The Python-based Simulations of Chemistry Framework (PySCF)

Qiming Sun*[1], Timothy C. Berkelbach[2], Nick S. Blunt[3,4], George H. Booth[5], Sheng Guo[1], Zhendong Li[1], Junzi Liu[6], James McClain[1], Sandeep Sharma[7], Sebastian Wouters[8], and Garnet Kin-Lic Chan†[1]

[1]Division of Chemistry and Chemical Engineering, California Institute of Technology, Pasadena CA 91125, USA
[2]Department of Chemistry and James Franck Institute, University of Chicago, Chicago, Illinois 60637, USA
[3]Chemical Science Division, Lawrence Berkeley National Laboratory, Berkeley, California 94720, USA
[4]Department of Chemistry, University of California, Berkeley, California 94720, USA
[5]Department of Physics, King's College London, Strand, London WC2R 2LS, United Kingdom
[6]Institute of Chemistry Chinese Academy of Sciences, Beijing 100190, P. R. China
[7]Department of Chemistry and Biochemistry, University of Colorado Boulder, Boulder, CO 80302, USA
[8]Brantsandpatents, Pauline van Pottelsberghelaan 24, 9051 Sint-Denijs-Westrem, Belgium

Abstract

PySCF is a general-purpose electronic structure platform designed from the ground up to emphasize code simplicity, both to aid new method development, as well as for flexibility in computational workflow. 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

*osirpt.sun@gmail.com
†gkc1000@gmail.com

post-mean-field methods with standard Gaussian basis functions. To ensure easy of extensibility, PySCF uses the Python language to implement almost all 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. In this paper we document the capabilities and design philosophy of the current version of the PySCF package.

1 INTRODUCTION

The Python programming language is playing an increasingly important role in scientific computing. As a high level language, Python supports rapid development practices and easy program maintenance. While programming productivity is hard to measure, it is commonly thought that it is more efficient to prototype new ideas in Python, rather than in traditional low-level compiled languages such as Fortran or C/C++. Further, through the use of the many high-quality numerical libraries available in Python – such as NumPy[1], SciPy[2], and MPI4Py[3] – Python programs can perform at competitive levels with optimized Fortran and C/C++ programs, including on large-scale computing architectures.

There have been several efforts in the past to incorporate Python into electronic structure programs. Python has been widely adopted in a scripting role: the Psi4[4] quantum chemistry package uses a custom "Psithon" dialect to drive the underlying C++ implementation, while general simulation environments such as ASE[5] and PyMatGen[6] provide Python frontends to multiple quantum chemistry and electronic structure packages, to organize complex workflows[7]. Python has also proved popular for implementing symbolic second-quantized algebra and code generation tools, such as the Tensor Contraction Engine[8] and the SecondQuantizationAlgebra library[9,10].

In the above cases, Python has been used as a supporting language, with the underlying quantum chemistry algorithms implemented in a compiled language. However, Python has also seen some use as a primary implementation language for electronic structure methods. PyQuante[11] was an early attempt to implement a Gaussian based quantum chemistry code in Python, although it did not achieve speed or functionality competitive with typical packages. Another early effort was the GPAW[12] code, which implements the projector augmented wave formalism for density functional theory, and which is still under active development in multiple groups. Nonetheless, it is probably fair to say that using Python as an implementational, rather than supporting language, remains the exception rather than the rule in modern quantum chemistry and electronic structure software efforts.

With the aim of developing a new highly functional, high-performance computing toolbox for the quantum chemistry of molecules and materials implemented primarily in the

Python language, we started the open-source project "Python-based Simulations of Chemistry Framework" (PySCF) in 2014. The program was initially ported from our quantum chemistry density matrix embedding theory (DMET) project[13] and contained only the Gaussian integral interface, a basic Hartree-Fock solver, and a few post-Hartree-Fock components required by DMET. In the next 18 months, multi-configurational self-consistent-field (MCSCF), density functional theory and coupled cluster theory, as well as relevant modules for molecular properties, were added into the package. In 2015, we released the first stable version, PySCF 1.0, wherein we codified our primary goals for further code development: to produce a package that emphasizes simplicity, generality, and efficiency, in that order. As a result of this choice, most functions in PySCF are written purely in Python, with a very limited amount of C code only for the most time-critical parts. The various features and APIs are designed and implemented in the simplest and most straightforward manner, so that users can easily modify the source code to meet their own scientific needs and workflow. However, although we have favored algorithm accessibility and extensibility over performance, we have found that the efficient use of numerical Python libraries allows PySCF to perform at least as fast as the best existing quantum chemistry implementations. In this article, we highlight the current capabilities and design philosophy of the PySCF package.

2 CAPABILITIES

Molecular electronic structure methods are typically the main focus of quantum chemistry packages. We have put significant effort towards the production of a stable, feature-rich and efficient molecular simulation environment in PySCF. In addition to molecular quantum chemistry methods, PySCF also provides a wide range of quantum chemistry methods for extended systems with periodic boundary conditions (PBC). Table 1 lists the main electronic structure methods available in the PySCF package. More detailed descriptions are presented in Section 2.1 - Section 2.7. Although not listed in the table, many auxiliary tools for method development are also part of the package. They are briefly documented in Section 2.8 - Section 2.13.

2.1 Self-consistent field (SCF) methods

SCF methods are the starting point for most electronic structure calculations. In PySCF, the SCF module includes implementations of Hartree-Fock (HF) and density functional theory (DFT) for restricted, unrestricted, closed shell and open shell Slater determinant references. A wide range of predefined exchange-correlation (XC) functionals are supported through a general interface to the Libxc[14] and Xcfun[15] functional libraries. Using the interface, as shown in Figure 1, one can easily customize the XC functionals in DFT calculations. PySCF uses the Libcint[16] Gaussian integral library, written by one of us (QS) as its integral engine. In its current implementation, the SCF program can handle over 5000 basis functions on a single symmetric multiprocessing (SMP) node without any approximations to the integrals. To obtain rapid convergence in the SCF iterations, we have also developed a second order co-iterative augmented Hessian (CIAH) algorithm for orbital optimization[17]. Using the direct SCF technique with the CIAH algorithm, we are able to converge a Hartree-Fock calculation for the open-shell molecule Fe(II)-porphine (2997 AOs) on a 16-core node in one day.

2.2 Post SCF methods

Single-reference correlation methods can be used on top of the HF or DFT references, including Moller-Plesset second-order perturbation theory (MP2), configuration interaction, and coupled cluster theory.

