py --np \${NPROCS} --start \{0\} --end \{1\} >&\n--calc_{0}to{1}.out\n".format(fcalc, ecalc)

lines += "export RESULTS_DIR=$PBS_O_WORKDIR\n"
lines += "mkdir -p $RESULTS_DIR\n"
lines += "cp -r * $RESULTS_DIR\n"
fname = "run.calc_C60_{0}to_{1}.sh".format(fcalc, ecalc)
f = open(fname, "w")
f.write(lines) # write bash script
f.close()
start = end

# submit job to Torque
subprocess.call("qsub " + fname, shell=True)
```

calc_polarizability.py

```python
#!/Path/To/Python/binary
from __future__ import print_function, division
import os, numpy as np
from pyscf.nao import tddft_iter
import sys
import argparse
from ase.units import Ry, eV, Ha
import subprocess
from timeit import default_timer as timer

def run_tddft_iter():
    t1 = timer()
    td = tddft_iter(label="siesta", iter_broadening=0.0035/Ha, xc_code='LDA,PZ', ...
    level=0, tddft_iter_tol=1e-3, tol_loc=1e-4, tol_biloc=1e-6)
    t2 = timer()
    print("time tddft_iter = ", t2-t1)
    omegas = np.arange(2.0, 8.0, 2.5E-3)/Ha + 1j*td.eps
    t3 = timer()
    pxx = -td.comp_polariz_nonin_ave(omegas)
    t4 = timer()
    print("time chi0 = ", t4-t3)
    data = np.zeros((omegas.shape[0], 3))
    data[:, 0] = omegas.real*Ha
    data[:, 1] = pxx.real
    data[:, 2] = pxx.imag
    np.savetxt('polarizability_nonin_siesta.avg.txt', data)
    t5 = timer()
    pxx = -td.comp_polariz_inter_ave(omegas)
    t6 = timer()
    print("time chi = ", t6-t5)
    data = np.zeros((omegas.shape[0], 3))
    data[:, 0] = omegas.real*Ha
    data[:, 1] = pxx.real
    data[:, 2] = pxx.imag
    np.savetxt('polarizability_inter_siesta.avg.txt', data)
    print("nb call:")
    print("rf0_ncalls = \{0\}, matvec_ncalls =\{1\}".format(td.rf0_ncalls, td.matvec_ ...
    --ncalls))
    t7 = timer()
```
print("total time = ", t7-t1)

parser = argparse.ArgumentParser()
parser.add_argument('--np', type=int, default=1, help="number of processor to use")
parser.add_argument('--start', type=int, default=0, help="starting calc")
parser.add_argument('--end', type=int, default=25, help="end calculation")

args_par = parser.parse_args()

xyz_range = np.arange(args_par.start, args_par.end, 25)
mpi_exe="/scratch/mbarbry/intel/intelpython2/bin/mpirun"
siesta_path="/scratch/software/SIESTA/4.0b-485-intel-2015b/siesta"

siesta_exe = mpi_exe + " -np {0} " .format(args_par.np) + siesta_path + " < siesta.fdf \rightarrow siesta.out"

for i, xyz in enumerate(xyz_range):
    if xyz < 10:
        num = "0000{0}".format(xyz)
    elif xyz < 100:
        num = "000{0}".format(xyz)
    elif xyz < 1000:
        num = "00{0}".format(xyz)
    elif xyz < 10000:
        num = "0{0}".format(xyz)
    else:
        raise ValueError("xyz too large?? {0}".format(xyz))

path = "calc_{0}" + num
os.chdir(path)
# Run siesta
subprocess.call("export OMP_NUM_THREADS=1", shell=True)
subprocess.call(siesta_exe, shell=True)

# Run pyscf.nao
subprocess.call("export OMP_NUM_THREADS={0}".format(args_par.np), shell=True)
run_tddft_iter()
oS.chdir("/../")

1.44  lib — Helper functions, parameters, and C extensions

1.44.1  lib.parameters

1.44.2  lib.logger

Logger object

1.44.3  numpy extensions

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.

lib.numpy_helper.cartesian_prod(arrays, out=None)
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:

```python
>>> 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]])
```

lib.numpy_helper.

**cond** *(x, p=None)*

Compute the condition number

**condense** *(opname, a, locs)*

```python
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
```

lib.numpy_helper.

**ddot** *(a, b, alpha=1, c=None, beta=0)*

Matrix-matrix multiplication for double precision arrays

**direct_sum** *(subscripts, *operands)*

Apply the summation over many operands with the einsum fashion.

Examples:

```python
>>> a = numpy.ones((6,5))
>>> b = numpy.ones((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)

lib.numpy_helper.dot(a, b, alpha=1, c=None, beta=0)
Matrix-matrix multiplication for double complex arrays using Gauss’s complex multiplication algorithm

lib.numpy_helper.hermi_sum(a, axes=None, hermi=1, inplace=False, out=None)
a + a.T for better memory efficiency
Examples:

```python
>>> transpose_sum(numpy.arange(4.).reshape(2,2))
[[ 0. 3.]
[ 3. 6.]]
```

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:

```python
>>> unpack_row(numpy.arange(9.).reshape(3,3), 1)
[[ 0. 3. 6.]
[ 3. 4. 7.]
[ 6. 7. 8.]]
```

lib.numpy_helper.norm(x, ord=None, axis=None)
numpy.linalg.norm for numpy 1.6.*

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:

```python
>>> pack_tril(numpy.arange(9).reshape(3,3))
[0 3 4 6 7 8]
```

lib.numpy_helper.solve_lineq_by_SVD(a, b)
a * x = b

lib.numpy_helper.take_2d(a, idx, idy, out=None)
a[idx,idy]
Examples:

```python
>>> out = numpy.arange(9.).reshape(3,3)
>>> take_2d(a, [0,2], [0,2])
[[ 0. 2.]
[ 6. 8.]]
```

lib.numpy_helper.takebak_2d(out, a, idx, idy)
Reverse operation of take_2d. out[idx,idy] += a
Examples:

```python
>>> 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.]]
```

lib.numpy_helper.transpose(a, axes=None, inplace=False, out=None)

Transpose array for better memory efficiency

Examples:

```python
>>> transpose(numpy.ones((3,2)))
[[ 1.  1.  1.]
 [ 1.  1.  1.]]
```

lib.numpy_helper.transpose_sum(a, inplace=False, out=None)

$a + a.T$ for better memory efficiency

Examples:

```python
>>> transpose_sum(numpy.arange(4.).reshape(2,2))
[[ 0.  3.]
 [ 3.  6.]]
```

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:

```python
>>> unpack_row(numpy.arange(6.), 0)
[ 0.  1.  3.]
>>> unpack_tril(numpy.arange(6.))[0]
[ 0.  1.  3.]
```

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:

```python
>>> 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.]]
```
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

### 1.44.4 scipy extensions

Extension to scipy.linalg module

lib.linalg_helper.cho_solve($a, b$)
Solve $ax = b$, where $a$ is hermitian matrix

lib.linalg_helper.davidson($aop, x0, precond, tol=1e-14, max_cycle=50, max_space=12, lindep=1e-14, max_memory=2000, dot=<built-in function dot>, callback=None, nroots=1, lessio=False, verbose=2$)

**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] 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 environment.

