Quickstart

This quickstart provides a brief introduction to the use of PySCF in common quantum chemical simulations. These make reference to specific examples within the dedicated examples [https://github.com/pyscf/pyscf/tree/master/examples] directory. For brevity, and so as to not repeat a number of function calls, please note that the cells below often share objects in-between one another. The selection below is far from exhaustive: additional details on the various modules are presented in the accompanying user guide and within the examples directory.

Input Parsing

Molecules (or unit cells, cf. the periodic section) can be created using the convenient shortcut function `Mole.build()` as in (gto/00-input_mole.py [https://github.com/pyscf/pyscf/blob/master/examples/gto/00-input_mole.py]):

```
>>> mol_h2o = gto.M(atom = 'O 0 0 0; H 0 1 0; H 0 0 1', basis =
'ccpvdz')
```

Whenever you change the value of the attributes of **`Mole`**, you'll need to call `build()` again to refresh the internal data of the object.

Symmetry may be specified in the `Mole.symmetry` attribute as either `True` or `False` (default is `False`, i.e., off). Alternatively, a particular subgroup can be specified by a string argument (gto/13-symmetry.py [https://github.com/pyscf/pyscf/blob/master/examples/gto/13-symmetry.py]):

```
>>> mol_c2 = gto.M(atom = 'C 0 0 .625; C 0 0 -.625', symmetry = 'd2h')
```

There are many other ways to input a molecular or crystalline geometry (e.g., by means of Z-matrices or reading in .xyz files), cf. the complete suite of gto [https://github.com/pyscf/pyscf/blob/master/examples/gto] and pbc [https://github.com/pyscf/pyscf/blob/master/examples/pbc] examples.

Mean-Field Theory

Hartree-Fock

A mean-field (e.g., Hartree-Fock) calculation on the H_2O geometry of the above [#input] section can be performed by (cf. scf/00-simple_hf.py [https://github.com/pyscf/pyscf/blob/master/examples/scf/00-simple_hf.py]):

```
>>> from pyscf import scf
>>> rhf_h2o = scf.RHF(mol_h2o)
>>> e_h2o = rhf_h2o.kernel()
```

Besides the final converged ground-state energy, the mean-field object stores the accompanying MO coefficients, occupations, etc. To illustrate how open-shell, possibly spin-polarized calculations are performed, different Hartree-Fock simulations of the O_2 dimer - with its triplet ground state - can be setup as (cf. scf/02-rohf_uhf.py [https://github.com/pyscf/pyscf/blob/master/examples/scf/02-rohf_uhf.py]):

```
>>> mol_o2 = gto.M(atom='O 0 0 0; O 0 0 1.2', spin=2) # (n+2 alpha, n
beta) electrons
>>> uhf_o2 = scf.UHF(mol_o2)
>>> uhf_o2.kernel()
>>> rohf_o2 = scf.ROHF(mol_o2)
>>> rohf_o2.kernel()
```

Finally, a second-order, Newton-Raphson SCF method is available, (which is also compatible with most but not all XC functionals, cf. below [#ksdft]). This requires a set of orthonormal orbitals and their corresponding occupancies as initial guesses.

These are automatically generated if none are provided (cf. scf/22-newton.py [https://github.com/pyscf/pyscf/blob/master/examples/scf/22-newton.py]):

```
>>> rhf_h2o = rhf_h2o.newton()
>>> e_h2o = rhf_h2o.kernel()
```

Kohn-Sham Density Functional Theory

Running a KS-DFT calculation is as straightforward as the above [#hf] HF counterpart (cf. dft/00-simple_dft.py [https://github.com/pyscf/pyscf/blob/master/examples/dft/00-simple_dft.py]):

```
>>> from pyscf import dft
>>> rks_h2o = dft.RKS(mol) # likewise for UKS and ROKS
>>> rks_h2o.xc = 'b3lyp'
```