Canonical single-reference coupled cluster theory has been implemented at the coupled cluster singles doubles (CCSD)[18], and CCSD with perturbative triples [CCSD(T)] level. The associated derivative routines include CCSD and CCSD(T) density matrices, CCSD and CCSD(T) analytic gradients, and equation-of-motion CCSD for the ionization potentials, electron affinities, and excitation energies (EOM-IP/EA/EE-CCSD)[19–21]. The package contains two complementary implementations of each of these methods. The first set are straightforward spin-orbital and spatial-orbital implementations, which are optimized for readability and written in pure Python using the NumPy `einsum` function (which can use either the default Numpy implementation or a custom `gemm`-based version) for tensor con-

traction. These implementations are easy for the user to modify. A second spatial-orbital implementation has been intensively optimized to minimize dataflow and uses asynchronous I/O and a threaded `gemm` function for efficient tensor contractions. For a system of 25 occupied orbitals and 1500 virtual orbitals, the efficient CCSD implementation takes less than 3 hours to finish one iteration using 28 CPU cores.

The configuration interaction code implements two solvers: a solver for configuration interaction with singles and doubles (CISD), and a determinant-based full configuration interaction (FCI) solver[22] for fermion, boson or coupled fermion-boson Hamiltonians. The CISD solver has a similar program layout to the CCSD solver. The FCI solver additionally implements the spin-squared operator, second quantized creation and annihilation operators (from which arbitrary second quantized algebra can be implemented), functions to evaluate the density matrices and transition density matrices (up to fourth order), as well as a function to evaluate the overlap of two FCI wavefunctions in different orbital bases. The FCI solver is intensively optimized for multi-threaded performance. It can perform one matrix-vector operation for 16 electrons and 16 orbitals using 16 CPU cores in 30 seconds.

2.3 Multireference methods

For multireference problems, the PySCF package provides the complete active space self consistent field (CASSCF) method[23,24] and N-electron valence perturbation theory (NEVPT2)[25,26]. When the size of the active space exceeds the capabilities of the conventional FCI solver, one can switch to external variational solvers such as a density matrix renormalization group (DMRG) program[27–29] or a full configuration interaction quantum Monte Carlo (FCIQMC) program[30,31]. Incorporating external solvers into the CASSCF optimizer widens the range of possible applications, while raising new challenges for an efficient CASSCF algorithm. One challenge is the communication between the external solver and the orbital optimization driver; communication must be limited to quantities that are easy to obtain from the external solver. A second challenge is the cost of handling quantities associated with the active space; for example, as the active space becomes large, the memory required to hold integrals involving active labels can easily exceed available memory. Finally, any approximations introduced in the context of the above two challenges should not interfere with the

quality of convergence of the CASSCF optimizer.

To address these challenges, we have implemented a general AO-driven CASSCF optimizer[29] that provides second order convergence and which may easily be combined with a wide variety of external variational solvers, including DMRG, FCIQMC and their state-averaged solvers. Only the 2-particle density matrix and Hamiltonian integrals are communicated between the CASSCF driver and the external CI solver. Further, the AO-driven algorithm has a low memory and I/O footprint. The current implementation supports calculations with 3000 basis functions and 30–50 active orbitals on a single SMP node with 128 GB memory, without any approximations to the AO integrals.

A simple interface is provided to use an external solver in multiconfigurational calculations. Figure 2 shows how to perform a DMRG-CASSCF calculation by replacing the `fcisolver` attribute of the CASSCF method. DMRG-SC-NEVPT2[26], and ic-MPS-PT2 and ic-MPS-LCC[32] methods are also available through the interface to the DMRG program package Block[27,33–35], and the ic-MPS-LCC program of Sharma[32].

2.4 Molecular properties

At the present stage, the program can compute molecular properties such as analytic nuclear gradients, analytic nuclear Hessians, and NMR shielding parameters at the SCF level. The CCSD and CCSD(T) modules include solvers for the Λ-equations. As a result, we also provide one-particle and two-particle density matrices, as well as the analytic nuclear gradients, for the CCSD and CCSD(T) methods[36].

For excited states, time-dependent HF (TDHF) and time-dependent DFT (TDDFT) are implemented on top of the SCF module. The relevant analytic nuclear gradients are also programmed[37]. The CCSD module offers another option to obtain excited states using the EOM-IP/EA/EE-CCSD methods. The third option to obtain excited states is through the multi-root CASCI/CASSCF solvers, optionally followed by the MRPT tool chain. Based on multi-root CASCI/CASSCF solutions, the program can compute the density matrices of all the states and the transition density matrices between any two states. One can contract these density matrices with specific AO integrals to obtain different first order molecular properties.

2.5 Relativistic effects

Many different relativistic treatments are available in PySCF. Scalar relativistic effects can be added to all SCF and post-SCF methods through relativistic effective core potentials (ECP)[38] or the all-electron spin-free X2C[39] relativistic correction. For a more advanced treatment, PySCF also provides 4-component relativistic Hartree-Fock and no-pair MP2 methods with Dirac-Coulomb, Dirac-Coulomb-Gaunt, and Dirac-Coulomb-Breit Hamiltonians. Although not programmed as a standalone module, the no-pair CCSD electron correlation energy can also be computed with the straightforward spin-orbital version of the CCSD program. Using the 4-component Hamiltonian, molecular properties including analytic nuclear gradients and NMR shielding parameters are available at the mean-field level[40].

2.6 Orbital localizer and result analysis

Two classes of orbital localization methods are available in the package. The first emphasizes the atomic character of the basis functions. The relevant localization functions can generate intrinsic atomic orbitals (IAO)[41], natural atomic orbitals (NAO)[42], and meta-Löwdin orbitals[13] based on orbital projection and orthogonalization. With these AO-based local orbitals, charge distributions can be properly assigned to atoms in population analysis[41]. In the PySCF population analysis code, meta-Löwdin orbitals are the default choice.

The second class, represented by Boys-Foster, Edmiston-Ruedenberg, and Pipek-Mezey localization, require minimizing (or maximizing) the dipole, the Coulomb self-energy, or the atomic charges, to obtain the optimal localized orbitals. The localization routines can take arbitrary orthogonal input orbitals and call the CIAH algorithm to rapidly converge the solution. For example, using 16 CPU cores, it takes 3 minutes to localize 1620 HF unoccupied orbitals for the C_{60} molecule using Boys localization.

A common task when analysing the results of an electronic structure calculation is to visualize the orbitals. Although PySCF does not have a visualization tool itself, it provides a module to convert the given orbital coefficients to the `molden`[43] format which can be read and visualized by other software, e.g. Jmol[44]. Figure 3 is an example to run Boys localization for the C_{60} HF occupied orbitals and to generate the orbital surfaces of the

localized σ-bond orbital in a single Python script.