- $nroots$ [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 I/O and less computational cost. When I/O is slow, the second method can be considered.

**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:
```python
>>> 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)
```

Lib. linalg.helper. davidson1(aop, x0, precond, tol=1e-14, 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)


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
- **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 environment.
- **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.

Examples:

```python
>>> 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


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] 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 environment.
- **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** [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:

```python
>>> 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, nroots=2)
>>> len(e)
2
lib.linalg_helper.dsolve(aop, b, precond, tol=1e-14, max_cycle=30, dot=<built-in function dot>,
lindep=1e-16, verbose=0)
Davidson iteration to solve linear equation. It works bad.

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.

Kwarg:

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:

```python
>>> from pyscf import lib
g = 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([-0.22975583, -0.22975583, -1.78085650, -1.78085650, -1.74189134,
        -0.98998583, -0.98998583, -0.40322226, -0.30242374, -0.07608981]),
   array([[ 1.00000000,  0.00000000,  0.00000000,  0.00000000],
           [ 0.00000000,  1.00000000,  0.00000000,  0.00000000],
           [ 0.00000000,  0.00000000,  1.00000000,  0.00000000],
           [ 0.00000000,  0.00000000,  0.00000000,  1.00000000]])
```

```python
>>> from pyscf import gto, lib
>>> 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('cint1e_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, -1.78085650, -1.78085650, -1.74189134,
        -0.98998583, -0.98998583, -0.40322226, -0.30242374, -0.07608981]),
   array([[ 0.70710678,  0.70710678,  0.70710678,  0.70710678,  0.70710678],
           [ 0.70710678,  0.70710678,  0.70710678,  0.70710678,  0.70710678],
           [ 0.70710678,  0.70710678,  0.70710678,  0.70710678,  0.70710678],
           [ 0.70710678,  0.70710678,  0.70710678,  0.70710678,  0.70710678],
           [ 0.70710678,  0.70710678,  0.70710678,  0.70710678,  0.70710678]]))
```

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).

396 Chapter 1. Contents
Args:

- **aop** [function(x) => array_like] 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 environment.

**Returns:** x : 1D array like b

Examples:

```python
>>> 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
```

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.

1.44.5 **lib.chkfile**

Chkfile is a HDF5 file.
Functions to access key/value in chkfile

```python
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:**

```python
>>> 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']
```

```python
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

```python
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:**

```python
>>> 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>
```

Quickly loading object from chkfile

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`, e.g.:
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'))
cmc.kernel()

1.44.6 lib.diis

DIIS

class lib.diis.DIIS(dev=None, filename=None)
Direct inversion in the iterative subspace method.

Attributes:

- space [int] DIIS subspace size. The maximum number of the vectors to be stored.

- min_space The minimal size of subspace before DIIS extrapolation.

Functions:

- update(x, xerr=None) : If xerr the error vector is given, this function will push the target vector and error vector in the DIIS subspace, and use the error vector to extrapolate the vector and return the extrapolated vector. If xerr is None, this function will take the difference between the current given vector and the last given vector as the error vector to extrapolate the vector.

Examples:

```python
>>> from pyscf import gto, scf, lib
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1', basis='ccpvdz')
>>> mf = scf.RHF(mol)
>>> h = mf.get_hcore()
>>> s = mf.get_ovlp()
>>> e, c = mf.eig(h, s)
>>> occ = mf.get_occ(e, c)
>>> # DIIS without error vector
>>> adiis = lib.diis.DIIS()
>>> for i in range(7):
...     dm = mf.make_rdm1(c, occ)
...     f = h + mf.get_veff(mol, dm)
...     if i > 1:
...         f = adiis.update(f)
...     e, c = mf.eig(f, s)
...     print('%d = %12.10f' % (i, mf.energy_tot(dm, h, mf.get_veff(mol, dm)))
E_0 = -1.050329433306
E_1 = -1.098566175145
E_2 = -1.100103795287
E_3 = -1.100152104615
E_4 = -1.100153709222
E_5 = -1.100153764848
E_6 = -1.100153764878
```
```python
>>> # Take Hartree-Fock gradients as the error vector
>>> adiis = lib.diis.DIIS()
>>> for i in range(7):
...     dm = mf.make_rdm1(c, occ)
...     f = h + mf.get_veff(mol, dm)
...     if i > 1:
...         f = adiis.update(f, mf.get_grad(c, occ, f))
...     e, c = mf.eig(f, s)
...     print('E_%d = %.12f' % (i, mf.energy_tot(dm, h, mf.get_veff(mol, dm))))
E_0 = -1.050329433306
E_1 = -1.098566175145
E_2 = -1.10013795287
E_3 = -1.100152104615
E_4 = -1.100153763813
E_5 = -1.100153764878
E_6 = -1.100153764878
```

**update** *(x, xerr=None)*

Extrapolate vector

- If xerr the error vector is given, this function will push the target vector and error vector in the DIIS subspace, and use the error vector to extrapolate the vector and return the extrapolated vector. * If xerr is None, this function will take the difference between the current given vector and the last given vector as the error vector to extrapolate the vector.

```python
lib.diis.restore(dev, filename)
```

Restore DIIS vectors/error-vectors from given file.

### 1.44.7 Other helper functions

**Background mode**

```python
lib.call_in_background(*fns, **kwargs)
```

Within this macro, function(s) can be executed asynchronously (the given functions are executed in background).

Attributes:

- **sync (bool)**: Whether to run in synchronized mode. The default value is `False` (asynchronous mode).

Examples:

```python
>>> with call_in_background(fun) as async_fun:
...     async_fun(a, b)  # == fun(a, b)
...     do_something_else()

>>> with call_in_background(fun1, fun2) as (afun1, afun2):
...     afun2(a, b)
...     do_something_else()
...     afun2(a, b)
...     do_something_else()
...     afun1(a, b)
...     do_something_else()
```
Temporary HDF5 file

class lib.H5TmpFile(filename=None, *args, **kwargs)
Create and return an HDF5 temporary file.

Kwargs:

filename [str or None] If a string is given, an HDF5 file of the given filename will be created. The temporary file will exist even if the H5TmpFile object is released. If nothing is specified, the HDF5 temporary file will be deleted when the H5TmpFile object is released.

The return object is an h5py.File object. The file will be automatically deleted when it is closed or the object is released (unless filename is specified).

Examples:

>>> from pyscf import lib
>>> ftmp = lib.H5TmpFile()

OpenMP threads

lib.num_threads(n=None)
Set the number of OMP threads. If argument is not specified, the function will return the total number of available OMP threads.

It’s recommended to call this function to set OMP threads than “os.environ[‘OMP_NUM_THREADS’] = int(n)”. This is because environment variables like OMP_NUM_THREADS were read when a module was imported. They cannot be reset through os.environ after the module was loaded.

Examples:

>>> from pyscf import lib
>>> print(lib.num_threads())
8
>>> lib.num_threads(4)
4
>>> print(lib.num_threads())
4

class lib.with_omp_threads(nthreads=None)
Using this macro to create a temporary context in which the number of OpenMP threads are set to the required value. When the program exits the context, the number OpenMP threads will be restored.

Args: nthreads : int

Examples:

>>> from pyscf import lib
>>> print(lib.num_threads())
8
>>> with lib.with_omp_threads(2):
...     print(lib.num_threads())
2
>>> print(lib.num_threads())
8
Capture stdout

**class** `lib.capture_stdout`
redirect all stdout (c printf & python print) into a string

Examples:

```python
>>> import os
>>> from pyscf import lib
>>> with lib.capture_stdout() as out:
...    os.system('ls')
>>> print(out.read())
```

Other helper functions in `lib.misc`

Some hacky functions

```python
lib.misc.flatten(lst)
flatten nested lists x[0] + x[1] + x[2] + ...
```

Examples:

```python
>>> flatten([[0, 2], [1], [[9, 8, 7]]])
[0, 2, 1, [9, 8, 7]]
```

1.45 prop — Molecular properties

The `prop` module provides electric and magnetic properties for molecules.

The list of modules described in this chapter is:

1.45.1 efg —

*Examples*

No examples

*Program reference*

**dhf**

Electric field gradients, nuclear quadrupolar coupling and Mossbauer spectroscopy for relativistic 4-component DHF and DKS methods. (In testing)

**rhf**

Electric field gradients, nuclear quadrupolar coupling and Mossbauer spectroscopy for non-relativistic (or sf-x2c) mean-field and post-HF methods. (In testing)

Ref:
uhf

1.45.2 gtensor —

The gtensor module computes electronic g tensors for mean-field wavefunctions.

To calculate the g-tensor of CN molecule, first run an unrestricted HF using the module scf, then calculate the g-tensor using the module gtensor:

```python
from pyscf import gto, scf, dft
from pyscf.prop import gtensor
mol = gto.M(atom='''
  C 0 0 0
  N 0 0 1.1747
''',
  basis='ccpvdz', spin=1, charge=0, verbose=3)
mf = scf.UHF(mol).run()
gobj = gtensor.uhf.GTensor(mf).set(verbose=4)
gobj.kernel()
```

Details about the examples can be find from here:

**Examples**

Relevant examples examples/prop/02-g_tensor.py

**Program reference**

**dhf**

Dirac Hartree-Fock g-tensor (In testing)

Refs: TCA, 129, 715

**uhf**

Non-relativistic unrestricted Hartree-Fock g-tensor

**Refs:** JPC, 101, 3388 JCP, 115, 11080 JCP, 119, 10489

Note g-tensor = 1/\mu_B d^2 E/ dB dS In some literature, \mu_B is not explicitly presented in the perturbation formula.

```python
class pyscf.prop.gtensor.uhf.GTensor(mf)
dE = B dot gtensor dot s
```

Attributes:
dia_soc2e [str or bool] 2-electron spin-orbit coupling for diamagnetic term. Its value can be ‘SSO’, ‘SOO’, ‘SSO+SOO’. None/False or True (=‘SSO+SOO’). Default is False.


koseki_charge [bool] Whether to use Koseki effective SOC charge in 1-electron diamagnetic term and paramagnetic term. Default is False.

get_fock (nmrobj, dm0=None, gauge_orig=None)
First order partial derivatives of Fock matrix wrt external magnetic field. frac{partial F}{partial B}

make_dia_gc2e (gobj, dm0, gauge_orig, sso_qed_fac=1)
Note the side effects of set_common_origin

solve_mol (nmrobj, mo_energy=None, mo_coeff=None, mo_occ=None, h1=None, s1=None, with_cphf=None)
Solve the first order equation

Kwargs:

    with_cphf [boolean or function(dm_mo) => v1_mo] If a boolean value is given, the value determines whether CPHF equation will be solved or not. The induced potential will be generated by the function gen_vind. If a function is given, CPHF equation will be solved, and the given function is used to compute induced potential

pyscf.prop.gtensor.uhf.align (gtensor)
Transform the orientation of g-tensor. The new orientations are the eigenvector of G matrix (G=g.gT)

pyscf.prop.gtensor.uhf.dia (gobj, dm0, gauge_orig=None)
Note the side effects of set_common_origin

pyscf.prop.gtensor.uhf.get_j_amfi (mol, dm0)
Atomic-mean-field approximation

pyscf.prop.gtensor.uhf.get_jk_amfi (mol, dm0)
Atomic-mean-field approximation

pyscf.prop.gtensor.uhf.make_dia_gc2e (gobj, dm0, gauge_orig, sso_qed_fac=1)
Note the side effects of set_common_origin

uks

Non-relativistic unrestricted Kohn-Sham g-tensor (In testing)

Refs: JPC, 101, 3388 JCP, 115, 11080 JCP, 119, 10489

class pyscf.prop.gtensor.uks.GTensor (scf_method)
    dE = B dot gtensor dot s

    get_fock (nmrobj, dm0=None, gauge_orig=None)
        First order Fock matrix wrt external magnetic field

1.45.3 magnetizability —

The magnetizability module implements magnetizabilities for mean-field wavefunctions.
Examples

No example

Program reference

rhf

Non-relativistic magnetizability tensor for RHF (In testing)

```python
import pyscf

pyscf.prop.magnetizability.rhf.dia(magobj, gauge_orig=None)
```

Diamagnetic term of magnetizability.

See also J. Olsen et al., Theor. Chem. Acc., 90, 421 (1995)

```python
pyscf.prop.magnetizability.rhf.para(magobj, gauge_orig=None, h1=None, s1=None, with_cphf=None)
```

Paramagnetic susceptibility tensor

Kwarg:

- **h1**: `(3,nmo,nocc)` array
  - First order Fock matrix in MO basis.

- **s1**: `(3,nmo,nocc)` array
  - First order overlap matrix in MO basis.

- **with_cphf** [boolean or function(dm_mo) => v1_mo] If a boolean value is given, the value determines whether CPHF equation will be solved or not. The induced potential will be generated by the function gen_vind. If a function is given, CPHF equation will be solved, and the given function is used to compute induced potential

rks

Non-relativistic magnetizability tensor for DFT

uhf

Non-relativistic magnetizability tensor for UHF (In testing)

