`Fanpy`: A Python Library for Prototyping Multideterminant Methods in *Ab Initio* Quantum Chemistry

Taewon D. Kim[1,2] M. Richer[1] Gabriela Sánchez-Díaz[1]
Ramón Alain Miranda-Quintana[2] Toon Verstraelen[3]
Farnaz Heidar-Zadeh[4] Paul W. Ayers[1]

March 7, 2023

[1]Department of Chemistry and Chemical Biology, McMaster University, Hamilton, Ontario, L8S-4L8, Canada
[2]Department of Chemistry and Quantum Theory Project, University of Florida, Gainesville, FL 32603, USA
[3]Center for Molecular Modeling (CMM), Ghent University, Technologiepark-Zwijnaarde 46, B-9052, Zwijnaarde, Belgium
[4]Department of Chemistry, Queen's University, Kingston, Ontario, K7L-3N6, Canada

Abstract

`Fanpy` is a free and open-source Python library for developing and testing multideterminant wavefunctions and related *ab initio* methods in electronic structure theory. The main use of `Fanpy` is to quickly prototype new methods by making it easier to convert the mathematical formulation of a new wavefunction ansätze to a working implementation. `Fanpy` is designed based on our recently introduced Flexible Ansatz for N-electron Configuration Interaction (FANCI) framework, where multideterminant wavefunctions are represented by their overlaps with Slater determinants of orthonormal spin-orbitals. In the simplest case, a new wavefunction ansatz can be implemented by simply writing a function for evaluating its overlap with an arbitrary Slater determinant. `Fanpy` is modular in both implementation and theory: the wavefunction model, the system's Hamiltonian, and the choice of objective function are all independent modules. This modular structure makes it easy for users to mix and match different methods and for developers to quickly explore new ideas. `Fanpy` is written purely in Python with standard dependencies, making it accessible for various operating systems. In addition, it adheres to principles of modern software development, including comprehensive documentation, extensive testing, quality assurance, and continuous integration and delivery protocols. This article is considered to be the official release notes for the `Fanpy` library.

1 What is `Fanpy`?

`Fanpy` is a free, open-source, and cross-platform Python 3 library for *ab initio* electronic structure calculations. The key innovative aspect of this library is the adoption of the Flexible Ansatz for N-electron Configuration Interaction (FANCI) mathematical framework[1]. By adopting this framework, *ab initio* electronic structure methods are represented as a collection of four parts, each of which is developed as an independent module of `Fanpy`: (a) the (multideterminant) wavefunction model (b) the system Hamiltonian represented by its one- and two-electron integrals (c) an equation (or system of equations) that is equivalent to the Schrödinger equation, (d) an algorithm for optimizing the objective function(s). Section 4 details the features of each module, though the main advantage of `Fanpy` is that new features can be easily implemented in each module.

In the FANCI framework, a wavefunction is represented by its overlap with a set of reference states (e.g., Richardson eigenfunctions[2, 3]). In `Fanpy`, Slater determinants are explicitly used as the reference states and the wavefunction ansätze of interest are multideterminant wavefunctions with parameterized coefficients,

$$|\Psi_{\text{FANCI}}\rangle = \sum_{\mathbf{m} \in S_{\mathbf{m}}} f(\mathbf{m}, \vec{P})\, |\mathbf{m}\rangle \tag{1}$$

where

$$|\mathbf{m}\rangle = |m_1 m_2 \dots m_{N-1} m_N\rangle = a^{\dagger}_{m_1} a^{\dagger}_{m_2} \dots a^{\dagger}_{m_{N-1}} a^{\dagger}_{m_N} \,|\rangle \tag{2}$$

denotes a Slater determinant, $S_{\mathbf{m}}$ is the set of Slater determinants included in the wavefunction, and f is a function that determines the coefficient of each Slater determinant, $\mathbf{m}$, using the parameters, $\vec{P}$. Note that f is simply the overlap of the parameterized wavefunction with the Slater determinant,

$$f(\mathbf{m}, \vec{P}) = \langle \mathbf{m} | \Psi_{\text{FANCI}} \rangle \tag{3}$$