2.7 Extended systems with periodic boundary conditions

PBC implementations typically use either plane waves[45–48] or local atomic functions[12,49–53] as the underlying orbital basis. The PBC implementation in PySCF uses the local basis formulation, specifically crystalline orbital Gaussian basis functions ϕ, expanded in terms of a lattice sum over local Gaussians χ

$$\phi_{\boldsymbol{k},\chi}(\boldsymbol{r}) = \sum_{\boldsymbol{T}} e^{i\boldsymbol{k}\cdot\boldsymbol{T}} \chi(\boldsymbol{r} - \boldsymbol{T})$$

where $\boldsymbol{k}$ is a vector in the first Brillouin zone and $\boldsymbol{T}$ is a lattice translational vector. We use a pure Gaussian basis in our PBC implementation for two reasons: to simplify the development of post-mean-field methods for extended systems and to have a seamless interface and direct comparability to finite-sized quantum chemistry calculations. Local bases are favourable for post-mean-field methods because they are generally quite compact, resulting in small virtual spaces[54], and further allow locality to be exploited. Due to the use of local bases, various boundary conditions can be easily applied in the PBC module, from zero-dimensional systems (molecules) to extended one-, two- and three-dimensional periodic systems.

The PBC module supports both all-electron and pseudopotential calculations. Both separable pseudopotentials (e.g. Hartwigsen-Goedecker-Hutter (HGH) pseudopotentials) and non-separable pseudopotentials (quantum chemistry ECPs and Burkatzi-Filippi-Dolg pseudopotentials[55]) can be used. In the separable pseudopotential implementation, the associated orbitals and densities are guaranteed to be smooth, allowing a grid-based treatment that uses discrete fast Fourier transforms[53,56]. In both the pseudopotential and all-electron PBC calculations, Coulomb-based integrals are handled via density fitting as described in Section 2.10.

The PBC implementation is organized in direct correspondence to the molecular implementation. We implemented the same function interfaces as in the molecular code, with the same module and function names. Consequently, methods defined in the molecular part of the code can be seamlessly mixed with the PBC functions without modification, especially in Γ-point calculations where the PBC wave functions are real. Thus, starting from

PBC Γ-point mean-field orbitals, one can, for example, carry out CCSD, CASSCF, TDDFT, etc. calculations using the molecular implementations. We also introduce specializations of the PBC methods to support k-point (Brillouin zone) sampling. The k-point methods slightly modify the Γ-point data structures, but inherit from and reuse almost all of the Γ-point functionality. Explicit k-point sampling is supported at the HF and DFT level, and on top of this we have also implemented k-point MP2, CCSD, CCSD(T) and EOM-CCSD methods[56], with optimizations to carefully distribute work and data across cores. On 100 computational cores, mean-field simulations including unit cells with over 100 atoms, or k-point CCSD calculations with over 3000 orbitals, can be executed without difficulty.

2.8 General AO integral evaluator and J/K builds

Integral evaluation forms the foundation of Gaussian based electronic structure simulation. The general integral evaluator library Libcint supports a wide range of GTO integrals, and PySCF exposes simple APIs to access the Libcint integral functions. As the examples in Figure 4 show, the PySCF integral API allows users to access AO integrals either in a giant array, or in individual shells, with a single line of Python code. The integrals provided include,

- Integrals in the basis of Cartesian, real-spherical and j-adapted spinor GTOs;
- Arbitrary integral expressions built from $\boldsymbol{r}$, $\boldsymbol{p}$, and σ polynomials;
- 2-center, 3-center and 4-center 2-electron integrals for the Coulomb operator $1/r_{12}$, range-separated Coulomb operator $\mathrm{erf}(\omega r_{12})/r_{12}$, Gaunt interaction, and Breit interaction.

Based on the general AO integral evaluator, the package provides a general AO-driven J/K contraction function. J/K-matrix construction involves a contraction over a high order tensor (e.g. 4-index 2-electron integrals $(ij|kl)$) and a low order tensor (e.g. the 2-index density matrix γ)

$$J_{ij} = \sum_{kl} (ij|kl)\gamma_{kl}$$

$$K_{il} = \sum_{jk} (ij|kl)\gamma_{jk}$$

When both tensors can be held in memory, the NUMPY package offers a convenient tensor contraction function `einsum` to quickly construct J/K matrices. However, it is common for the high order tensor to be too large to fit into the available memory. Using the Einstein summation notation of the NUMPY `einsum` function, our AO-driven J/K contraction implementation offers the capability to contract the high order tensor (e.g. 2-electron integrals or their high order derivatives) with multiple density matrices, with a small memory footprint. The J/K contraction function also supports subsystem contraction, in which the 4 indices of the 2-electron integrals are distributed over different segments of the system which may or may not overlap with each other. This subsystem contraction is particularly useful in two scenarios: in fragment-based methods, where the evaluation of Coulomb or exchange energies involves integral contraction over different fragments, and in parallel algorithms, where one partitions the J/K contraction into small segments and distributes them to different computing nodes.

2.9 General integral transformations

Integral transformations are another fundamental operation found in quantum chemistry programs. A common kind of integral transformation is to transform the 4 indices of the 2-electron integrals by 4 sets of different orbitals. To satisfy this need, we designed a general integral transformation function to handle the arbitrary AO integrals provided by the LIBCINT library and arbitrary kinds of orbitals. To reduce disk usage, we use permutation symmetry over i and j, k and l in $(ij|kl)$ whenever possible for real integrals.

Integral transformations involve high computational and I/O costs. A standard approach to reduce these costs involves precomputation to reduce integral costs and data compression to increase I/O throughput. However, we have not adopted such an optimization strategy in our implementation because it is against the objective of simplicity for the PYSCF package. In our implementation, initialization is not required for the general integral transformation function. Similarly to the AO integral API, the integral transformation can thus be launched with one line of Python code. In the integral data structure, we store the transformed inte-

grals by chunks in the HDF5 format without compression. This choice has two advantages. First, it allows for fast indexing and hyperslab selection for subblocks of the integral array. Second, the integral data can be easily accessed by other program packages without any overhead for parsing the integral storage protocol.

2.10 Density fitting

The density fitting (DF) technique is implemented for both finite-sized systems and crystalline systems with periodic boundary conditions.

In finite-sized systems, one can use DF to approximate the J/K matrix and the MO integrals for the HF, DFT and MP2 methods. To improve the performance of the CIAH algorithm, one can use the DF orbital Hessian in the CIAH orbital optimization for Edmiston-Ruedenberg localization and for the HF, DFT and CASSCF algorithms.

In the PBC module, the 2-electron integrals are represented as the product of two 3-index tensors which are treated as DF objects. Based on the requirements of the system being modelled, we have developed various DF representations. When the calculation involves only smooth bases (typically with pseudopotentials), plane waves are used as the auxiliary fitting functions and the DF 3-index tensor is computed within a grid-based treatment using discrete fast Fourier transforms[56]. When high accuracy in all-electron calculations is required, a mixed density fitting technique is invoked in which the fitting functions are Gaussian functions plus plane waves. Besides the choice of fitting basis, different metrics (e.g. overlap, kinetic, or Coulomb) can be used in the fitting to balance performance against computational accuracy.