```python
pyscf.prop.magnetizability.uhf.para(magobj, gauge_orig=None, h1=None, s1=None, with_cphf=None)
```

Paramagnetic susceptibility tensor

Kwarg:

- **h1**: A list of arrays. Shapes are `[3,nmo_a,nocc_a), (3,nmo_b,nocc_b)]` First order Fock matrices in MO basis.

- **s1**: A list of arrays. Shapes are `[3,nmo_a,nocc_a), (3,nmo_b,nocc_b)]` First order overlap matrices in MO basis.
with_cphf [boolean or function(dm_mo) => v1_mo] If a boolean value is given, the value determines whether CPHF equation will be solved or not. The induced potential will be generated by the function gen_vind. If a function is given, CPHF equation will be solved, and the given function is used to compute induced potential.

uks

Non-relativistic magnetizability tensor for UKS (In testing)

1.45.4 nsr —

Examples
No example

Program reference

rhf

Non-relativistic nuclear spin-rotation tensors for RHF

class pyscf.prop.nsr.rhf.NSR (scf_method)
    Nuclear-spin rotation tensors
        dia(nsrobj, gauge_orig=None, shielding_nuc=None, dm0=None)
            Diamagnetic part of NSR tensors.
        para(nsrobj, mo10=None, mo_coeff=None, mo_occ=None, shielding_nuc=None)
            Paramagnetic part of NSR shielding tensors.

rks

Non-relativistic nuclear spin-rotation tensors for RKS

class pyscf.prop.nsr.rks.NSR (scf_method)
    Nuclear-spin rotation tensors for RKS
        get_fock(nsrobj, dm0=None, gauge_orig=None)
            First order Fock matrix wrt external magnetic field
uhf

Non-relativistic nuclear spin-rotation tensors for UHF
class pyscf.prop.nsr.uhf.NSR(scf_method)
   Nuclear-spin rotation tensors for UHF
dia (nsrobj, gauge_orig=None, shielding_nuc=None, dm0=None)
   Diamagnetic part of NSR tensors.
get_fock (nmrobj, dm0=None, gauge_orig=None)
   First order partial derivatives of Fock matrix wrt external magnetic field. \( \frac{\partial F}{\partial B} \)
para (nsrobj, mo10=None, mo_coeff=None, mo_occ=None, shielding_nuc=None)
   Paramagnetic part of NSR shielding tensors.
solve_mol (nmrobj, mo_energy=None, mo_coeff=None, mo_occ=None, h1=None, s1=None, with_cphf=None)
   Solve the first order equation
   Kwargs:
   with_cphf [boolean or function(dm_mo) => v1_mo] If a boolean value is given, the value determines whether CPHF equation will be solved or not. The induced potential will be generated by the function gen_vind. If a function is given, CPHF equation will be solved, and the given function is used to compute induced potential

uks

Non-relativistic nuclear spin-rotation tensors for UKS
class pyscf.prop.nsr.uks.NSR(scf_method)
   Nuclear-spin rotation tensors for UKS
get_fock (nmrobj, dm0=None, gauge_orig=None)
   First order Fock matrix wrt external magnetic field

1.45.5 rotational_gtensor —

Examples

No example

Program reference

rhf

Non-relativistic rotational g-tensor for RHF

class pyscf.prop.rotational_gtensor.rhf.RotationalGTensor (mf)
    HF rotational g-tensors
    
    dia (magobj, gauge_orig=None)
    Part of rotational g-tensors. It is the direct second derivatives of the Lagrangian (corresponding to the zeroth order wavefunction). Unit hbar/\mu_N is not included. This part may be different to the conventional dia-magnetic contributions of rotational g-tensors.

    para (magobj, gauge_orig=None, h1=None, s1=None, with_cphf=None)
    Part of rotational g-tensors from the first order wavefunctions. Unit hbar/\mu_N is not included. This part may be different to the conventional para-magnetic contributions of rotational g-tensors.

pyscf.prop.rotational_gtensor.rhf.dia (magobj, gauge_orig=None)
    Part of rotational g-tensors. It is the direct second derivatives of the Lagrangian (corresponding to the zeroth order wavefunction). Unit hbar/\mu_N is not included. This part may be different to the conventional dia-magnetic contributions of rotational g-tensors.

pyscf.prop.rotational_gtensor.rhf.inertia_tensor (mol)
    Inertia tensor, mass center

pyscf.prop.rotational_gtensor.rhf.nuc (mol)
    Nuclear contributions

pyscf.prop.rotational_gtensor.rhf.para (magobj, gauge_orig=None, h1=None, s1=None, with_cphf=None)
    Part of rotational g-tensors from the first order wavefunctions. Unit hbar/\mu_N is not included. This part may be different to the conventional para-magnetic contributions of rotational g-tensors.

rks

Non-relativistic rotational g-tensor for DFT

class pyscf.prop.rotational_gtensor.rks.RotationalGTensor (mf)
    Rotational g-tensors for RKS
    
    dia (magobj, gauge_orig=None)
    Part of rotational g-tensors. It is the direct second derivatives of the Lagrangian (corresponding to the zeroth order wavefunction). Unit hbar/\mu_N is not included. This part may be different to the conventional dia-magnetic contributions of rotational g-tensors.

    get_fock (nmrobj, dm0=None, gauge_orig=None)
    First order Fock matrix wrt external magnetic field

pyscf.prop.rotational_gtensor.rks.dia (magobj, gauge_orig=None)
    Part of rotational g-tensors. It is the direct second derivatives of the Lagrangian (corresponding to the zeroth order wavefunction). Unit hbar/\mu_N is not included. This part may be different to the conventional dia-magnetic contributions of rotational g-tensors.

uhf

Non-relativistic rotational g-tensor for UHF

class pyscf.prop.rotational_gtensor.uhf.RotationalGTensor (mf)
    Rotational g-tensors for UHF
**dia** *(magobj, gauge_orig=None)*

Part of rotational g-tensors. It is the direct second derivatives of the Lagrangian (corresponding to the zeroth order wavefunction). Unit $\hbar/\mu_N$ is not included. This part may be different to the conventional diamagnetic contributions of rotational g-tensors.

**get_fock** *(nmrobj, dm0=None, gauge_orig=None)*

First order partial derivatives of Fock matrix wrt external magnetic field, $\frac{\partial F}{\partial B}$

**para** *(magobj, gauge_orig=None, h1=None, s1=None, with_cphf=None)*

Part of rotational g-tensors from the first order wavefunctions. Unit $\hbar/\mu_N$ is not included. This part may be different to the conventional paramagnetic contributions of rotational g-tensors.

**uks**

Non-relativistic rotational g-tensor for UKS

**class** *pyscf.prop.rotational_gtensor.uks.RotationalGTensor* *(mf)*

Rotational g-tensors for UKS

**dia** *(magobj, gauge_orig=None)*

Part of rotational g-tensors. It is the direct second derivatives of the Lagrangian (corresponding to the zeroth order wavefunction). Unit $\hbar/\mu_N$ is not included. This part may be different to the conventional diamagnetic contributions of rotational g-tensors.

**get_fock** *(nmrobj, dm0=None, gauge_orig=None)*

First order Fock matrix wrt external magnetic field

**1.45.6 zfs —**

**Examples**

No example

**Program reference**

**uhf**

Non-relativistic unrestricted Hartree-Fock zero-field splitting (In testing)

**Refs:** JCP, 134, 194113 PRB, 60, 9566 JCP, 127, 164112

---

**1.45. prop — Molecular properties** 409
pyscf.prop.zfs.uhf.ZFS
   alias of ZeroFieldSplitting

class pyscf.prop.zfs.uhf.ZeroFieldSplitting(scf_method)
   dE = 1 dot gtensor dot s

pyscf.prop.zfs.uhf.koseki_charge(z)
   Koseki effective charge in SO correction
      Ref: JPC 96, 10768 JPC, 99, 12764 JPCA, 102, 10430

1.45.7 freq —

The freq module handles vibrational frequencies for mean-field wavefunctions.
To compute the frequency of a water molecule under restricted spin symmetry, first perform a restricted Kohn-Sham calculation using density functional theory module dft, then calculate the frequency using the restricted Kohn-Sham module rks:

```python
from pyscf import gto, dft
from pyscf.prop.freq import rks
mol = gto.M(atom='''
    O 0 0 0
    H 0 -0.757 0.587
    H 0 0.757 0.587''',
    basis='ccpvdz', verbose=4)
mf = dft.RKS(mol).run()
w, modes = rks.Freq(mf).kernel()
```

Examples

Relevant examples examples/prop/01-freq.py

Program reference

rks

See also pyscf/hessian/rks.py

uks

See also pyscf/hessian/uks.py

rhf

See also pyscf/hessian/rhf.py

uhf

See also pyscf/hessian/uhf.py
1.45.8 hfc — Hyperfine coupling tensor

To compute the hyperfine coupling tensor, one could follow the following example:

```python
from pyscf import gto, scf, dft
from pyscf.prop import hfc
mol = gto.M(atom='''
C 0 0 0
N 0 0 1.1747
''',
basis='ccpvdz', spin=1, charge=0, verbose=3)
mf = scf.UHF(mol).run()
gobj = hfc.uhf.HFC(mf).set(verbose=4)
gobj.para_soc2e = 'SSO+SOO'
gobj.so_eff_charge = False
obj.kernel()
```

Further examples can be found here:

Examples

Relevant examples examples/prop/03-hfc.py

Program reference

dhf

Dirac Hartree-Fock hyperfine coupling tensor (In testing)
Refs: JCP, 134, 044111

uhf

Non-relativistic unrestricted Hartree-Fock hyperfine coupling tensor
Refs: JCP, 120, 2127 JCP, 118, 3939

pyscf.prop.hfc.uhf.HFC

class pyscf.prop.hfc.uhf.HyperfineCoupling

def make_fcdip(hfcobj, dm0, hfc_nuc=None, verbose=None)
    The contribution of Fermi-contact term and dipole-dipole interactions

def make_pso_soc(hfcobj, hfc_nuc=None)
    Spin-orbit coupling correction

def make_h1_soc(hfcobj, dm0)
    1-electron and 2-electron spin-orbit coupling integrals.
    1-electron SOC integral is the imaginary part of [i sigma dot pV x p], ie [sigma dot pV x p].
Note sigma_z is considered in the SOC integrals (the (-) sign for beta-beta block is included in the integral). The factor 1/2 in the spin operator $s=sigma/2$ is not included.

```python
def make_pso_soc(hfcobj, hfc_nuc=None):
    Spin-orbit coupling correction
```

def uks:
    Non-relativistic unrestricted Hartree-Fock hyperfine coupling tensor (In testing)

Refs: JCP, 120, 2127 JCP, 118, 3939

1.45.9 nmr — NMR

To compute NMR shielding constants, one could follow the following example:

```python
from pyscf import gto, dft
from pyscf.prop import nmr
mol = gto.M(atom='''
  C 0 0 0
  O 0 0 1.1747
  ''',
    basis='ccpvdz', verbose=3)
mf = dft.RKS(mol)
mf.xc = 'b3lyp'
mf.run()
nmr.RKS(mf).kernel()
```

Examples

Relevant examples :file: examples/prop/04-nmr.py

Program reference

dhf

NMR shielding of Dirac Hartree-Fock

```python
class pyscf.prop.nmr.dhf.NMR(scf_method):
magnetic shielding constants
    get_fock (nmrobj, dm0=None, gauge_orig=None)
    First order Fock matrix wrt external magnetic field. Note the side effects of set_common_origin.
    make_h10 (nmrobj, dm0=None, gauge_orig=None)
    First order Fock matrix wrt external magnetic field. Note the side effects of set_common_origin.
    solve_mol (nmrobj, mo_energy=None, mo_coeff=None, mo_occ=None, h1=None, s1=None, with_cphf=None)
    Solve the first order equation
    Kwargs:
```

412 Chapter 1. Contents
with_cphf [boolean or function(dm_mo) => v1_mo] If a boolean value is given, the value determines whether CPHF equation will be solved or not. The induced potential will be generated by the function gen_vind. If a function is given, CPHF equation will be solved, and the given function is used to compute induced potential

```
with_cphf
```

```
pyscf.prop.nmr.dhf.dia (mol, dm0, gauge_orig=None, shielding_nuc=None, mb='RMB')

   Note the side effects of set_common_origin
```

```
pyscf.prop.nmr.dhf.gen_vind (mf, mo_coeff, mo_occ)

   Induced potential
```

```
pyscf.prop.nmr.dhf.get_fock (nmrobj, dm0=None, gauge_orig=None)

   First order Fock matrix wrt external magnetic field. Note the side effects of set_common_origin.
```

```
pyscf.prop.nmr.dhf.get_ovlp (mol, gauge_orig=None, mb='RMB')

   First order overlap matrix wrt external magnetic field. Note the side effects of set_common_origin.
```

```
pyscf.prop.nmr.dhf.make_h10rkb (mol, dm0, gauge_orig=None, with_gaunt=False, verbose=2)

   Note the side effects of set_common_origin
```

```
pyscf.prop.nmr.dhf.make_h10rmb (mol, dm0, gauge_orig=None, with_gaunt=False, verbose=2)

   Note the side effects of set_common_origin
```

```
pyscf.prop.nmr.dhf.make_s10 (mol, gauge_orig=None, mb='RMB')

   First order overlap matrix wrt external magnetic field.
```

```
pyscf.prop.nmr.dhf.solve_mol (nmrobj, mo_energy=None, mo_coeff=None, mo_occ=None, h1=None, s1=None, with_cphf=None)

   Solve the first order equation

   **Kwargs:**

   ```
   with_cphf [boolean or function(dm_mo) => v1_mo] If a boolean value is given, the value determines whether CPHF equation will be solved or not. The induced potential will be generated by the function gen_vind. If a function is given, CPHF equation will be solved, and the given function is used to compute induced potential
```

```
rhf
```