Besides the use of predefined XC functionals (cf. 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] for the complete lists of available functionals), these can also be fully defined by the user (dft/24-custom_xc_functional.py [https://github.com/pyscf/pyscf/blob/master/examples/dft/24-custom_xc_functional.py]), as can the angular and radial grids used in the calculation (dft/11-grid_scheme.py [https://github.com/pyscf/pyscf/blob/master/examples/dft/11-grid_scheme.py]):

```
>>> rks_h2o.xc = '.2 * HF + .08 * LDA + .72 * B88, .81 * LYP + .19 *
VWN' # B3LYP
>>> rks_h2o.grids.atom_grid = (100, 770)
>>> rks_h2o.grids.prune = None
>>> e_rks = rks_h2o.kernel()
```

Combining dense and sparse grids is important whenever XC functionals with non-local dispersion (van der Waals) corrections are employed (cf. dft/33-

nlc_functionals.py [https://github.com/pyscf/pyscf/blob/master/examples/dft/33-nlc_functionals.py]):

```
>>> rks_c2 = dft.RKS(mol_c2)
>>> rks_c2.xc = 'wb97m_v'
>>> rks_c2.nlc = 'vv10'
>>> rks_c2.grids.atom_grid = (99,590)
>>> rks_c2.grids.prune = None
>>> rks_c2.nlcgrids.atom_grid = (50,194)
>>> rks_c2.nlcgrids.prune = dft.gen_grid.sg1_prune
```

Time-Dependent Mean-Field Theory

Linear response theory is available for both HF and KS-DFT (cf. tddft/00-simple_tddft.py [https://github.com/pyscf/pyscf/blob/master/examples/tddft/00-simple_tddft.py]):

```
>>> from pyscf import tdscf
>>> tdhf_h2o = tdscf.TDHF(rhf_h2o)
>>> tdhf_h2o.nstates = 6
>>> tdhf_h2o.kernel()
>>> tddft_h2o = tdscf.TDA(rks_h2o) # TDDFT with Tamm-Dancoff
approximation
>>> tddft_h2o.nstates = 4
>>> tddft_h2o.kernel()
```

From a converged time-dependent mean-field calculation, the corresponding natural transition orbitals for a particular excited state may be recovered as (cf. tddft/01-nto_analysis.py [https://github.com/pyscf/pyscf/blob/master/examples/tddft/01-nto_analysis.py]):

```
>>> weights, nto = tdhf_h2o.get_nto(state=2)
```

As an alternative to response theory, Δ-SCF with Gill's maximum overlap method is available to converge specific excited states, cf. scf/50-mom-deltaSCF.py [https://github.com/pyscf/pyscf/blob/master/examples/scf/50-mom-deltaSCF.py].

Spatially Localized Molecular Orbitals

PySCF offers a number of different standard schemes for localizing MOs, e.g., Pipek-Mezey, Foster-Boys, and Edmiston-Ruedenberg (cf. local_orb/03-split_localization.py [https://github.com/pyscf/pyscf/blob/master/examples/local_orb/03-split_localization.py]):

```
>>> from pyscf import lo
>>> occ_orbs = rhf_h2o.mo_coeff[:, rhf_h2o.mo_occ > 0.]
>>> fb_h2o = lo.Boys(mol_h2o, occ_orbs, rhf_h2o) # Foster-Boys
>>> loc_occ_orbs = fb.kernel()
>>> virt_orbs = rhf_h2o.mo_coeff[:, rhf_h2o.mo_occ == 0.]
>>> pm_h2o = lo.PM(mol_h2o, virt_orbs, rhf_h2o) # Pipek-Mezey
>>> loc_virt_orbs = pm.kernel()
```

Knizia's intrinsic bond orbitals can be computed as (cf. local_orb/04-ibo_benzene_cubegen.py [https://github.com/pyscf/pyscf/blob/master/examples/local_orb/04-ibo_benzene_cubegen.py]):

```
>>> iao = lo.iao.iao(mol, occ_orbs)
>>> iao = lo.vec_lowdin(iao, rhf_h2o.get_ovlp())
>>> ibo = lo.ibo.ibo(mol, occ_orbs, iaos=iao)
```