The 3-index DF tensor is stored as a giant array in the HDF5 format without compression. With this design, it is straightforward to access the 2-electron integrals with the functions of the PySCF package. Moreover, it allows us to supply 2-electron integrals to calculations by overloading the DF object in cases where direct storage of the 4-index integrals in memory or on disk is infeasible (see discussion in Section 2.11).

2.11 Custom Hamiltonians

Most quantum chemistry approximations are not tied to the details of the ab initio molecular or periodic Hamiltonian. This means that they can also be used with arbitrary model Hamiltonians, which is of interest for semi-empirical quantum chemistry calculations as well as condensed-matter model studies. In PySCF, overwriting the predefined Hamiltonian is straightforward. The Hamiltonian is an attribute of the mean-field calculation object. Once the 1-particle and 2-particle integral attributes of the mean-field object are defined, they are used by the mean-field calculation and all subsequent post-Hartree-Fock correlation treatments. Users can thus carry out correlated calculations with model Hamiltonians in exactly the same way as with standard ab initio Hamiltonians. Figure 5 displays an example of how to input a model Hamiltonian.

2.12 Interfaces to external programs

PySCF can be used either as the driver to execute external programs or as an independent solver to use as part of a computational workflow involving other software. In PySCF, the DMRG programs Block[27] and CheMPS2[28,57] and the FCIQMC program NECI[58] can be used as a replacement for the FCI routine for large active spaces in the CASCI/CASSCF solver. In the QM/MM interface, by supplying the charges and the positions of the MM atoms, one can compute the HF, DFT, MP2, CC, CI and MCSCF energies and their analytic nuclear gradients.

To communicate with other quantum chemistry programs, we provide utility functions to read and write Hamiltonians in the Molpro[59] `FCIDUMP` format, and arbitrary orbitals in the `molden`[43] format. The SCF solution and CI wavefunction can be dumped in the GAMESS[60] `WFN` format. The real space electron density can be output on cubic grids in the Gaussian[61] `cube` format.

2.13 Numerical tools

Although the Numpy and Scipy libraries provide a wide range of numerical tools for scientific computing, there are some numerical components commonly found in quantum chem-

istry algorithms that are not provided by these libraries. For example, the direct inversion of the iterative space (DIIS) method[62,63] is one of the most commonly used tools in quantum chemistry to speed up optimizations when a second order algorithm is not available. In PySCF we provide a general DIIS handler for an object array of arbitrary size and arbitrary data type. In the current implementation, it supports DIIS optimization both with or without supplying the error vectors. For the latter case, the differences between the arrays of adjacent iterations are minimized. Large scale eigenvalue problems and linear equation solves are also common components of many quantum chemistry methods. The Davidson diagonalization algorithm and Arnoldi/Krylov subspace solver are accessible in PySCF through simple APIs.

The Numpy `einsum` function is a convenient tool for tensor contraction. Although the default `einsum` function can handle a wide range of contraction patterns, its performance is not optimal. To improve the performance of Numpy, one direction is to optimize the contraction ordering for multiple tensors[64]. Another optimization, which is our focus, is to improve the efficiency by contracting two high-dimensional tensors using the BLAS `gemm` functions. Our custom `einsum` employs the same API as the Numpy `einsum` but provides 10-fold to 100-fold speedups for larger matrices.

3 Design and implementation of PySCF

While we have tried to provide rich functionality for quantum chemical simulations with the built-in functions of the PySCF package, it will nonetheless often be the case that a user's actual needs are not covered by the built-in functionality. A major design goal has been to implement PySCF in a sufficiently flexible way so that users can easily extend its functionality. To provide robust components for complex problems and non-trivial workflows, we have made the following general design choices in PySCF:

1. *Language*: Mostly Python, with a little C. We believe that it is easiest to develop and test new functionality in Python. For this reason, most functions in PySCF are written in pure Python. Only a few computational hot spots have been rewritten and optimized in C.

2. *Style*: Mostly functional, with a little object-oriented programming (OOP). Although OOP is a successful and widely used programming paradigm, we feel that it is hard for users to customize typical OOP programs without learning details of the object hierarchy and interfaces. We have adopted a functional programming style, where most functions are pure, and thus can be invoked alone and independently of each other. This allows users to mix functionality with a minimal knowledge of the PySCF internals.

We elaborate on these choices below.

3.1 Input language

Almost every quantum chemistry package today uses its own custom input language. This is a burden to the user, who must become familiar with a new domain specific language for every new package. In contrast, PySCF does not have an input language. Rather, the functionality is simply called from an input script written in the host Python language. This choice has clear benefits:

1. *There is no need to learn a domain specific language.* Python, as a general programming language, is already widely used for numerical computing, and is taught in modern computer science courses. For novices, the language is easy to learn and help is readily available from the large Python community.

2. *One can use all Python language features in the input script.* This allows the input script to implement complex logic and computational workflows, and to carry out tasks (e.g. data processing and plotting) in the same script as the electronic structure simulation (see Figure 6 for an example).

3. *The computational environment is easily extended beyond that provided by the* PySCF *package.* The PySCF package is a regular Python module which can be mixed and matched with other Python modules to build a personalized computing environment.

4. *Computing can be carried out interactively.* Simulations can be tested, debugged, and executed step by step within the Python interpreter shell.

3.2 Enabling interactive computing

As discussed above, a strength of the PySCF package is that its functionality can be invoked from the interactive Python shell. However, maximizing its usability in this interactive mode entails additional design optimizations. There are three critical considerations to facilitate such interactive computations:

1. The functions and data need to be easy to access;
2. Functions should be insensitive to execution order (when and how many times a function is called should not affect the result);
3. Computations should not cause (significant) halts in the interactive shell.

To address these requirements, we have enforced the following design rules wherever possible in the package:

1. Functions are pure (i.e. state free). This ensures that they are insensitive to execution order;
2. Method objects (classes) only hold results and control parameters;
3. There is no initialization of functions, or at most a short initialization chain;
4. Methods are placed at both the module level and within classes so that the methods and their documentation can be easily accessed by the interactive shell (see Figure 7).

A practical solution to eliminate halting of the interactive shell is to overlap the REPL (read-eval-print-loop) and task execution. Such task parallelism requires the underlying tasks to be independent of each other. Although certain dependence between methods is inevitable, the above design rules greatly reduce function call dependence. Most functions in PySCF can be safely placed in the background using the standard Python `threading` and `multiprocessing` libraries.