Non-relativistic NMR shielding tensor

```
pyscf.prop.nmr.rhf.dia (nmrobj, gauge_orig=None, shielding_nuc=None, dm0=None)

   Diamagnetic part of NMR shielding tensors.

   See also J. Olsen et al., Theor. Chem. Acc., 90, 421 (1995)
```

```
pyscf.prop.nmr.rhf.gen_vind (mf, mo_coeff, mo_occ)

   Induced potential
```

```
pyscf.prop.nmr.rhf.get_fock (nmrobj, dm0=None, gauge_orig=None)

   First order partial derivatives of Fock matrix wrt external magnetic field. frac{partial F}{partial B}
```

```
pyscf.prop.nmr.rhf.get_ovlp (mol, gauge_orig=None)

   First order overlap matrix wrt external magnetic field.
```

```
pyscf.prop.nmr.rhf.make_h10 (mol, dm0, gauge_orig=None, verbose=2)

   Imaginary part of first order Fock operator

   Note the side effects of set_common_origin
```

```
pyscf.prop.nmr.rhf.make_s10 (mol, gauge_orig=None)

   First order overlap matrix wrt external magnetic field.
```

1.45. prop — Molecular properties
pyscf.prop.nmr.rhf.\texttt{para}(\texttt{nmrobj}, \texttt{mo10}=\texttt{None}, \texttt{mo\_coeff}=\texttt{None}, \texttt{mo\_occ}=\texttt{None}, \texttt{shielding\_nuc}=\texttt{None})

Paramagnetic part of NMR shielding tensors.

pyscf.prop.nmr.rhf.\texttt{solve\_mol}(\texttt{nmrobj}, \texttt{mo\_energy}=\texttt{None}, \texttt{mo\_coeff}=\texttt{None}, \texttt{mo\_occ}=\texttt{None}, \texttt{h1}=\texttt{None}, \texttt{s1}=\texttt{None}, \texttt{with\_cphf}=\texttt{None})

Solve the first order equation

\textbf{Kwargs:}

\texttt{with\_cphf} [boolean or function(dm\_mo) => v1\_mo] If a boolean value is given, the value determines whether CPHF equation will be solved or not. The induced potential will be generated by the function \texttt{gen\_vind}. If a function is given, CPHF equation will be solved, and the given function is used to compute induced potential

\texttt{rks}

pyscf.prop.nmr.rks.\texttt{get\_fock}(\texttt{nmrobj}, \texttt{dm0}=\texttt{None}, \texttt{gauge\_orig}=\texttt{None})

First order Fock matrix wrt external magnetic field

\texttt{uhf}

Non-relativistic NMR shielding tensor

pyscf.prop.nmr.uhf.\texttt{gen\_vind}(\texttt{mf}, \texttt{mo\_coeff}, \texttt{mo\_occ})

Induced potential

pyscf.prop.nmr.uhf.\texttt{get\_fock}(\texttt{nmrobj}, \texttt{dm0}=\texttt{None}, \texttt{gauge\_orig}=\texttt{None})

First order partial derivatives of Fock matrix wrt external magnetic field. \(\frac{\partial F}{\partial B}\)

pyscf.prop.nmr.uhf.\texttt{solve\_mol}(\texttt{nmrobj}, \texttt{mo\_energy}=\texttt{None}, \texttt{mo\_coeff}=\texttt{None}, \texttt{mo\_occ}=\texttt{None}, \texttt{h1}=\texttt{None}, \texttt{s1}=\texttt{None}, \texttt{with\_cphf}=\texttt{None})

Solve the first order equation

\textbf{Kwargs:}

\texttt{with\_cphf} [boolean or function(dm\_mo) => v1\_mo] If a boolean value is given, the value determines whether CPHF equation will be solved or not. The induced potential will be generated by the function \texttt{gen\_vind}. If a function is given, CPHF equation will be solved, and the given function is used to compute induced potential

\texttt{uks}

pyscf.prop.nmr.uks.\texttt{get\_fock}(\texttt{nmrobj}, \texttt{dm0}=\texttt{None}, \texttt{gauge\_orig}=\texttt{None})

First order Fock matrix wrt external magnetic field

1.45.10 polarizability —

\textbf{Examples}

No example
Program reference

rhf

Non-relativistic static and dynamic polarizability and hyper-polarizability tensor

rks

Non-relativistic static and dynamic polarizability and hyper-polarizability tensor (In testing)

uhf

Non-relativistic static and dynamic polarizability and hyper-polarizability tensor (In testing)

uks

Non-relativistic static and dynamic polarizability and hyper-polarizability tensor (In testing)

1.45.11 ssc — Spin-spin coupling

To compute the nuclear spin-spin coupling constants, one could follow the following example:

```python
from pyscf import gto, scf, dft
from pyscf.prop import ssc
mol = gto.M(atom='''
O 0 0 0
H 0 -0.757 0.587
H 0 0.757 0.587''',
basis='ccpvdz')
mf = scf.UHF(mol).run()
ssc.UHF(mf).kernel()
mf = dft.UKS(mol).set(xc='b3lyp').run()
ssc.UKS(mf).kernel()
```

Examples

Relevant examples examples/prop/05-ssc.py

Program reference

dhf

4-component Dirac-Hartree-Fock spin-spin coupling (SSC) constants

```python
pyscf.prop.ssc.dhf.make_dia (sscobj, mol, dm0, nuc_pair=None, mb='RMB')
nbital diamagnetic term
pyscf.prop.ssc.dhf.sa01sa01_integral(mol, orig1, orig2)

vec[r]_A/r_A^3 times sigma vec[r]_B/r_B^3 times sigma
```
rhf

Non-relativistic RHF spin-spin coupling (SSC) constants
Ref. Chem. Rev., 99, 293 JCP, 113, 3530 JCP, 113, 9402
pyscf.prop.ssc.rhf.\texttt{dso\_integral}(\texttt{mol, orig1, orig2})
Integral of $\vec{r}\vec{r}/(|\vec{r}-\text{orig1}|^3 |\vec{r}-\text{orig2}|^3)$ Ref. JCP, 73, 5718
pyscf.prop.ssc.rhf.\texttt{gen\_vind}(\texttt{mf, mo\_coeff, mo\_occ})
Induced potential associated with h1_PSO
pyscf.prop.ssc.rhf.\texttt{make\_dso}(\texttt{sscobj, mol, dm0, nuc\_pair=None})
orbital diamagnetic term
pyscf.prop.ssc.rhf.\texttt{make\_fc}(\texttt{sscobj, nuc\_pair=None})
Only Fermi-contact
pyscf.prop.ssc.rhf.\texttt{make\_fcsd}(\texttt{sscobj, nuc\_pair=None})
FC + SD contributions to 2nd order energy
pyscf.prop.ssc.rhf.\texttt{make\_hl\_fcsd}(\texttt{mol, mo\_coeff, mo\_occ, atmlst})
MO integrals for FC + SD

rks

Non-relativistic RKS spin-spin coupling (SSC) constants

uhf

Non-relativistic UHF spin-spin coupling (SSC) constants
pyscf.prop.ssc.uhf.\texttt{gen\_vind}(\texttt{mf, mo\_coeff, mo\_occ})
Induced potential associated with h1_PSO
pyscf.prop.ssc.uhf.\texttt{make\_dso}(\texttt{sscobj, mol, dm0, nuc\_pair=None})
orbital diamagnetic term
pyscf.prop.ssc.uhf.\texttt{make\_fc}(\texttt{sscobj, nuc\_pair=None})
Only Fermi-contact
pyscf.prop.ssc.uhf.\texttt{make\_fcsd}(\texttt{sscobj, nuc\_pair=None})
FC + SD contributions to 2nd order energy
pyscf.prop.ssc.uhf.\texttt{make\_hl\_fcsd}(\texttt{mol, mo\_coeff, mo\_occ, atmlst})
FC + SD

uks

Non-relativistic UKS spin-spin coupling (SSC) constants (In testing)