Relativistic Effects

PySCF implements a Dirac-Hartree-Fock solver for relativistic effects. Different Hamiltonians are available, for example with Breit and/or Gaunt interactions (cf. scf/05-breit_gaunt.py [https://github.com/pyscf/pyscf/blob/master/examples/scf/05-breit_gaunt.py]):

```
>>> dhf_c2 = scf.DHF(mol_c2)
>>> dhf_c2.with_gaunt = True
>>> dhf_c2.with_breit = True
>>> dhf_c2.kernel()
```

At lower cost, scalar relativistic effects can be included in a mean-field treatment by decorating the a **SCF** object (either HF or KS-DFT) with the `.x2c()` method (cf. scf/21-x2c.py [https://github.com/pyscf/pyscf/blob/master/examples/scf/21-x2c.py]). This modifies the Hamiltonian and affects subsequent correlated calculations:

```
>>> uks_o2_x2c = scf.UKS(mol_o2).x2c()
>>> uks_o2_x2c.kernel()
```

Symmetry Handling

Wave function symmetry may be explicitly controlled in an SCF calculation. For example, in the C_2 calculation of the above [#input] section, one can specify a given orbital occupancy through the `SCF.irrep_nelec` attribute (scf/13-symmetry.py [https://github.com/pyscf/pyscf/blob/master/examples/scf/13-symmetry.py]):

```
>>> rhf_c2 = scf.RHF(mol_c2)
>>> rhf_c2.irrep_nelec = {'Ag': 4, 'B1u': 4, 'B2u': 2, 'B3u': 2}
>>> e_c2 = rhf_c2.kernel()
```

Likewise, orbital symmetries may deduced from the MO coefficients (symm/32-symmetrize_natural_orbital [https://github.com/pyscf/pyscf/blob/master/examples/symm/32-symmetrize_natural_orbital.py]):

```
>>> from pyscf import symm
>>> orbsym = symm.label_orb_symm(mol_c2, mol_c2.irrep_id,
mol_c2.symm_orb, rhf_c2.mo_coeff)
```

Integrals & Density Fitting

1- and 2-Electron Integrals

A common use of PySCF is to obtain 1- and 2-electron integrals within a chosen MO basis. The latter are stored by default as (ij|kl) with 4-fold symmetry (cf. also ao2mo/00-mo_integrals.py [https://github.com/pyscf/pyscf/blob/master/examples/ao2mo/00-mo_integrals.py]):

```
>>> import numpy as np
>>> from pyscf import ao2mo
>>> hcore_ao = mol_h2o.intor_symmetric('int1e_kin') +
mol_h2o.intor_symmetric('int1e_nuc')
>>> hcore_mo = np.einsum('pi,pq,qj->ij', rhf_h2o.mo_coeff, hcore_ao,
rhf_h2o.mo_coeff)
>>> eri_4fold_ao = mol_h2o.intor('int2e_sph', aosym=4)
>>> eri_4fold_mo = ao2mo.incore.full(eri_4fold_ao, rhf_h2o.mo_coeff)
```

The transformed 2-electron integrals may be saved to and read from a file in HDF5 format (ao2mo/01-outcore.py [https://github.com/pyscf/pyscf/blob/master/examples/ao2mo/01-outcore.py]):

```
>>> import tempfile
>>> import h5py
>>> ftmp = tempfile.NamedTemporaryFile()
>>> ao2mo.kernel(mol_h2o, rhf_h2o.mo_coeff, ftmp.name)
>>> with h5py.File(ftmp.name) as f:
>>>     eri_4fold = f['eri_mo']
```

User-defined Hamiltonians can be used in PySCF, e.g., as input to a mean-field calculation and subsequent correlated treatments (mcscf/40-customizing_hamiltonian.py [https://github.com/pyscf/pyscf/blob/master/examples/mcscf/40-customizing_hamiltonian.py]):

```
>>> # 1D anti-PBC Hubbard model at half filling
>>> n, u = 12, 2.
>>> mol_hub = gto.M()
>>> mol_hub.nelectron = n // 2
>>> mol_hub.incore_anyway = True
>>> h1 = np.zeros([n] * 2, dtype=np.float64)
>>> for i in range(n-1):
```

```
>>>     h1[i, i+1] = h1[i+1, i] = -1.
>>> h1[n-1, 0] = h1[0, n-1] = -1.
>>> eri = np.zeros([n] * 4, dtype=np.float64)
>>> for i in range(n):
>>>     eri[i, i, i, i] = u
>>> rhf_hub = scf.RHF(mol_hub)
>>> rhf_hub.get_hcore = lambda *args: h1
>>> rhf_hub.get_ovlp = lambda *args: np.eye(n)
>>> rhf_hub._eri = ao2mo.restore(8, eri, n) # 8-fold symmetry
>>> rhf_hub.init_guess = '1e'
>>> rhf_hub.kernel()
```