Similarly, the Hamiltonian is expressed in terms of its CI matrix elements,

$$\langle \mathbf{m} | \hat{H} | \mathbf{n} \rangle = \langle \mathbf{m} | \sum_{ij} h_{ij} a^{\dagger}_i a_j + \frac{1}{2} \sum_{ijkl} g_{ijkl} a^{\dagger}_i a^{\dagger}_j a_l a_k | \mathbf{n} \rangle \tag{4}$$

The objective functions supported by `Fanpy` combine the overlaps and the CI matrix elements to approximate the Schrödinger equation.

2 About `Fanpy`

The source code of `Fanpy` is maintained on GitHub; see `https://github.com/theochem/fanpy`, and a comprehensive documentation, including useful tests, scripts, and examples, is hosted on Read the Docs; see `https://fanpy.readthedocs.io/en/latest/index.html`. In this article, we list the key features and capabilities of `Fanpy` in section 4 to establish its philosophy and framework, and exemplify them in section 5.

3 Why `Fanpy`?

Many quantum chemistry packages support computations with multideterminant methods. Most of these packages (e.g. Gaussian[4] and MolPro[5]) are closed-source, making it nearly impossible to develop new methods without special permissions. Even for packages whose source code is available, the code is often monolithic, making it difficult to implement new fundamental methods without thoroughly understanding nearly the entire code base. Moreover, the low-level code is often highly optimized, abstracts away critical components of *ab initio* methods, and not suitable for subsequent modification. Such code often remain unchanged for decades, has little documentation, and rarely follows modern software development principles. Though some packages try to address this issue, the development of post-Hartree-Fock (post-HF) methods remains difficult. For example, Psi4Numpy[6] is a collection of Python scripts and Jupyter notebooks that implement several post-HF methods using Psi4 to generate necessary inputs, such as CI matrices and one- and two-electron integrals. It is an excellent package for learning about standard quantum chemistry methods or implementing embellishments of standard methods. However, developing a novel method (e.g. a new wavefunction ansatz) would likely require all related processes to be implemented in or interfaced to Psi4, which requires a thorough understanding of the Psi4 package as a whole[7]. The Ghent Quantum Chemistry Package (GQCP) is quite modular, and has a convenient Python interface that exposes some code features/options, but one must still delve deep into the source code to construct bespoke wavefunction models or specific active/projection spaces. `Fanpy` is distinct from GQCP as it compromises performance (to a great extent) in return for flexibility and ease of prototyping[8]. In particular, we are aware of no other software that makes it quick and easy to implement a new objective function and its associated optimization algorithm, implement the action of the Hamiltonian on a new wavefunction ansatz, and select alternative active and/or projection spaces.

Due to the difficulty of developing new methods in these legacy codes, we developed our own Helpful Open-Source Research TOol for N-electron systems (`HORTON`)[9]. The first two versions of `HORTON` were also monolithic, and even wavefunction models that were simple on paper were difficult to implement. These are resolved in `HORTON3` with separate modules for `IOData`[10] (input/output), `Grid` (numerical integration), `GBasis` (Gaussian-basis-set evaluation and integrals), `GOpt` (geometry optimization), and `Meanfield` (self-consistent field calculations). `Fanpy` is the correlated wavefunction module of `HORTON3`. Pybest is an alternative approach that meets some of the objectives of `Fanpy`, based on a (heavily revised, modernized, and extensively extended) version of `HORTON2`.[11] Like `Fanpy`, Pybest supports unconventional wavefunction methods (e.g., various seniority-zero methods).