1.46 data — Data for constants

Data for constants. Physical constants and other useful data
1.47 shciscf — Semistochastic Heat Bath Configuration Interaction for SCF

1.47.1 Examples

For a complete list of FCI examples, see pyscf/examples/shciscf

1.47.2 Program reference

shci

You can also download the PDF version of this manual.
a
 ao2mo, 89

cc, 267
 ccn, 374
 ci, 263
 cornell_shci, 290

d
 df, 100
 dft, 213
 doci, 267

g
 grad, 240
 gw, 275

l
 lib, 386
 lib.chkfile, 394
 lib.d募集资金, 396
 lib.linalg_helper, 389
 lib.misc, 399
 lib.numpy_helper, 386
 lib.parameters, 386

p
 pbc.df, 341
 pyscf.a02mo, 89
 pyscf.a02mo.addons, 99
 pyscf.a02mo.incore, 90
 pyscf.a02mo.outcore, 93
 pyscf.a02mo.semi_incore, 92
 pyscf.cc.addons, 274
 pyscf.cc.ccsd, 271
 pyscf.cc.ccsd_grad, 274
 pyscf.cc.ccsd_t, 274
 pyscf.cc.rccsd, 273
 pyscf.cc.uccsd, 274
 pyscf.ccn, 375
 pyscf.ccn.cc, 375
 pyscf.ccn.equations, 377
 pyscf.ci.addons, 265
 pyscf.ci.cisd, 263
 pyscf.ci.gcisd, 265
 pyscf.ci.ucisd, 266
 pyscf.data, 413
 pyscf.df.addons, 106
 pyscf.df.df, 106
 pyscf.df.df_jk, 107
 pyscf.df.grad.rhf, 107
 pyscf.df.grad.rks, 108
 pyscf.df.grad.uhf, 108
 pyscf.df.grad.unks, 108
 pyscf.df.hessian.rhf, 108
 pyscf.df.hessian.rks, 108
 pyscf.df.hessian.uhf, 108
 pyscf.df.hessian.unks, 108
 pyscf.df.incore, 105
 pyscf.df.outcore, 105
 pyscf.df.r_incore, 107
 pyscf.dft.dks, 225
 pyscf.dft.gen_grid, 226
 pyscf.dft.gen_xcfun_param, 221
 pyscf.dft.gks, 219
 pyscf.dft.gks_symm, 219
 pyscf.dft.libxc, 221
 pyscf.dft.numint, 231
 pyscf.dft.radi, 226
 pyscf.dft.rks, 214
 pyscf.dft.rks_symm, 220
 pyscf.dft.unks, 219
 pyscf.dft.unks_symm, 221
 pyscf.dft.vxc, 237
 pyscf.dft.xc, 239
 pyscf.dft.xcfun, 237
 pyscf.dftd3, 240
 pyscf.dftd3.itrf, 240
 pyscf.dmrgscf.chemps2, 313
 pyscf.dmrgscf.dmrg_sym, 313
 pyscf.dmrgscf.dmrgci, 313
 pyscf.doci, 267
 pyscf.doci.doci_mcscf, 267
<table>
<thead>
<tr>
<th>Module Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>pyscf.prop.magnetizability.rhf</td>
<td>401</td>
</tr>
<tr>
<td>pyscf.prop.magnetizability.rks</td>
<td>402</td>
</tr>
<tr>
<td>pyscf.prop.magnetizability.uhf</td>
<td>402</td>
</tr>
<tr>
<td>pyscf.prop.magnetizability.uks</td>
<td>402</td>
</tr>
<tr>
<td>pyscf.prop.nmr</td>
<td>409</td>
</tr>
<tr>
<td>pyscf.prop.nmr.dhf</td>
<td>409</td>
</tr>
<tr>
<td>pyscf.prop.nmr.rhf</td>
<td>410</td>
</tr>
<tr>
<td>pyscf.prop.nmr.rks</td>
<td>411</td>
</tr>
<tr>
<td>pyscf.prop.nmr.uhf</td>
<td>411</td>
</tr>
<tr>
<td>pyscf.prop.nmr.uksf</td>
<td>411</td>
</tr>
<tr>
<td>pyscf.prop.nsr</td>
<td>403</td>
</tr>
<tr>
<td>pyscf.prop.nsr.rhf</td>
<td>403</td>
</tr>
<tr>
<td>pyscf.prop.nsr.rks</td>
<td>403</td>
</tr>
<tr>
<td>pyscf.prop.nsr.uhf</td>
<td>403</td>
</tr>
<tr>
<td>pyscf.prop.nsr.uxf</td>
<td>404</td>
</tr>
<tr>
<td>pyscf.prop.nsr.uxs</td>
<td>404</td>
</tr>
<tr>
<td>pyscf.prop.polarizability</td>
<td>411</td>
</tr>
<tr>
<td>pyscf.prop.polarizability.rhf</td>
<td>411</td>
</tr>
<tr>
<td>pyscf.prop.polarizability.rks</td>
<td>411</td>
</tr>
<tr>
<td>pyscf.prop.polarizability.uhf</td>
<td>411</td>
</tr>
<tr>
<td>pyscf.prop.polarizability.uks</td>
<td>411</td>
</tr>
<tr>
<td>pyscf.prop.rotational_gtensor.rhf</td>
<td>404</td>
</tr>
<tr>
<td>pyscf.prop.rotational_gtensor.rks</td>
<td>405</td>
</tr>
<tr>
<td>pyscf.prop.rotational_gtensor.uhf</td>
<td>405</td>
</tr>
<tr>
<td>pyscf.prop.rotational_gtensor_uks</td>
<td>405</td>
</tr>
<tr>
<td>pyscf.prop.scc</td>
<td>412</td>
</tr>
<tr>
<td>pyscf.prop.scc.dhf</td>
<td>412</td>
</tr>
<tr>
<td>pyscf.prop.scc.rhf</td>
<td>413</td>
</tr>
<tr>
<td>pyscf.prop.scc.rks</td>
<td>413</td>
</tr>
<tr>
<td>pyscf.prop.scc.uhf</td>
<td>413</td>
</tr>
<tr>
<td>pyscf.prop.scc.uks</td>
<td>413</td>
</tr>
<tr>