Density Fitting Techniques

Density fitting of 2-electron integrals is conveniently invoked in either of two ways (cf. df/00-with_df.py [https://github.com/pyscf/pyscf/blob/master/examples/df/00-with_df.py]):

```
>>> rhf_c2_df = rhf_c2.density_fit(auxbasis='def2-universal-jfit') #
option 1
>>> from pyscf import df
>>> rhf_c2_df = df.density_fit(scf.RHF(mol_c2), auxbasis='def2-
universal-jfit') # option 2
```

In the former, decoration by the `scf.density_fit` function generates a new object that works in exactly the same way as the regular **SCF** object.

For an example of how to use density fitting alongside the Newton-Raphson SCF algorithm [#hf] and scalar relativistic effects [#rel], please see scf/23-decorate_scf.py [https://github.com/pyscf/pyscf/blob/master/examples/scf/23-decorate_scf.py].

Correlated Wave Function Theory

Perturbation Theory, Coupled Cluster, and Algebraic Diagrammatic Constructions

PySCF offers many facilities for correlated calculation. Common used examples include second-order Møller-Plesset, coupled cluster, and algebraic diagrammatic construction approximations. These modules can employ density fitting [#df], based on the `SCF.with_df` attribute of the underlying mean-field object (cf. mp/00-simple_mp2.py [https://github.com/pyscf/pyscf/blob/master/examples/mp/00-simple_mp2.py]):

```
>>> from pyscf import mp
>>> mp2_c2 = mp.MP2(rhf_c2)
>>> e_c2 = mp2_c2.kernel()[0]
>>> mp2_c2_df = mp.MP2(rhf_c2_df)
>>> e_c2_df = mp2_c2_df.kernel()[0]
```

At the coupled cluster level of theory, CCD, CCSD, and CCSD(T) calculation can be performed for both closed- and open-shell systems (cf. cc/00-simple_ccsd_t.py [https://github.com/pyscf/pyscf/blob/master/examples/cc/00-simple_ccsd_t.py]):

```
>>> from pyscf import cc
>>> ccsd_h2o = cc.CCSD(rhf_h2o)
>>> ccsd_h2o.direct = True # AO-direct algorithm to reduce I/O
overhead
>>> ccsd_h2o.frozen = 1 # frozen core
>>> e_ccsd = ccsd_h2o.kernel()[1]
>>> e_ccsd_t = e_ccsd + ccsd_h2o.ccsd_t()
```

Starting from a ground-state CCSD calculation, subsequent EOM-CCSD calculations in the spin-flip, IP/EA, and EE channels can be performed (cf. cc/20-ip_ea_eom_ccsd.py [https://github.com/pyscf/pyscf/blob/master/examples/cc/20-ip_ea_eom_ccsd.py]):

```
>>> e_ip_ccsd = ccsd_h2o.ipccsd(nroots=1)[0]
>>> e_ea_ccsd = ccsd_h2o.eaccsd(nroots=1)[0]
>>> e_ee_ccsd = ccsd_h2o.eeccsd(nroots=1)[0]
```