`Fanpy` was envisioned as a development tool for new correlated-wavefunction methods; the goal is to help researchers quickly implement and test their ideas. Towards this goal, `Fanpy` is designed to be modular and general. Its modularity helps to isolate and minimize the amount of code that needs to be understood, and perhaps modified, to implement a new method. For example, implementing a new wavefunction ansatz requires modifying only the wavefunction module, and does not require explicit consideration of how the Hamiltonian will act upon that wavefunction nor of how the orbitals and parameters within the wavefunction will be optimized. The modules of `Fanpy` are designed to be as general as possible, so that features from one module are compatible with features from the other modules. The compatibility between the modules ensures that any developed method (e.g. a wavefunction ansatz) can be used in conjunction with the other methods (e.g. orbital optimization, model Hamiltonians, the projected Schrödinger equation, etc). We provide comprehensive documentation and examples to further aid the development of new methods in `Fanpy`.

4 Features of `Fanpy`

We display various features of `Fanpy` by discussing each module and their intended purposes:

- The wavefunction module is developed in accordance with the FANCI framework[1]. In the FANCI framework, the wavefunction is entirely represented by its overlaps with Slater determinants built from orthonormal orbitals. Similarly, each wavefunction in `Fanpy` is defined by its parameters and a function that returns an overlap for the given Slater determinant. The overlap can be provided as a standalone function or defined within a class structure, templated from an abstract base class. The following wavefunctions have already been implemented: configuration interaction (CI) with single and double excitations (CISD)[12]; doubly-occupied configuration interaction (DOCI)[13–16]; full CI[17]; selected CI wavefunctions with a user-specified set of Slater determinants; antisymmetrized products of geminals (APG)[18–28]; antisymmetrized products of geminals with disjoint orbital sets (APsetG)[29]; antisymmetrized product of interacting geminals (APIG)[29–57]; antisymmetric product of 1-reference-orbital interacting geminals (AP1roG; equivalent to pair-coupled-cluster doubles)[58]; antisymmetric product of rank-two interacting geminals (APr2G)[2]; determinant ratio wavefunctions[1]; antisymmetrized products of tetrets (4-electron wavefunctions)[1]; matrix product states (MPS)[59]; neural network wavefunctions; coupled-cluster (CC) with arbitrary excitations (including, but not limited to, CCSD, CCSDT, and CC with seniority-specific excitations)[1, 60–65], geminal coupled-cluster wavefunctions[37–39, 41, 58], generalized CC, and seniority-increasing CC. We also support these wavefunctions with nonorthogonal orbitals, and linear combinations of any of the aforementioned wavefunctions.

- The Hamiltonian module contains Hamiltonians commonly used in electronic structure theory. Similar to the wavefunctions, each Hamiltonian in `Fanpy` is defined by its representation in orbital basis set (i.e. one- and two-electron integrals) and a function that returns the integral of the Hamiltonian with respect to the given Slater determinants. The following Hamiltonians have already been implemented: the electronic Hamiltonian in the restricted, unrestricted, and generalized basis; the seniority-zero electronic Hamiltonian[66]; and the Fock operator in the restricted basis. In addition, the Pariser-Parr-Pople[67–70], Hubbard[70, 71], Hückel[70, 72], Ising[70, 73, 74], Heisenberg[70, 74, 75], and Richardson[76, 77] model Hamiltonians are available as restricted electronic Hamiltonians through the ModelHamiltonian GitHub repository[78]. Orbital optimization is available if a function returning the derivative with respect to orbital rotation parameters is provided. At the moment, only restricted electronic Hamiltonians support orbital optimization.

- The objective module is responsible for combining the wavefunction and the Hamiltonian to form an equation or a system of equations that represents the Schrödinger equation. In `Fanpy`, the objective function can be the variational optimization of the expectation value of the energy[79–83], the projected Schrödinger equation[29, 41, 54], or a local energy expression to be sampled (as in variational quantum Monte Carlo)[84–89].

