Density functional theory (DFT)

Modules: `pyscf.dft` [../pyscf_api_docs/pyscf.dft.html#module-pyscf.dft], `pyscf.pbc.dft` [../pyscf_api_docs/pyscf.pbc.dft.html#module-pyscf.pbc.dft]

Introduction

Kohn-Sham density functional theory (KS-DFT) has been implemented through derived classes of the `pyscf.scf.hf.SCF` [../pyscf_api_docs/pyscf.scf.html#pyscf.scf.hf.SCF] parent class. As such, the methods and capabilities introduced in Self-consistent field (SCF) methods [scf.html#user-scf] are also available to the `dft` module, e.g., the efficient second-order Newton-Raphson algorithm.

A minimal example of using the `dft` module reads, cf. dft/22-newton.py [https://github.com/pyscf/pyscf/blob/master/examples/dft/22-newton.py]:

```
>>> from pyscf import gto, dft
>>> mol_hf = gto.M(atom = 'H 0 0 0; F 0 0 1.1', basis = 'ccpvdz',
symmetry = True)
>>> mf_hf = dft.RKS(mol_hf)
>>> mf_hf.xc = 'lda,vwn' # default
>>> mf_hf = mf_hf.newton() # second-order algortihm
>>> mf_hf.kernel()
```

This will run a restricted, closed-shell Kohn-Sham DFT calculation with the default LDA functional.

Theory

In KS-DFT, as first proposed by Kohn and Sham [18 [reference.html#id27]], the electron density of a reference noninteracting system is used to represent the density of the true interacting system. As a result, the computational formulation of KS-DFT resembles that of Hartree-Fock (HF) theory, but with a different effective Fock potential. In KS-DFT, the total electronic energy is defined as follows:

$$E = T_s + E_{\text{ext}} + E_J + E_{\text{xc}} \, .$$

Here, T_s is the noninteracting kinetic energy, E_{ext} is the energy due to the external potential, E_J is the Coulomb energy, and E_{xc} is the exchange-correlation (*xc*) energy. In practice, E_{xc} is approximated by a density functional approximation, which themselves may be divided into several classes along different rungs of Jacob's ladder [19 [reference.html#id28]]:

- local density approximations (e.g. LDA; *xc* energy depends only on the electron density, ρ),
- generalized gradient approximations (GGA; *xc* energy also depends on the density gradient, $|\nabla\rho|$),
- meta-GGAs (*xc* energy also depends on the kinetic energy density and/or the density Laplacian, $\sum_i |\nabla\psi_i|^2$, $\nabla^2\rho$; the latter is not supported in PySCF at the moment),
- non-local correlation functionals (*xc* energy involves a double integral)
- hybrid density functionals (a fraction of exact exchange is used), and
- long-range corrected density functionals (exact exchange is used with a modified interaction kernel)

Variationally minimizing the total energy with respect to the density yields the KS equations for the non-interacting reference orbitals, on par with HF theory, and these have the same general form as the Fock equations in Self-consistent field (SCF) methods [scf.html#user-scf]. However, the exact exchange, $\hat{K}$, is replaced by the *xc* potential, $\hat{v}_{\text{xc}} = \delta E_{\text{xc}}/\delta\rho$. For hybrid and meta-GGA calculations, PySCF uses the generalized KS formalism [20 [reference.html#id33]], in which the so-called

generalized KS equations minimize the total energy with respect to the orbitals themselves.

Predefined *xc* Functionals and Functional Aliases

The choice of *xc* functional is assigned via the attribute `DFT.xc`. This is a comma separated string (precise grammar discussed below [#user-dft-custom-func], e.g., `xc = 'pbe,pbe'` denotes PBE exchange plus PBE correlation. In common usage, a single name (alias) is often used to refer to the combination of a particular exchange and correlation approximation instead. To support this, PySCF will first examine a lookup table to see if `DFT.xc` corresponds to a common compound name, and if so, the implementation dispatches to the appropriate exchange and correlation forms, e.g., `xc = 'pbe'` directly translates to `xc = 'pbe,pbe'`. However, if the name is not found in the compound functional table, and only a single string is given, it will be treated as an exchange functional only, e.g., `xc = 'b86'` leads to B86 exchange only (without correlation). Please note that earlier PySCF versions (1.5.0 or earlier) did not support compound functional aliases, and both exchange and correlation always had to be explicitly assigned.