ADC(2), ADC(2)-X, and ADC(3) schemes have been implemented using a similar API (cf. adc/01-closed_shell.py [https://github.com/pyscf/pyscf/blob/master/examples/adc/01-closed_shell.py]):

```
>>> from pyscf import adc
>>> adc_h2o = adc.ADC(rhf_h2o)
>>> e_ip_adc2 = adc_h2o.kernel()[0] # IP-ADC(2) for 1 root
>>> adc_h2o.method = "adc(2)-x"
>>> adc_h2o.method_type = "ea"
>>> e_ea_adc2x = adc_h2o.kernel()[0] # EA-ADC(2)-x for 1 root
>>> adc_h2o.method = "adc(3)"
>>> adc_h2o.method_type = "ea"
>>> e_ea_adc3 = adc_h2o.kernel(nroots = 3)[0] # EA-ADC(3) for 3 roots
```

Full Configuration Interaction

PySCF offers powerful kernels to perform exact diagonalization of Hamiltonians optimized for different possible symmetries. For the simplest case, where all electrons of a given system are correlated among all orbitals, the syntax follows that of other correlation methods for closed- and open-shell systems (cf. fci/00-simple_fci.py [https://github.com/pyscf/pyscf/blob/master/examples/fci/00-simple_fci.py]):

```
>>> from pyscf import fci
>>> fci_h2o = fci.FCI(rhf_h2o)
>>> e_fci = fci_h2o.kernel()[0]
```

However, the various FCI solvers (tabulated in pyscf/fci/__init__.py [https://github.com/pyscf/pyscf/blob/master/pyscf/fci/__init__.py]) support user-defined 1- and 2-electron (`h1e,h2e`) Hamiltonians (cf. fci/01-given_h1e_h2e.py [https://github.com/pyscf/pyscf/blob/master/examples/fci/01-given_h1e_h2e.py]):

```
>>> fs = fci.direct_spin1.FCI() # direct_spin0 instead for singlet
system ground states
>>> e, fcivec = fs.kernel(h1e, h2e, N, 8) # 8 electrons in N orbitals
>>> e, fcivec = fs.kernel(h1e, h2e, N, (5,4))  # (5 alpha, 4 beta)
electrons
>>> e, fcivec = fs.kernel(h1e, h2e, N, (3,1))  # (3 alpha, 1 beta)
electrons
```

One can obtain multiple states by setting `FCI.nroots > 1`, and 1- to 4-electron density matrices, alongside 1- and 2-electron transition density matrices, (cf. fci/14-density_matrix.py [https://github.com/pyscf/pyscf/blob/master/examples/fci/14-density_matrix.py]):

```
>>> rdm1 = fs.make_rdm1(fcivec, N, (5, 4)) # spin-traced 1-electron
density matrix
>>> rdm1a, rdm1b = fs.make_rdm1s(fcivec, norb, (5, 4)) # alpha and
beta 1-electron density matrices
>>> t_rdm1 = fs.trans_rdm1(fcivec0, fcivec1, N, (5, 4)) # spin-traced
1-electron transition density matrix
```

In addition, the FCI code is accompanied by a large library of tools to manipulate and inspect wave functions, assign spin states and symmetry [#sym], etc.

Multiconfigurational Methods

Complete Active Space Configuration Interaction & Self-Consistent Field

The FCI solvers discussed above [#fci] can be used with complete active space methods in PySCF, all of which share a similar API (cf. mcscf/00-simple_casci.py [https://github.com/pyscf/pyscf/blob/master/examples/mcscf/00-simple_casci.py] & mcscf/00-simple_casscf.py [https://github.com/pyscf/pyscf/blob/master/examples/mcscf/00-simple_casscf.py]):

```
>>> from pyscf import mcscf
>>> casci_h2o = mcscf.CASCI(rhf_h2o, 6, 8)
>>> e_casci = casci_h2o.kernel()[0]
>>> casscf_h2o = mcscf.CASSCF(rhf_h2o, 6, 8)
>>> e_casscf = casscf_h2o.kernel()[0]
```

The CASCI and CASSCF classes support many variations, such as, density fitting of 2-electron integrals [#df] (cf. mcscf/16-density_fitting.py [https://github.com/pyscf/pyscf/blob/master/examples/mcscf/16-density_fitting.py]) and the standard frozen-core approximation (cf. mcscf/19-frozen_core.py [https://github.com/pyscf/pyscf/blob/master/examples/mcscf/19-frozen_core.py]):

```
>>> casscf_h2o_df = mcscf.DFCASSCF(rhf_h2o, 6, 8,
auxbasis='ccpvtzfit')
>>> casscf_h2o_df.frozen = 1 # frozen core
>>> e_casscf_df = casscf_h2o_df.kernel()[0]
```