- The solver module contains algorithms that optimize/solve the equations from the objective module. It supports optimizers from `SciPy`[90], which includes constrained/unconstrained local/global optimizers for multivariate scalar functions (i.e. energy) and algorithms for solving nonlinear least-squares problems and for finding roots of a system of nonlinear equations (i.e., projected Schrödinger equation). For CI wavefunctions, we also support brute-force eigenvalue decomposition. In addition, `Fanpy` interfaces to several algorithms for derivative-free global optimization problems including the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) algorithm[91] from `pycma`[92] and algorithms using decision trees and Bayesian optimization from `scikit-optimize`[93]. At the moment, no in-house optimization algorithms specialized for electronic structure theory problems are included. However, `Fanpy`'s modular design makes it easier to develop sophisticated domain-specific optimization algorithms. The objective module provides high-level control over the parameters involved in the optimization (e.g., active and frozen parameters) and can be changed dynamically throughout the optimization process. These parameters can be saved as a checkpoint throughout the optimization. (The default is to checkpoint at each function evaluation.) Furthermore, the objective module provides flexibility to add additional parameters (e.g., model hyperparameters) and to add nonlinear constraints to the Projected Schrödinger equation.

- The `tool` module provides various utility functions used throughout the `Fanpy` package. Though some tools have specialized uses, the tools for manipulating and generating Slater determinants are used frequently throughout `Fanpy`. These tools are essential when developing methods in `Fanpy` because Slater determinants are the common language of the independent modules. The `slater` module provides functions for manipulating Slater determinants and converting them from one form to another. Within `Fanpy`, Slater determinants are represented as a binary number, where the positions of 1's are the indices of the occupied spin-orbitals. The `slater` module can, for example, provide the occupied spin orbital indices from the given Slater determinant. The `sd_list` module provides easy ways to generate Slater determinants of the desired characteristics (e.g. order of excitation from ground state, spin, seniority). This module is frequently used to construct the projection space by which the objective function is evaluated. In addition, the `tool` module provides wrappers to other modules of `HORTON3` and other quantum chemistry software, including `Gaussian`[4], `PySCF`[94], and `Psi4`[7]. These programs can then be used to generate one- and two-electron integrals for `Fanpy`.

5 Examples

These examples are based on the version 1.0 of `Fanpy`. Please refer to the `Fanpy` website for the latest documentation and examples.

Running a calculation: `Fanpy` can be used directly as a Python library (e.g., in a script or Jupyter notebook) or through its command-line utility,

```
fanpy_run_calc -h
```

For ease of use, `fanpy_run_calc` provides limited access to `Fanpy`'s features using sensible default settings. However, it is recommended to create and execute a Python script/notebook because it provides a transparent record of the calculation (and its settings) and the full range of `Fanpy`'s features. For assistance in creating a Python script, `Fanpy` provides a command-line tool, `fanpy_make_script`. This tool creates a script from the given specifications, which can then be modified if a desired feature is not covered by the command-line tool.

The following Python script gives an example of AP1roG calculation for oxygen molecule in a double zeta basis set:

```python
import numpy as np
from fanpy.wfn.geminal.ap1rog import AP1roG
```

```python
from fanpy.ham.restricted_chemical import RestrictedMolecularHamiltonian
from fanpy.eqn.projected import ProjectedSchrodinger
from fanpy.solver.system import least_squares
from fanpy.tools.sd_list import sd_list

nelec = 16

# Hamiltonian
oneint = np.load('one_oxygen.npy')
twoint = np.load('two_oxygen.npy')
ham = RestrictedMolecularHamiltonian(oneint, twoint)

# Wavefunction
wfn = AP1roG(nelec, ham.nspin)

# Projection space of first and second order excitation
pspace = sd_list(nelec, ham.nspin, exc_orders=[2], seniority=0)

# Projected Schrodinger Equation
eqns = ProjectedSchrodinger(wfn, ham, pspace=pspace)

# Solve
results = least_squares(eqns)
print('AP1roG electronic energy (Hartree):', results['energy'])
```