PySCF supports two independent libraries of *xc* functional implementations, namely Libxc [https://www.tddft.org/programs/libxc/] and XCFun [https://xcfun.readthedocs.io/en/latest/]. The former of these is the default, but the latter may be selected by setting `DFT._numint.libxc = dft.xcfun`, cf. dft/32-xcfun_as_default.py [https://github.com/pyscf/pyscf/blob/master/examples/dft/32-xcfun_as_default.py]. For complete lists of the available density functional approximations, the user is referred to the `XC_CODES` dictionaries in pyscf/dft/libxc.py [https://github.com/pyscf/pyscf/blob/master/pyscf/dft/libxc.py] and pyscf/dft/xcfun.py [https://github.com/pyscf/pyscf/blob/master/pyscf/dft/xcfun.py], respectively. The user can choose the library at runtime in order to leverage any of its exclusive features.

The constant maintenance and development of density functional libraries is hard work at little personal benefit, while everyone benefits from having a huge variety of density functionals in numerically stable form for use in applications. If you use Libxc in your calculations, please cite the most up-to-date work on Libxc in your paper. You can see the most up-to-date citation on the Libxc web page [https://www.tddft.org/programs/libxc/]; at the moment, this is [21 [reference.html#id29]]. Likewise, if you use XCFun in your calculations, please cite the most up-to-date work on XCFun in your paper. You can find recent citations on the XCFun web page [https://github.com/dftlibs/xcfun/]; at present, this is [22 [reference.html#id30]]. Please check your log files for the library used in your calculation (you may need to increase the `DFT.verbose` setting of your calculation to see this).

Customizing *xc* functionals

The *xc* functional of choice can be customized. The simplest way to customize to achieve this is to assign a string expression to the `DFT.xc` attribute:

```
>>> HF_X, LDA_X = .6, .08
>>> B88_X = 1. - HF_X - LDA_X
>>> LYP_C = .81
>>> VWN_C = 1. - LYP_C
>>> mf_hf.xc = f'{HF_X:} * HF + {LDA_X:} * LDA + {B88_X:} * B88,
{LYP_C:} * LYP + {VWN_C:} * VWN'
>>> mf_hf.kernel()
>>> mf_hf.xc = 'hf'
>>> mf_hf.kernel()
```

The XC functional string is parsed against a set of 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 a `,`. The first part describes the exchange functional, the second part sets the correlation functional (as for aliases [#user-dft-predef-func])

  - If a `","` does not appear in the string, the entire string is treated as the name of a compound functional (containing both the exchange and the correlation functional) which should be in the list of functional aliases. Again, if the string is not found in the aliased functional list, it is treated as an exchange functional
  - To input only an exchange functional (without a correlation functional), one should leave the second part blank. E.g., `slater,` implies a functional with the LDA contribution only
  - Correspondingly, to neglect the contribution of the exchange functional (i.e. to just use a correlation functional), one should leave the first part blank, e.g., `',vwn'` means a functional with VWN only
  - If a compound *xc* functional is specified, no matter whether it is in the exchange part (the string in front of the comma) or the correlation part (the string behind the comma), both exchange and correlation functionals of the compound *xc* functional will be used
- The functional name can be placed in an arbitrary order. Two names need be separated by operators `+` or `-`. Blank spaces are ignored. NOTE the parser only reads the operators `+, -, *`, while `/` is not supported
- A functional name can have at most one factor. If a factor is not given, it is set to `1`. Compound functionals can be scaled as a unit. For example, `.5 * b3lyp` is equivalent to `.1 * HF + .04 * LDA + .36 * B88, .405 * LYP + .095 * VWN`
- The string `HF` stands for exact exchange (HF K matrix). `HF` can be put in the correlation functional part (after the comma). Putting `HF` in the correlation part is the same as putting `HF` in the exchange part
- The special string `RSH` means a range-separated operator. Its format is `RSH(omega, alpha, beta)`. Another way to input range separation is to use keywords `SR_HF` and `LR_HF`, e.g., `SR_HF(.1) * alpha_plus_beta` and `LR_HF(.1) * alpha` where the number in the parenthesis is the value of `omega`
- The `RSH` kernel in PySCF is based on the error function kernel; Yukawa kernels are not supported at present