3.3 Methods as plugins

Ease-of-use is the primary design objective of the PySCF package. However, function simplicity and versatility are difficult to balance in the same software framework. To balance readability and complexity, we have implemented only the basic algorithmic features in the main methods, and placed advanced features in additional "plugins". For instance, the main mean-field module implements only the basic self-consistent loop. Corrections, such as for relativistic effects are implemented in an independent plugin module, which can be activated by reassigning the mean-field 1-electron Hamiltonian method at runtime. Although this design increases the complexity of implementation of the plugin functions, the core methods retain a clear structure and are easy to comprehend. Further, this approach decreases the coupling between different features: for example, independent features can be modified and tested independently and combined in calculations. In the package, this plugin design has been widely used, for example, to enable molecular point group symmetry, relativistic corrections, solvation effects, density fitting approximations, the use of second-order orbital optimization, different variational active space solvers, and many other features (Figure 8).

3.4 Seamless MPI functionality

The Message Passing Interface (MPI) is the most popular parallel protocol in the field of high performance computing. Although MPI provides high efficiency for parallel programming, it is a challenge to develop a simple and efficient MPI program. In compiled languages, the program must explicitly control data communication according to the MPI communication protocol. The most common design is to activate MPI communication from the beginning and to update the status of the MPI communicator throughout the program. When developing new methods, this often leads to extra effort in code development and debugging. To sustain the simplicity of the PySCF package, we have designed a different mechanism to execute parallel code with MPI. We use MPI to start the Python interpreter as a daemon to receive both the functions and data on the remote nodes. When a parallel session is activated, the master process sends to the remote Python daemons both the functions and

the data. The function is decoded remotely and then executed. This design allows one to develop code mainly in serial mode and to switch to the MPI mode only when high performance is required. Figure 9 shows an example to perform a periodic calculation with and without a parallel session. Comparing to the serial mode invocation, we see that one only has to change the density fitting object to acquire parallel functionality.

4 CONCLUSIONS

Python and its large collection of third party libraries are helping to revolutionize how we carry out and implement numerical simulations. It is potentially much more productive to solve computational problems within the Python ecosystem because it frees researchers to work at the highest level of abstraction without worrying about the details of complex software implementation. To bring all the benefits of the Python ecosystem to quantum chemistry and electronic structure simulations, we have started the open source PySCF project.

PySCF is a simple, light-weight, and efficient computational chemistry program package, which supports ab initio calculations for both molecular and extended systems. The package serves as an extensible electronic structure toolbox, providing a large number of fundamental operations with simple APIs to manipulate methods, integrals, and wave functions. We have invested significant effort to ensure simplicity of use and implementation while preserving competitive functionality and performance. We believe that this package represents a new style of program and library design that will be representative of future software developments in the field.

5 ACKNOWLEDGMENTS

QS would like to thank Junbo Lu and Alexander Sokolov for testing functionality and for useful suggestions for the program package. The development of different components of the PySCF package has been generously supported by several sources. Most of the molecular quantum chemistry software infrastructure was developed with support from the US National

Science Foundation, through grants NSF:CHE:1650436 and NSF:CHE:1657286. The periodic mean-field infrastructure was developed with support from NSF:CHE:1657286. The excited state periodic coupled cluster methods were developed with support from the US Department of Energy, Office of Science, through the grants de-sc0010530 and de-sc0008624. Additional support for the extended system methods has been provided by the Simons Foundation through the Simons Collaboration on the Many Electron Problem, a Simons Investigatorship in Theoretical Physics, the Princeton Center for Theoretical Science, and startup funds from Princeton University and the California Institute of Technology.

References

1. S. van der Walt, S. C. Colbert, and G. Varoquaux, Comput. Sci. Eng. **13**, 22 (2011), ISSN 1521-9615.

2. E. Jones, T. Oliphant, P. Peterson, et al., SciPy: Open source scientific tools for Python (2001–), URL `http://www.scipy.org/`.

3. L. D. Dalcin, R. R. Paz, P. A. Kler, and A. Cosimo, Adv. Water Resour **34**, 1124 (2011), ISSN 0309-1708, new Computational Methods and Software Tools, URL `http://www.sciencedirect.com/science/article/pii/S0309170811000777`.

4. J. M. Turney, A. C. Simmonett, R. M. Parrish, E. G. Hohenstein, F. A. Evangelista, J. T. Fermann, B. J. Mintz, L. A. Burns, J. J. Wilke, M. L. Abrams, et al., WIREs: Comput. Mol. Sci. **2**, 556 (2012), ISSN 1759-0884, URL `http://dx.doi.org/10.1002/wcms.93`.

5. S. R. Bahn and K. W. Jacobsen, Comput. Sci. Eng. **4**, 56 (2002), ISSN 1521-9615.

6. S. P. Ong, W. D. Richards, A. Jain, G. Hautier, M. Kocher, S. Cholia, D. Gunter, V. L. Chevrier, K. A. Persson, and G. Ceder, Computational Materials Science **68**, 314 (2013), ISSN 0927-0256, URL `http://www.sciencedirect.com/science/article/pii/S0927025612006295`.

7. C. R. Jacob, S. M. Beyhan, R. E. Bulo, A. S. P. Gomes, A. W. Götz, K. Kiewisch,

J. Sikkema, and L. Visscher, J. Comput. Chem. **32**, 2328 (2011), ISSN 1096-987X, URL http://dx.doi.org/10.1002/jcc.21810.

8. S. Hirata, J. Phys. Chem. A **107**, 9887 (2003), http://dx.doi.org/10.1021/jp034596z, URL http://dx.doi.org/10.1021/jp034596z.

9. E. Neuscamman, T. Yanai, and G. K.-L. Chan, The Journal of Chemical Physics **130**, 124102 (2009), URL http://scitation.aip.org/content/aip/journal/jcp/130/12/10.1063/1.3086932.

10. M. Saitow, Y. Kurashige, and T. Yanai, The Journal of Chemical Physics **139**, 044118 (2013), URL http://scitation.aip.org/content/aip/journal/jcp/139/4/10.1063/1.4816627.

11. R. P. Muller, PyQuante, Version 1.6.3, URL http://pyquante.sourceforge.net/.

12. J. Enkovaara, C. Rostgaard, J. J. Mortensen, J. Chen, M. Duak, L. Ferrighi, J. Gavnholt, C. Glinsvad, V. Haikola, H. A. Hansen, et al., J. Phys-Condens. Mat. **22**, 253202 (2010), URL http://stacks.iop.org/0953-8984/22/i=25/a=253202.

13. Q. Sun and G. K.-L. Chan, J. Chem. Theory Comput. **10**, 3784 (2014), http://dx.doi.org/10.1021/ct500512f, URL http://dx.doi.org/10.1021/ct500512f.

14. M. A. Marques, M. J. Oliveira, and T. Burnus, Comput. Phys. Commun. **183**, 2272 (2012), ISSN 0010-4655, URL http://www.sciencedirect.com/science/article/pii/S0010465512001750.

15. U. Ekström, L. Visscher, R. Bast, A. J. Thorvaldsen, and K. Ruud, J. Chem. Theory Comput. **6**, 1971 (2010), http://pubs.acs.org/doi/pdf/10.1021/ct100117s, URL http://pubs.acs.org/doi/abs/10.1021/ct100117s.