Since `Fanpy` targets post-HF methods, the orbitals (and the corresponding system specific information) must be provided in the form of one- and two-electron integrals. The one- and two-electron integrals must be provided as two- and four-dimensional numpy arrays, respectively, whose indices are in the same order as the integrals in the physicists' notation. To generate the integrals from a single-determinant calculation, `Fanpy` provides wrappers for `HORTON`, `PySCF`, and `Psi4` via the `fanpy.tools.wrapper` module. The Gaussian `.fchk` file can be converted into `.npy` file using the `HORTON` wrapper, which will also compute the required one- and two-electron integrals.

Implementing a wavefunction: New wavefunctions can be implemented in `Fanpy` by making a subclass of the wavefunction base class or by providing the overlap function to a utility function. The subclass requires the method `get_overlap` to be defined.

As a simple example, recall that expansion of a Slater determinant of nonorthogonal orbitals in orthogonal Slater determinants is given by:

$$\begin{aligned} |\Psi\rangle &= \prod_{i=1}^{N} \sum_{j}^{2K} C_{ij} a_j^\dagger |\theta\rangle \\ &= \sum_{\mathbf{m}} |C(\mathbf{m})|^- |\mathbf{m}\rangle \end{aligned} \tag{5}$$

In `Fanpy`, this corresponds to:

```python
import numpy as np
from fanpy.wfn.base import BaseWavefunction
from fanpy.tools.slater import occ_indices

class NonorthogonalSlaterDeterminant(BaseWavefunction):
    def get_overlap(self, sd, deriv=None):
        # get indices of the occupied spin orbitals
        occs = occ_indices(sd)
        # reshape the parameters
        # NOTE: parameters are stored as a one-dimensional array by default
        params = self.params.reshape(self.nelec, self.nspin)
        # compute the overlap
        if deriv is None:
            return np.linalg.det(params[:, occs])

        # compute the derivative of the overlap
        output = np.zeros(params.shape)
        for deriv_row in range(self.nelec):
            for j, deriv_col in enumerate(occs):
                # compute the sign associated with Laplace formula
                sign = (-1)**(deriv_row + j)
                # get rows and columns with the appropriate row/column removed
                row_inds = np.arange(self.nelec)
                row_inds = row_inds[row_inds != deriv_row]
                col_inds = occs[occs != deriv_col]
                # compute minors (determinant after removing row and column)
                minor = np.linalg.det(params[row_inds[:, None], col_inds[None, :]])
                output[deriv_row, deriv_col] = sign * minor
        # derivative is returned as a flattened array
        # deriv contains the indices of the parameters with respect to which
        # the overlap is derivatized
        return output.ravel()[deriv]
```

The method `get_overlap` returns the overlap of the given Slater determinant when `deriv=None` and returns its gradient with respect to the parameters specified by `deriv` otherwise (`deriv` is a one-dimensional numpy array of parameter indices). Further details on the API of `get_overlap` are provided in the online documentation. Unlike the wavefunctions already implemented in `Fanpy`, this wavefunction does not have default initial parameters, which means that they must be supplied when instantiating the wavefunction. For example, the following code block shows how to initialize to the ground-state (orthogonal) Slater determinant.

```
from fanpy.tools.slater import ground, occ_indices

# get indices of the HF ground state
ground_indices = occ_indices(ground(16, 36))
# initial parameters (only contain the occupied orbitals in HF ground state)
init_params = np.zeros((16, 36))
init_params[np.arange(16), ground_indices] = 1

wfn = NonorthogonalSlaterDeterminant(16, 36)
# assign parameters
wfn.assign_params(init_params)
```