In the case of CASSCF calculations, these may be performed in a state-specific or state-averaged manner (cf. mcscf/41-state_average.py [https://github.com/pyscf/pyscf/blob/master/examples/mcscf/41-state_average.py]):

```
>>> casscf_c2 = mcscf.CASSCF(rhf_c2, 8, 8)
>>> solver_t = fci.direct_spin1_symm.FCI(mol_c2)
>>> solver_t.spin = 2
>>> solver_t.nroots = 1
>>> solver_t = fci.addons.fix_spin(solver_t, shift=.2, ss=2) # 1
triplet
>>> solver_s = fci.direct_spin0_symm.FCI(mol_c2) # 2 singlets
>>> solver_s.spin = 0
>>> solver_s.nroots = 2
>>> mcscf.state_average_mix_(casscf_c2, [solver_t, solver_s],
np.ones(3) / 3.)
>>> casscf_c2.kernel()
```

Finally, additional dynamic correlation may be added by means of second-order perturbation theory in the form of NEVPT2 (cf. mrpt/02-cr2_nevpt2/cr2-scan.py [https://github.com/pyscf/pyscf/blob/master/examples/mrpt/02-cr2_nevpt2/cr2-scan.py]):

```
>>> from pyscf import mrpt
>>> e_nevpt2 = mrpt.NEVPT(casscf_h2o).kernel()
```

External Approximate Full Configuration Interaction Solvers

Besides the exact solvers discussed earlier [#fci], PySCF has interfaces to efficient approximate solvers. For instance, the StackBlock [https://github.com/sanshar/StackBlock] or block2 [https://github.com/block-hczhai/block2-preview] code can be used as a DMRG solver to perform parallel DMRGSCF calculations across several processes (cf. dmrg/01-dmrg_casscf_with_stackblock.py [https://github.com/pyscf/dmrgscf/blob/master/examples/01-dmrg_casscf_with_stackblock.py]):

```
>>> from pyscf import dmrgscf
>>> import os
>>> from pyscf.dmrgscf import settings
>>> if 'SLURMD_NODENAME' in os.environ: # slurm system
>>>     settings.MPIPREFIX = 'srun'
>>> elif 'PBS_NODEFILE' in os.environ: # PBS system
>>>     settings.MPIPREFIX = 'mpirun'
>>> else: # MPI on single node
>>>     settings.MPIPREFIX = 'mpirun -np 4'
>>> dmrgscf_c2 = dmrgscf.DMRGSCF(rhf_c2, 8, 8)
>>> dmrgscf_c2.state_average_([.5] * 2)
>>> dmrgscf_c2.fcisolver.memory = 4 # in GB
>>> dmrgscf_c2.fcisolver.num_thrds = 8 # number of threads to spawn on
each MPI process
>>> e_dmrgscf = dmrgscf_c2.kernel()
```

Similar interfaces to i-FCIQMC (using NECI [https://github.com/ghb24/NECI_STABLE]) and SHCI (using either Dice [https://github.com/sanshar/Dice] or Arrow [https://github.com/QMC-Cornell/shci]) are available.

Geometry Optimization

In PySCF, geometry optimizations can be performed using the geomeTRIC [https://github.com/leeping/geomeTRIC] or PyBerny [https://github.com/jhrmnn/pyberny] libraries (cf. geomopt/01-geomeTRIC.py [https://github.com/pyscf/pyscf/blob/master/examples/geomopt/01-geomeTRIC.py] and geomopt/01-pyberny.py [https://github.com/pyscf/pyscf/blob/master/examples/geomopt/01-pyberny.py], respectively):

```
>>> from pyscf.geomopt.geometric_solver import optimize as
geometric_opt
>>> mol_h2o_rhf_eq = geometric_opt(rhf_h2o)
>>> from pyscf.geomopt.berny_solver import optimize as pyberny_opt
>>> mol_h2o_casscf_eq = pyberny_opt(casscf_h2o)
```