16. Q. Sun, J. Comput. Chem. **36**, 1664 (2015), ISSN 1096-987X, URL http://dx.doi.org/10.1002/jcc.23981.

17. Q. Sun, arXiv:1610.08423v1 [physics.chem-ph] (2016).

18. H. Koch, A. S. de Mers, T. Helgaker, and O. Christiansen, J. Chem. Phys. **104**, 4157 (1996), URL http://dx.doi.org/10.1063/1.471227.

19. H. Sekino and R. J. Bartlett, Int. J. of Quantum Chem. **26**, 255 (1984), ISSN 1097-461X, URL http://dx.doi.org/10.1002/qua.560260826.

20. M. Nooijen and R. J. Bartlett, J. Chem. Phys. **102**, 3629 (1995), URL http://dx.doi.org/10.1063/1.468592.

21. M. Musiał, S. A. Kucharski, and R. J. Bartlett, J. Chem. Phys. **118**, 1128 (2003), URL http://dx.doi.org/10.1063/1.1527013.

22. P. Knowles and N. Handy, Chem. Phys. Lett. **111**, 315 (1984), ISSN 0009-2614, URL http://www.sciencedirect.com/science/article/pii/000926148485513X.

23. H. J. A. Jensen, P. Jørgensen, and H. Ågren, J. Chem. Phys. **87**, 451 (1987), URL http://dx.doi.org/10.1063/1.453590.

24. H.-J. Werner and P. J. Knowles, J. Chem. Phys. **82**, 5053 (1985), URL http://scitation.aip.org/content/aip/journal/jcp/82/11/10.1063/1.448627.

25. C. Angeli, R. Cimiraglia, S. Evangelisti, T. Leininger, and J.-P. Malrieu, J. Chem. Phys. **114**, 10252 (2001), URL http://dx.doi.org/10.1063/1.1361246.

26. S. Guo, M. A. Watson, W. Hu, Q. Sun, and G. K.-L. Chan, Journal of Chemical Theory and Computation **12**, 1583 (2016), pMID: 26914415, http://dx.doi.org/10.1021/acs.jctc.5b01225, URL http://dx.doi.org/10.1021/acs.jctc.5b01225.

27. S. Sharma and G. K.-L. Chan, J. Chem. Phys. **136**, 124121 (2012), URL http://scitation.aip.org/content/aip/journal/jcp/136/12/10.1063/1.3695642.

28. S. Wouters, W. Poelmans, P. W. Ayers, and D. Van Neck, Comput. Phys. Comm. **185**, 1501 (2014).

29. Q. Sun, J. Yang, and G. K.-L. Chan (2017), in press.

30. G. H. Booth, A. J. W. Thom, and A. Alavi, J. Chem. Phys. **131**, 054106 (2009), URL http://scitation.aip.org/content/aip/journal/jcp/131/5/10.1063/1.3193710.

31. R. E. Thomas, Q. Sun, A. Alavi, and G. H. Booth, J. Chem. Theory Comput. **11**, 5316 (2015), http://dx.doi.org/10.1021/acs.jctc.5b00917, URL http://dx.doi.org/10.1021/acs.jctc.5b00917.

32. S. Sharma, G. Jeanmairet, and A. Alavi, J. Chem. Phys. **144**, 034103 (2016), URL http://scitation.aip.org/content/aip/journal/jcp/144/3/10.1063/1.4939752.

33. G. K.-L. Chan and M. Head-Gordon, J. Chem. Phys. **116**, 4462 (2002), URL http://dx.doi.org/10.1063/1.1449459.

34. G. K.-L. Chan, J. Chem. Phys. **120**, 3172 (2004), URL http://dx.doi.org/10.1063/1.1638734.

35. D. Ghosh, J. Hachmann, T. Yanai, and G. K.-L. Chan, J. Chem. Phys. **128**, 144117 (2008), URL http://dx.doi.org/10.1063/1.2883976.

36. E. A. Salter, G. W. Trucks, and R. J. Bartlett, J. Chem. Phys. **90**, 1752 (1989), URL http://dx.doi.org/10.1063/1.456069.

37. F. Furche and R. Ahlrichs, J. Chem. Phys. **117**, 7433 (2002), URL http://dx.doi.org/10.1063/1.1508368.

38. R. Flores-Moreno, R. J. Alvarez-Mendez, A. Vela, and A. M. Köster, J. Comput. Chem. **27**, 1009 (2006), ISSN 1096-987X, URL http://dx.doi.org/10.1002/jcc.20410.

39. W. Liu and D. Peng, J. Chem. Phys. **131**, 031104 (2009), URL http://scitation.aip.org/content/aip/journal/jcp/131/3/10.1063/1.3159445.

40. L. Cheng, Y. Xiao, and W. Liu, J. Chem. Phys. **130**, 144102 (2009), URL http://dx.doi.org/10.1063/1.3110602.

41. G. Knizia, J. Chem. Theory Comput. **9**, 4834 (2013), http://pubs.acs.org/doi/pdf/10.1021/ct400687b, URL http://pubs.acs.org/doi/abs/10.1021/ct400687b.

42. A. E. Reed, L. A. Curtiss, and F. Weinhold, Chem. Rev. **88**, 899 (1988).

43. G. Schaftenaar and J. Noordik, J. Comput. Aided Mol. Des **14**, 123 (2000), ISSN 1573-4951, URL http://dx.doi.org/10.1023/A:1008193805436.

44. Jmol: an open-source java viewer for chemical structures in 3d., URL http://www.jmol.org/.

45. X. Gonze, F. Jollet, F. A. Araujo, D. Adams, B. Amadon, T. Applencourt, C. Audouze, J.-M. Beuken, J. Bieder, A. Bokhanchuk, et al., Comput. Phys. Commun. **205**, 106 (2016), ISSN 0010-4655, URL http://www.sciencedirect.com/science/article/pii/S0010465516300923.

46. H. van Dam, W. de Jong, E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, and M. Valiev, WIREs: Comput. Mol. Sci. **1**, 888 (2011), ISSN 1759-0884, URL http://dx.doi.org/10.1002/wcms.62.

47. P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G. L. Chiarotti, M. Cococcioni, I. Dabo, et al., J. Phys-Condens. Mat. **21**, 395502 (2009), URL http://stacks.iop.org/0953-8984/21/i=39/a=395502.

48. K. Schwarz and P. Blaha, Computational Materials Science **28**, 259 (2003), ISSN 0927-0256, proceedings of the Symposium on Software Development for Process and Materials Design, URL http://www.sciencedirect.com/science/article/pii/S0927025603001125.

49. R. Dovesi, R. Orlando, A. Erba, C. M. Zicovich-Wilson, B. Civalleri, S. Casassa, L. Maschio, M. Ferrabone, M. De La Pierre, P. D'Arco, et al., Int. J. Quantum Chem. **114**, 1287 (2014), ISSN 1097-461X, URL http://dx.doi.org/10.1002/qua.24658.