Alternatively, the wavefunction can be constructed using the utility function: `wfn_factory`.

```
import numpy as np
from fanpy.wfn.utils import wfn_factory

def olp(sd, params):
    occs = occ_indices(sd)
    # NOTE: Since the only information available come from the arguments sd and
    # params, additional information that would otherwise be stored as
    # instance's attributes and properties must be explicitly defined
    nelc = 16
    nspin = 36
    # reshape the parameters
    params = params.reshape(nelc, nspin)
    return np.linalg.det(params[:, occs])

def olp_deriv(sd, params):
    occs = occ_indices(sd)
    # hardcode essential information
    nelc = 16
    nspin = 36
    # reshape the parameters
    params = params.reshape(nelc, nspin)
    # same as above except replace self.nelec with nelec
    # ...
    # NOTE: the overlap is derivatized with respect to all wavefunction
    # parameters unlike above
    return output.ravel()

```

```python
# number of electrons
nelec = 16
# number of spin orbitals
norbs = 36
# construct wavefunction using the initial parameters
wfn = wfn_factory(olp, olp_deriv, 16, 36, init_params)
```

It is recommended to implement wavefunctions using the class structure because it helps make the code cleaner by limiting repetitions and makes the code easier to unit test by breaking it into smaller pieces. For a quick and dirty implementation, however, the utility function may be easier.

Implementing a Hamiltonian: Similar to the wavefunction, new Hamiltonians can be implemented in `Fanpy` by making a subclass of the Hamiltonian base class or by passing a function that evaluates the integrals to a utility function. In addition to the general Hamiltonian base class, `Fanpy` provides base classes according to the type of orbitals used in the Hamiltonian: restricted, unrestricted, and generalized. The subclass to the orbital specific Hamiltonian base class requires the method `integrate_sd_sd` to be defined.

For example, to implement the Hückel Hamiltonian:[70]

$$\begin{aligned}\hat{H} &= \sum_{ij}\sum_{\sigma} h_{ij} a^\dagger_{i\sigma} a_{j\sigma} \\ h_{ij} &= \begin{cases} \alpha_i & \text{if } i = j \\ \beta_{ij} & \text{if spatial orbitals } i \text{ and } j \text{ belong to atoms that participate in a bond} \\ 0 & \text{else} \end{cases}\end{aligned} \tag{6}$$

```python
from fanpy.ham.base import BaseHamiltonian
from fanpy.tools import slater

class HuckelHamiltonian(BaseHamiltonian):
    def __init__(self, one_int):
        # NOTE: provided integrals correspond to spatial orbitals
        self.one_int = one_int

    @property
    def nspin(self):
        return self.one_int.shape[0] * 2

    def integrate_sd_wfn(self, wfn, sd, wfn_deriv=None, ham_deriv=None):
        # use the default method except only the first order excitations are used
        return super().integrate_sd_wfn(
            wfn, sd, wfn_deriv=wfn_deriv, ham_deriv=ham_deriv, orders=(1,)
```

```python
        )

    def integrate_sd_sd(self, sd1, sd2, deriv=None, components=False):
        # get the difference of the Slater determinants (i.e. which orbitals are
        # occupied in one determinant but not in the other)
        diff_sd1, diff_sd2 = slater.diff_orbs(sd1, sd2)
        # derivative not supported here
        if deriv:
            raise NotImplementedError
        # if order of excitation between the two Slater determinants is two or greater
        if len(diff_sd1) >= 2 or len(diff_sd2) >= 2:
            return 0.0
        # if two Slater determinants do not have the same number of electrons
        if len(diff_sd1) != len(diff_sd2):
            return 0.0
        # if two Slater determinants are the same
        if len(diff_sd1) == 0:
            # get the indices of the spatial orbitals that correspond to the
            # occupied spin orbitals
            shared_alpha_sd, shared_beta_sd = slater.split_spin(
                slater.shared_sd(sd1, sd2), self.nspatial
            )
            shared_alpha = slater.occ_indices(shared_alpha_sd)
            shared_beta = slater.occ_indices(shared_beta_sd)
            # sum over the occupied orbitals
            output = np.sum(self.one_int[shared_alpha, shared_alpha])
            output += np.sum(self.one_int[shared_beta, shared_beta])
            return output
        # if two Slater determinants are different by one-electron excitation
        # get indices of the spatial orbitals
        spatial_ind1 = slater.spatial_index(diff_sd1[0], self.nspatial)
        spatial_ind2 = slater.spatial_index(diff_sd2[0], self.nspatial)
        return self.one_int[spatial_ind1, spatial_ind2]
```