For multiconfigurational methods [#cas], the geometry of an excited state can be optimized in either a state-specific or state-averaged manner (cf. geomopt/12-mcscf_excited_states.py [https://github.com/pyscf/pyscf/blob/master/examples/geomopt/12-mcscf_excited_states.py]):

```
>>> casci_h2o.state_specific_(2) # state-specific opt
>>> casci_grad = casci_h2o.nuc_grad_method().as_scanner()
>>> mol_h2o_casci_2nd_ex = casci_grad.optimizer().kernel()
>>> casscf_h2o.state_average_([.25] * 4) # state-averaged opt
>>> casscf_grad = casscf_h2o.nuc_grad_method().as_scanner()
>>> mol_h2o_sa_casscf = casscf_grad.optimizer().kernel()
```

Solvent Effects

Polarizable Continuum & COSMO Methods

PySCF can include solvent effects into most types of calculations, as implemented by either PCM or COSMO (both in their domain-decomposed formulations [https://www.ddpcm.org/] that yields a fast discretization of the polarization equations). For instance, geometry optimizations [#geomopt] can be performed for ground and excited states in the presence of a solvent (cf. solvent/21-tddft_geomopt.py [https://github.com/pyscf/pyscf/blob/master/examples/solvent/21-tddft_geomopt.py]):

```
>>> rhf_h2o_pcm = mol_h2o.RHF().ddPCM()
>>> rhf_h2o_pcm.kernel()
>>> tdhf_h2o_pcm = rhf_h2o_pcm.TDA().ddPCM()
>>> tdhf_h2o_pcm.with_solvent.equilibrium_solvation = True
>>> mol_h2o_tdhf_pcm_2nd_ex =
tdhf_h2o_pcm.nuc_grad_method().as_scanner(state=2).optimizer().kernel()
```

Similarly, correlated methods, e.g., CCSD, may be performed in the presence of a solvent with either a relaxed or unrelaxed (mean-field) potential (cf. solvent/03-ccsd_with_ddcosmo.py [https://github.com/pyscf/pyscf/blob/master/examples/solvent/03-ccsd_with_ddcosmo.py]):

```
>>> from pyscf import solvent
>>> rhf_h2o_cosmo = mol_h2o.RHF().ddCOSMO()
>>> rhf_h2o_cosmo.kernel()
>>> ccsd_h2o_cosmo_rel = solvent.ddCOSMO(cc.CCSD(rhf_h2o))
>>> ccsd_h2o_cosmo_rel.kernel()
>>> ccsd_h2o_cosmo_unrel = solvent.ddCOSMO(cc.CCSD(rhf_h2o_cosmo),
dm=rhf_h2o_cosmo.make_rdm1())
>>> ccsd_h2o_cosmo_unrel.kernel()
```

Different solvents are chosen by setting the `SCF.with_solvent.eps` attribute.

Quantum Mechanics/Molecular Mechanics Methods

QM/MM calculations can be performed by either of two different methods in PySCF. Standard point charges can be included in most calculations, be these mean-field or correlated. In the latter case, background charges can be conveniently added to the underlying SCF calculation (cf. qmmm/03-ccsd.py [https://github.com/pyscf/pyscf/blob/master/examples/qmmm/03-ccsd.py]):

```
>>> from pyscf import qmmm
>>> coords = np.random.random((5, 3)) * 10.
>>> charges = (np.arange(5.) + 1.) * -.1
>>> rhf_h2o_qmmm = qmmm.mm_charge(rhf_h2o, coords, charges)
>>> rhf_h2o_qmmm.kernel()
>>> ccsd_h2o_qmmm = cc.CCSD(rhf_h2o_qmmm)
>>> e_ccsd = ccsd_h2o_qmmm.kernel()[1]
```

Alternatively, a combination of the PyFraME [https://gitlab.com/FraME-projects/PyFraME] framework and CPPE [https://github.com/maxscheurer/cppe] library can be used to prepare and run polarizable embedding calculations from within PySCF, cf. solvent/04-pe_potfile_from_pyframe.py [https://github.com/pyscf/pyscf/blob/master/examples/solvent/04-pe_potfile_from_pyframe.py] and solvent/04-scf_with_pe.py [https://github.com/pyscf/pyscf/blob/master/examples/solvent/04-scf_with_pe.py].