50. C. Pisani, M. Schutz, S. Casassa, D. Usvyat, L. Maschio, M. Lorenz, and A. Erba, Phys. Chem. Chem. Phys. **14**, 7615 (2012), URL http://dx.doi.org/10.1039/C2CP23927B.

51. V. Blum, R. Gehrke, F. Hanke, P. Havu, V. Havu, X. Ren, K. Reuter, and M. Scheffler, Comput. Phys. Comm. **180**, 2175 (2009).

52. E. Artacho, E. Anglada, O. Diéguez, J. D. Gale, A. García, J. Junquera, R. M. Martin, P. Ordejón, J. M. Pruneda, D. Sánchez-Portal, et al., J. Phys. Condens. Matter **20**, 064208 (2008).

53. J. Hutter, M. Iannuzzi, F. Schiffmann, and J. VandeVondele, WIREs: Comput. Mol. Sci. **4**, 15 (2014), ISSN 1759-0884, URL http://dx.doi.org/10.1002/wcms.1159.

54. G. H. Booth, T. Tsatsoulis, G. K.-L. Chan, and A. Grüneis, J. Chem. Phys. **145**, 084111 (2016).

55. M. Burkatzki, C. Filippi, and M. Dolg, The Journal of Chemical Physics **126**, 234105 (2007), URL http://dx.doi.org/10.1063/1.2741534.

56. J. McClain, Q. Sun, G. K.-L. Chan, and T. C. Berkelbach (2017), in press.

57. S. Wouters, V. Van Speybroeck, and D. Van Neck, J. Chem. Phys. **145**, 054120 (2016).

58. G. H. Booth, URL https://github.com/ghb24/NECI_STABLE.git.

59. H.-J. Werner, P. J. Knowles, G. Knizia, F. R. Manby, M. Schütz, P. Celani, W. Györffy, D. Kats, T. Korona, R. Lindh, et al., Molpro, version 2015.1, a package of ab initio programs (2015), see http://www.molpro.net.

60. M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, et al., J. Comput. Chem. **14**, 1347 (1993), ISSN 1096-987X, URL http://dx.doi.org/10.1002/jcc.540141112.

61. M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, J. R. Cheeseman, G. Scalmani, V. Barone, G. A. Petersson, H. Nakatsuji, et al., Gaussian 09 Revision A.02, gaussian Inc. Wallingford CT 2016.

62. P. Pulay, Chem. Phys. Lett. **73**, 393 (1980), ISSN 0009-2614, URL http://www.sciencedirect.com/science/article/pii/0009261480803964.

63. P. Pulay, J. Comput. Chem. **3**, 556 (1982), ISSN 1096-987X, URL http://dx.doi.org/10.1002/jcc.540030413.

64. URL https://github.com/numpy/numpy/releases/tag/v1.12.0b1.

Table 1: Features of the PySCF package as of the 1.3 release.

Method	Molecule	Solids	Comments
HF	Yes	Yes	$\sim$ 5000 AOs[b]
MP2	Yes	Yes	$\sim$ 1500 MOs[b]
CCSD	Yes	Yes	$\sim$ 1500 MOs[b]
EOM-CCSD	Yes	Yes	
CCSD(T)	Yes	Yes[a]	$\sim$ 1500 MOs[b]
MCSCF	Yes	Yes[a]	$\sim$ 3000 AOs[b]; 30–50 active orbitals[c]
MRPT	Yes	Yes[a]	$\sim$ 1500 MOs[b]; 30–50 active orbitals[c]
DFT	Yes	Yes	$\sim$ 5000 AOs[b]
TDDFT	Yes	Yes[a]	$\sim$ 5000 AOs[b]
CISD	Yes	Yes[a]	$\sim$ 1500 MOs[b]
FCI	Yes	Yes[a]	$\sim$ (18e, 18o)[b]
Localizer	Yes	No	IAO, NAO, meta-Löwdin Boys, Edmiston-Ruedenberg, Pipek-Mezey
Relativity	Yes	No	ECP and scalar-relativistic corrections for all methods 2-component, 4-component methods for HF and MP2
Gradients	Yes	No	HF, DFT, CCSD, CCSD(T), TDDFT
Hessian	Yes	No	HF and DFT
Property	Yes	No	non-relativistic, 4-component relativistic NMR
Symmetry	Yes	No	D_{2h} and subgroup
AO, MO integrals	Yes	Yes	1-electron, 2-electron integrals
Density fitting	Yes	Yes	HF, DFT, MP2

[a] Only available for Γ-point calculations;

[b] An estimation based on a single SMP node with 128 GB memory without density fitting;

[c] Using an external DMRG or FCIQMC program as active space solver.

```
from pyscf import gto, dft
mol = gto.Mole(atom='N 0 0 0; N 0 0 1.1', basis='ccpvtz')
mf = dft.RKS(mol)
mf.xc = '0.2*HF + 0.08*LDA + 0.72*B88, 0.81*LYP + 0.19*VWN'
mf.kernel()
```

Figure 1: Example to define a custom exchange-correlation functional for a DFT calculation.

```
from pyscf import gto, scf, mcscf, dmrgscf
mol = gto.Mole(atom='N 0 0 0; N 0 0 1.1', basis='ccpvtz')
mf = scf.RHF(mol).run()
mc = mcscf.CASSCF(mf, 8, 10)  # 8o, 10e
mc.fcisolver = dmrgscf.DMRGCI(mol)
mc.kernel()
```

Figure 2: Example to enable the DMRG solver in a CASSCF calculation.

```
from pyscf import gto, scf, lo, tools
mol = gto.Mole(atom=open('c60.xyz').read(),
               basis='ccpvtz')
mf = scf.RHF(mol).run()
orb = lo.Boys(mol).kernel(mf.mo_coeff[:,:180])
tools.molden.from_mo(mol, 'c60.molden', orb)

# Invoke Jmol to plot the orbitals
with open('c60.spt', 'w') as f:
  f.write('load c60.molden; isoSurface MO 002;\n')
import os
os.system('jmol c60.spt')
```

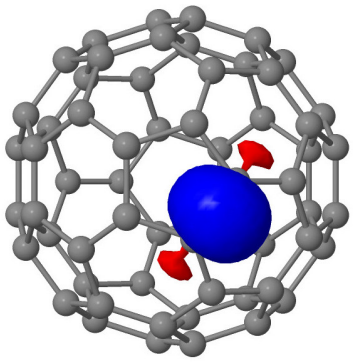

Figure 3: Example to generate localized orbitals and to plot them in Jmol.

```
from pyscf import gto
mol = gto.Mole(atom='N 0 0 0; N 0 0 1.1', basis='ccpvtz')
a = mol.intor('cint1e_nuc_sph')  # nuclear attraction as a giant array
a = mol.intor('cint2e_sph')      # 2e integrals as a giant array
a = mol.intor_by_shell('cint2e_sph', (0,0,0,0))  # (ss|ss) of first N atom
```