<td>pyscf.prop.zfs</td>
<td>406</td>
</tr>
<tr>
<td>pyscf.prop.zfs.uhf</td>
<td>406</td>
</tr>
<tr>
<td>pyscf.gmmmm.itrf</td>
<td>355</td>
</tr>
<tr>
<td>pyscf.scf</td>
<td>119</td>
</tr>
<tr>
<td>pyscf.scf._vhf</td>
<td>203</td>
</tr>
<tr>
<td>pyscf.scf.addos</td>
<td>207</td>
</tr>
<tr>
<td>pyscf.scf.atom_hf</td>
<td>186</td>
</tr>
<tr>
<td>pyscf.scf.chkfile</td>
<td>209</td>
</tr>
<tr>
<td>pyscf.scf.cphf</td>
<td>186</td>
</tr>
<tr>
<td>pyscf.scf.dhf</td>
<td>203</td>
</tr>
<tr>
<td>pyscf.scf.dls</td>
<td>209</td>
</tr>
<tr>
<td>pyscf.scf.ghf</td>
<td>187</td>
</tr>
<tr>
<td>pyscf.scf.ghf_symm</td>
<td>193</td>
</tr>
<tr>
<td>pyscf.scf.hf</td>
<td>148</td>
</tr>
<tr>
<td>pyscf.scf.hf_symm</td>
<td>179</td>
</tr>
<tr>
<td>pyscf.scf.jk</td>
<td>209</td>
</tr>
<tr>
<td>pyscf.scf.newton_ah</td>
<td>212</td>
</tr>
<tr>
<td>pyscf.scf.rohf</td>
<td>195</td>
</tr>
<tr>
<td>pyscf.scf.stability</td>
<td>212</td>
</tr>
<tr>
<td>pyscf.scf.stability_slow</td>
<td>213</td>
</tr>
<tr>
<td>pyscf.scf.ucphf</td>
<td>203</td>
</tr>
<tr>
<td>pyscf.scf.uhf</td>
<td>171</td>
</tr>
<tr>
<td>pyscf.scf.uhf_symm</td>
<td>184</td>
</tr>
<tr>
<td>pyscf.scf.x2c</td>
<td>207</td>
</tr>
<tr>
<td>pyscf.semiempirical</td>
<td>359</td>
</tr>
<tr>
<td>pyscf.semiempirical.mindo3</td>
<td>359</td>
</tr>
<tr>
<td>pyscf.semiempirical.mrdindo3_grad</td>
<td>361</td>
</tr>
<tr>
<td>pyscf.semiempirical.umrdindo3_grad</td>
<td>361</td>
</tr>
<tr>
<td>pyscf.sgx</td>
<td>251</td>
</tr>
<tr>
<td>pyscf.sgx.sgx</td>
<td>251</td>
</tr>
<tr>
<td>pyscf.sgx.sgx_symm</td>
<td>251</td>
</tr>
<tr>
<td>pyscf.shcisf</td>
<td>414</td>
</tr>
<tr>
<td>pyscf.solvent</td>
<td>357</td>
</tr>
<tr>
<td>pyscf.solvent.ddcosmo</td>
<td>357</td>
</tr>
<tr>
<td>pyscf.solvent.ddcosmo_grad</td>
<td>359</td>
</tr>
<tr>
<td>pyscf.solvent.ddpcm</td>
<td>357</td>
</tr>
<tr>
<td>pyscf.soscf</td>
<td>250</td>
</tr>
<tr>
<td>pyscf.soscf.ciah</td>
<td>251</td>
</tr>
<tr>
<td>pyscf.soscf.newton_ah</td>
<td>250</td>
</tr>
<tr>
<td>pyscf.symm</td>
<td>109</td>
</tr>
<tr>
<td>pyscf.symm.addos</td>
<td>111</td>
</tr>
<tr>
<td>pyscf.symm.basis</td>
<td>111</td>
</tr>
<tr>
<td>pyscf.symm.cg</td>
<td>114</td>
</tr>
<tr>
<td>pyscf.symm.Dmatrix</td>
<td>115</td>
</tr>
<tr>
<td>pyscf.symm.geom</td>
<td>111</td>
</tr>
<tr>
<td>pyscf.symm.param</td>
<td>114</td>
</tr>
<tr>
<td>pyscf.symm.ph</td>
<td>114</td>
</tr>
<tr>
<td>pyscf.tdscf</td>
<td>252</td>
</tr>
<tr>
<td>pyscf.tdscf.common_slow</td>
<td>257</td>
</tr>
<tr>
<td>pyscf.tdscf.proxy</td>
<td>259</td>
</tr>
<tr>
<td>pyscf.tdscf.rhf</td>
<td>252</td>
</tr>
<tr>
<td>pyscf.tdscf.rhf_slow</td>
<td>259</td>
</tr>
<tr>
<td>pyscf.tdscf.rks</td>
<td>256</td>
</tr>
<tr>
<td>pyscf.tdscf.uhf</td>
<td>255</td>
</tr>
<tr>
<td>pyscf.tdscf_uks</td>
<td>256</td>
</tr>
<tr>
<td>pyscf.tools</td>
<td>366</td>
</tr>
<tr>
<td>pyscf.tools.c60struct</td>
<td>374</td>
</tr>
<tr>
<td>pyscf.tools.chgcar</td>
<td>371</td>
</tr>
<tr>
<td>pyscf.tools.chkfile_util</td>
<td>372</td>
</tr>
<tr>
<td>pyscf.tools.cubegen</td>
<td>367</td>
</tr>
<tr>
<td>pyscf.tools.dump_mat</td>
<td>369</td>
</tr>
<tr>
<td>pyscf.tools.fcidump</td>
<td>366</td>
</tr>
<tr>
<td>pyscf.tools.mo_mapping</td>
<td>372</td>
</tr>
<tr>
<td>pyscf.tools.molden</td>
<td>367</td>
</tr>
<tr>
<td>pyscf.tools.ring</td>
<td>374</td>
</tr>
<tr>
<td>pyscf.tools.wfn_format</td>
<td>367</td>
</tr>
<tr>
<td>pyscf.x2c</td>
<td>316</td>
</tr>
<tr>
<td>pyscf.x2c.sfx2cie</td>
<td>316</td>
</tr>
<tr>
<td>pyscf.x2c.sfx2cie_grad</td>
<td>317</td>
</tr>
<tr>
<td>pyscf.x2c.sfx2cie_hess</td>
<td>317</td>
</tr>
<tr>
<td>pyscf.x2c.x2c</td>
<td>316</td>
</tr>
</tbody>
</table>

S

scc, 116
semiempirical, 359
sgx, 251
solvent, 356
soscf, 250

t

tools, 366