Periodic Boundary Conditions

PySCF provides equal support for molecular and crystalline materials calculations. The latter functionality is found in the PBC modules. As discussed above in input parsing [#input], the API to initialize unit cells is almost identical to that for finite-size molecules (cf. pbc/00-input_cell.py [https://github.com/pyscf/pyscf/blob/master/examples/pbc/00-input_cell.py]):

```
>>> from pyscf.pbc import gto as pbcgto
>>> cell_diamond = pbcgto.M(atom = '''C     0.      0.      0.
                                      C     .8917    .8917   .8917
                                      C     1.7834  1.7834  0.
                                      C     2.6751  2.6751   .8917
                                      C     1.7834  0.      1.7834
                                      C     2.6751   .8917  2.6751
                                      C     0.      1.7834  1.7834
                                      C     .8917   2.6751  2.6751''',
                            basis = 'gth-szv',
                            pseudo = 'gth-pade',
                            a = np.eye(3) * 3.5668)
```

Besides the difference in import of the `gto` module, the only new attributes are `Cell.a` and `Cell.pseudo`. The former (`a`) is a matrix of lattice vectors, where each row denotes a primitive vector, while the latter (`pseudo`) is an (optional) crystal pseudo potential (cf. pyscf/pbc/gto/pseudo/GTH_POTENTIALS [https://github.com/pyscf/pyscf/blob/master/pyscf/pbc/gto/pseudo/GTH_POTENTIALS] for a full list of these). All electron calculations can be performed by omitting `pseudo`. Alternatively, molecular effective core potentials can be used by setting the `Cell.ecp` attribute, cf. pbc/05-input_pp.py [https://github.com/pyscf/pyscf/blob/master/examples/pbc/05-input_pp.py].

Generally, crystal calculations use the same or similar APIs to the molecule calculations. For instance, an all-electron calculation with k-point sampling at the KS-DFT level using density fitting [#df] (recommended) and a second-order SCF algorithm [#hf] is specified as (cf. pbc/21-k_points_all_electron_scf.py [https://github.com/pyscf/pyscf/blob/master/examples/pbc/21-k_points_all_electron_scf.py]):

```
>>> from pyscf.pbc import dft as pbcdft
>>> kpts = cell_diamond.make_kpts([4] * 3) # 4 k-poins for each axis
>>> krks_diamond = pbcdft.KRKS(cell_diamond,
kpts).density_fit(auxbasis='weigend')
>>> krks_diamond.xc = 'bp86'
>>> krks_diamond = krks_diamond.newton()
>>> krks_diamond.kernel()
```

After converging the mean-field calculation, dynamic electron correlation can be treated by various methods, e.g. MP2 or (EOM-)CCSD. These calculations can again be performed with k-point sampling (cf. pbc/22-k_points_ccsd.py [https://github.com/pyscf/pyscf/blob/master/examples/pbc/22-k_points_ccsd.py]). At the Γ-point, all correlation methods available for molecules can be employed (cf. pbc/12-gamma_point_post_hf.py [https://github.com/pyscf/pyscf/blob/master/examples/pbc/12-gamma_point_post_hf.py]):

```
>>> from pyscf.pbc import scf as pbcscf
>>> rhf_diamond = pbcscf.RHF(cell_diamond).density_fit()
>>> rhf_diamond.kernel()
>>> ccsd_diamond = cc.CCSD(rhf_diamond)
>>> ccsd_diamond.kernel()
```

Summary

The above quickstart provides only a small sample of what is in PySCF from the perspective of a computational user. More details can be found in the user guide, and we highly encourage users to browse the large set of examples. Note that PySCF is designed for easy extensibility. Users interested in these aspects are encouraged to dive into the code! We sincerely hope you'll enjoy using the library. For bug reports and feature requests, please submit tickets on the issues [https://github.com/pyscf/pyscf/issues] page.