- One need in general be careful with the Libxc convention of GGA functionals, in which the LDA contribution is included

For completeness, it's worth mentioning that yet another way to customize *xc* functionals exists, which uses the **`eval_xc()`** method of the numerical integral class:

```
>>> def eval_xc(xc_code, rho, spin=0, relativity=0, deriv=1,
verbose=None):
>>>     # A fictitious functional to demonstrate the usage
>>>     rho0, dx, dy, dz = rho
>>>     gamma = (dx ** 2 + dy ** 2 + dz ** 2)
>>>     exc = .01 * rho0 ** 2 + .02 * (gamma + .001) ** .5
>>>     vrho = .01 * 2 * rho0
>>>     vgamma = .02 * .5 * (gamma + .001) ** (-.5)
>>>     vlapl = None
>>>     vtau = None
>>>     vxc = (vrho, vgamma, vlapl, vtau)
>>>     fxc = None # 2nd-order functional derivative
>>>     kxc = None # 3rd-order functional derivative
>>> return exc, vxc, fxc, kxc
>>> dft.libxc.define_xc_(mf_hf._numint, eval_xc, xctype='GGA')
>>> mf_hf.kernel()
```

By calling the **`dft.libxc.define_xc_()`** function, the customized **`eval_xc()`** function is patched to the numerical integration class `DFT._numint` dynamically.

For more examples of DFT *xc* functional customization, cf. dft/24-custom_xc_functional.py [https://github.com/pyscf/pyscf/blob/master/examples/dft/24-custom_xc_functional.py] and dft/24-define_xc_functional.py [https://github.com/pyscf/pyscf/blob/master/examples/dft/24-define_xc_functional.py].

Numerical integration grids

PySCF implements several numerical integration grids, which can be tuned in KS-DFT calculations following the examples in dft/11-grid_scheme.py

[https://github.com/pyscf/pyscf/blob/master/examples/dft/11-grid_scheme.py]. For instance, predefined grids (identical to those used in TURBOMOLE [https://www.turbomole.org/]) may be set by using levels from `0` (very sparse) to `9` (very dense), with a default values of `3`, cf. pyscf/dft/gen_grid.py [https://github.com/pyscf/pyscf/blob/master/pyscf/dft/gen_grid.py] for more details. Likewise, the default integration grids use Bragg radii for atoms, Treutler-Ahlrichs radial grids, Becke partitioning for grid weights, the pruning scheme of NWChem, and mesh grids, which are all setting that may be overwritten:

```
>>> mf_hf.grids.level = 5
>>> mf_hf.radi_method = dft.gauss_chebeshev
>>> mf_hf.grids.prune = None # disabling pruning of grids near core
regions
```

In addition, these grids can be used for the general numerical evaluation of basis functions, electron densities, and integrals. Some examples of these functionalities can be found in dft/30-ao_value_on_grid.py [https://github.com/pyscf/pyscf/blob/master/examples/dft/30-ao_value_on_grid.py] and dft/31-xc_value_on_grid.py [https://github.com/pyscf/pyscf/blob/master/examples/dft/31-xc_value_on_grid.py]. For instance, the electron density may be readily obtained:

```
>>> mf_hf.xc = 'b3lyp'
>>> coords = mf_hf.grids.coords
>>> weights = mf_hf.grids.weights
>>> ao_value = numint.eval_ao(mol_hf, coords, deriv=1) # AO value and
its gradients
>>> rho = numint.eval_rho(mol_hf, ao_value, dm, xctype='GGA') #
density & density gradients
```

From `rho`, the energy density and *xc* potential can be computed by calling into **`dft.libxc.eval_xc()`**.