Figure 4: Example to access AO integrals.

```python
# 10-site Hubbard model at half-filling with U/t = 4
import numpy as np
from pyscf import gto, scf, ao2mo, cc
mol = gto.Mole(verbose=4)
mol.nelectron = n = 10
t, u = 1., 4.

mf = scf.RHF(mol)
h1 = np.zeros((n,n))
for i in range(n-1):
    h1[i,i+1] = h1[i+1,i] = t
mf.get_hcore = lambda *args: h1
mf.get_ovlp = lambda *args: np.eye(n)
mf._eri = np.zeros((n,n,n,n))
for i in range(n):
    mf._eri[i,i,i,i] = u
# 2e Hamiltonian in 4-fold symmetry
mf._eri = ao2mo.restore(4, mf._eri, n)
mf.run()
cc.CCSD(mf).run()
```

Figure 5: Example to use a custom Hamiltonian.

```python
import numpy as np
from pyscf import gto, scf
bond = np.arange(0.8, 5.0, .1)
dm_init = None
e_hf = []
for r in reversed(bond):
  mol = gto.Mole(atom=[['N', 0, 0, 0],
                       ['N', 0, 0, r]],
                 basis='ccpvtz')
  mf = scf.RHF(mol).run(dm_init)
  dm_init = mf.make_rdm1()
  e_hf.append(mf.e_tot)

from matplotlib import pyplot
pyplot.plot(bond, e_hf[::-1])
pyplot.show()
```

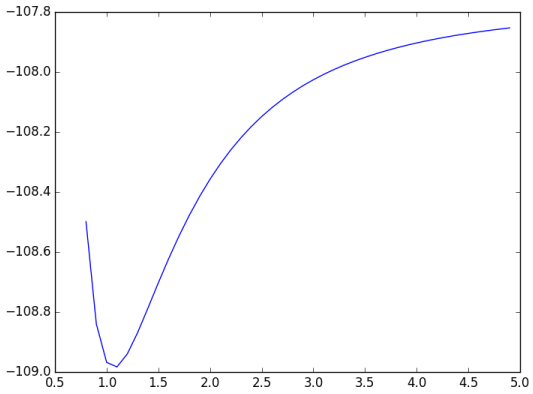

Figure 6: Using Python to combine the calculation and data post-processing in one script.

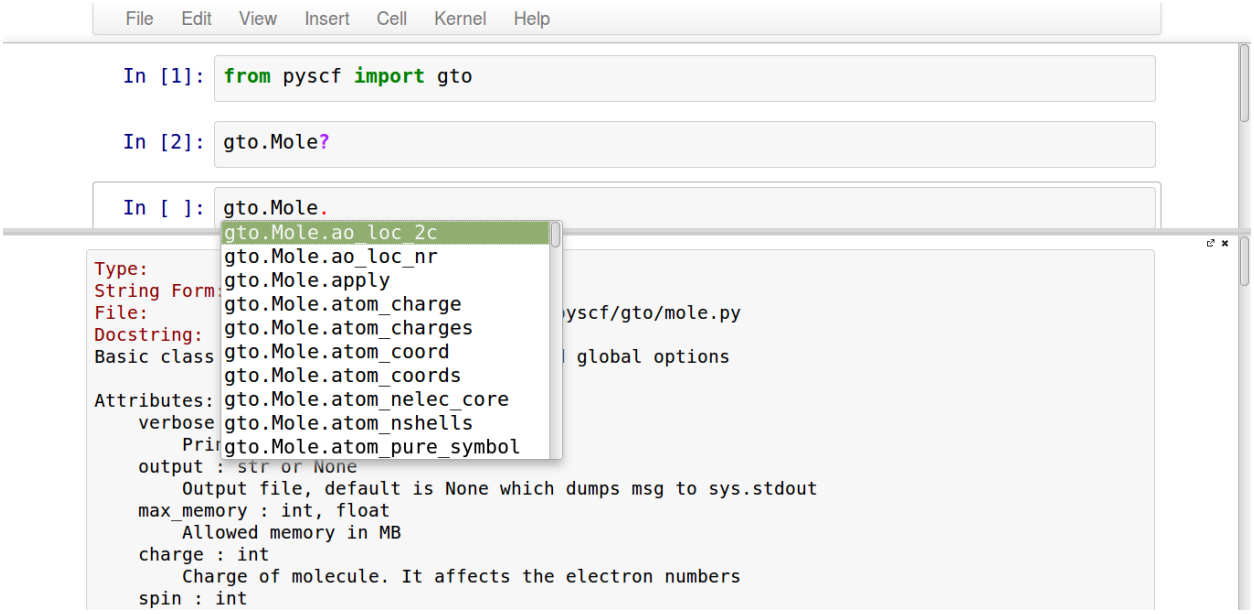

Figure 7: Accessing documentation within the IPython shell. The question mark activates the documentation window in the bottom area. The pop-up menu for code auto-completion is triggered by the <Tab> key.

```
from pyscf import gto, scf
mol = gto.Mole(atom='N 0 0 0; N 0 0 1.2', basis='ccpvtz')
mf = scf.newton(scf.sfx2c(scf.density_fit(scf.RHF(mol)))).run()
```

Figure 8: Example to use plugins in PySCF. The mean-field calculation is decorated by the density fitting approximation, X2C relativistic correction and second order SCF solver.

```python
# Serial mode
# run in cmdline:
# python input.py
from pyscf.pbc import gto, scf
from pyscf.pbc import df
cell = gto.Cell()
cell.atom = 'H 0 0 0; H 0 1.1 1.1'
cell.basis = 'ccpvdz'
# unit cell lattice vectors
cell.a = '2 0 0; 0 2 0; 0 0 2'
# grid for numerical integration
cell.gs = [10,10,10]
mf = scf.RHF(cell)
mf.with_df = df.DF(cell)
mf.kernel()
```

```python
# MPI mode
# run in cmdline:
# mpirun -np 4 python input.py
from pyscf.pbc import gto, scf
from mpi4pyscf.pbc import df
cell = gto.Cell()
cell.atom = 'H 0 0 0; H 0 1.1 1.1'
cell.basis = 'ccpvdz'
# unit cell lattice vectors
cell.a = '2 0 0; 0 2 0; 0 0 2'
# grid for numerical integration
cell.gs = [10,10,10]
mf = scf.RHF(cell)
mf.with_df = df.DF(cell)
mf.kernel()
```

Figure 9: Comparison of the input script for serial-mode and MPI-mode calculations. Except for the module to import, the MPI parallel mode takes exactly the same input as the serial mode.