A more specialized example is the following on computing the kinetic energy from the nonnegative kinetic energy density according to the formulas:

$$t_s(\mathbf{r}) = \frac{1}{2} \sum_{i \in occ} |\nabla \psi_i(\mathbf{r})|^2 ,$$

$$T_s = \int d\mathbf{r} t_s(\mathbf{r}) .$$

In PySCF, the code boils down to:

```
>>> import numpy as np
>>> occ_orbs = mf_hf.mo_coeff[:, mf_hf.mo_occ > 0.]
>>> grids = dft.gen_grid.Grids(mol_hf)
>>> grids.build(with_non0tab=True)
>>> weights = grids.weights
>>> ao1 = dft.numint.eval_ao(mol_hf, grids.coords, deriv=1,
non0tab=grids.non0tab)
>>> ts = 0.5 * np.einsum('xgp,pi,xgq,qi->g', ao1[1:], occ_orbs,
ao1[1:], occ_orbs)
>>> Ts = np.einsum('g,g->', weights, ts)
```

or - as an alternative - the same may be achieved in the following way:

```
>>> Ts_ao = mol_hf.intor('int1e_kin')
>>> Ts_analyt = np.einsum('ui,uv,vi->', occ_orbs, Ts_ao, occ_orbs)
```

Dispersion corrections

Two main ways exist for adding dispersion (van der Waals) corrections to KS-DFT calculations. One is to augment mean-field results by Grimme's D3 corrections [23 [reference.html#id31]], which can be added through an interface to the external library libdftd3 [https://github.com/cuanto/libdftd3], cf. dftd3/00-hf_with_dftd3.py [https://github.com/pyscf/pyscf/blob/master/examples/dftd3/00-hf_with_dftd3.py]:

```
>>> from pyscf import dftd3
>>> mf_hf_d3 = dftd3.dftd3(dft.RKS(mol_hf))
>>> mf_hf_d3.kernel()
```

Alternatively, non-local correlation may be added through the VV10 functional [24 [reference.html#id32]], cf. dft/33-nlc_functionals.py [https://github.com/pyscf/pyscf/blob/master/examples/dft/33-nlc_functionals.py]:

```
>>> mf_hf.xc = 'wb97m_v'
>>> mf_hf.nlc = 'vv10'
>>> mf_hf.grids.atom_grid = {'H': (99, 590), 'F': (99, 590)}
>>> mf_hf.grids.prune = None
>>> mf_hf.nlcgrids.atom_grid = {'H': (50, 194), 'F': (50, 194)}
>>> mf_hf.nlcgrids.prune = dft.gen_grid.sg1_prune
>>> mf_hf.kernel()
```

It's important to keep in mind that the evaluation of the VV10 functional involves a double grid integration, so differences in grid size can make an enormous difference in time.

Generalized KS and collinearity

When the Hamiltonian does not commute with $\hat{S}_z$, e.g. in the presence of spin-orbit coupling, generalized Kohn-Sham theory (GKS) can be invoked by `mf = dft.GKS(mol)`, cf. generalized calculations [scf.html#user-scf-restrict] and examples/dft/02-gks.py [https://github.com/pyscf/pyscf/tree/master/examples/dft/02-gks.py]. A molecular orbital from GKS may contain both spin-up and spin-down components. As a result, the spin magnetization vector may no longer be in the collinear form $\mathbf{m} = (0, 0, m_z)$ that an unrestricted calculation yields and which widely used collinear *xc* functionals assume. To handle any spin configuration, PySCF supports the non-collinear *xc* functionals from mcfun [https://github.com/Multi-collinear/MCfun] [25 [reference.html#id85]] via setting the DFT attribute `collinear = 'mcol'`, cf. examples/dft/14-collinear_gks.py [https://github.com/pyscf/pyscf/tree/master/examples/dft/14-collinear_gks.py]. Such a non-collinear functional generalizes the widely used collinear functionals that depend on m_z to depend on $\mathbf{m}$. It preserves the invariance with respect to global rotations while maintaining the sensitivity to local spin rotations. It

additionally has the advantage of well-defined functional derivatives, satisfying no net torque globally from the self-consistent xc magnetic field and retaining the local torque crucial to spin dynamics. For LDA functional, PySCF also implemented a non-collinear version that is accessible via `collinear = 'ncol'`.

Periodic Boundary Conditions

Besides finite-sized systems, PySCF further supports KS-DFT calculations with PBCs for performing solid-state calculations. The APIs for molecular and crystalline KS-DFT calculations have deliberately been made to align to the greatest extent possible, and an all-electron KS-DFT calculation for an initialized `Cell` object at either the Γ-point or with k-point sampling may be run through **`dft`** and **`pbc.dft`**, respectively. For more details on PBC functionalities, please see the dedicated sections on PBC-KS-DFT [pbc.html#user-pbc].