Though it is not necessary, the subclass defines `integrate_sd_wfn` to specify that the Hamiltonian only contains one-body operators. By default, `integrate_sd_wfn` assumes that the Hamiltonian contains one- and two-body operators.

Alternatively, the Hamiltonian can be constructed using the utility function

```python
from fanpy.ham.utils.factory import ham_factory
from fanpy.tools import slater

```

```
def integrate_sd_sd(sd1, sd2, one_int):
    diff_sd1, diff_sd2 = slater.diff_orbs(sd1, sd2)
    nspatial = one_int.shape[0]
    # same as above except replace self.one_int with one_int
    # and self.nspatial with nspatial
    # ...

# construct Hamiltonian for 36 electrons
ham = ham_factory(integrate_sd_sd, oneint, 36, orders=(1,))
```

Again, using the class structure is encouraged because its structure can be cleaner and more transparent and because it provides finer control over the Hamiltonian. For example, if `integrate_sd_wfn` is directly implemented rather than `integrate_sd_sd`, then `integrate_sd_wfn` can be vectorized over the given Slater determinant and its excitations associated with the application of the Hamiltonian. When the derivative of the integral is not provided, orbital optimization is only available through relatively inefficient gradient-free optimization algorithms, such as CMA-ES.

Since the Hückel Hamiltonian is defined by its one-electron integrals, this class can be used to describe any Hamiltonian with only one-body operators. The integrals for the Hückel Hamiltonian (and other model Hamiltonians) can be generated using the ModelHamiltonian GitHub repository[78].

Implementing an Objective: New objectives can be implemented in `Fanpy` by making a subclass of the objective base class. The subclass requires the method `objective` to be defined. To use the gradient (or Jacobian) in the optimization algorithm, the subclass must also contain the method `gradient` (or `jacobian`). The Jacobian is used to solve the system of nonlinear equations (projected Schrödinger equation). These objectives can then be solved using the appropriate methods in the `solver` module. For example, the energy related objectives can be solved via minimization and the projected Schrödinger equation related objectives can be solved via root-finding and least-squares algorithms.

For example, consider the local energy used in the orbital-space variational Quantum Monte Carlo[86],

$$E_L = \sum_i \frac{\langle \Phi_i | \hat{H} | \Psi \rangle}{\langle \Phi_i | \Psi \rangle} \tag{7}$$

where the Slater determinant, Φ_i, is sampled according to the distribution $p(\Phi_i) = \frac{\langle \Psi | \Phi_i \rangle^2}{\sum_k \langle \Psi | \Phi_j \rangle^2}$. This corresponds to:

```
from fanpy.eqn.base import BaseSchrodinger

class LocalEnergy(BaseSchrodinger):
    def __init__(self, wfn, ham, pspace, param_selection=None):
        super().__init__(wfn, ham, param_selection=param_selection)
```

```
        # param_selection is used to select the parameters that are active
        # throughout the optimization
        self.pspace = pspace
        # pspace is the list of Slater determinants from which local energy is
        # computed

    @property
    def num_eqns(self):
        # number of equations is used to differentiate objectives in the solver
        return 1

    def objective(self, params):
        # assign (active) parameters to the respective wavefunction and
        # Hamiltonian
        # note that params is always flattened (1-dimensional) for compatibility
        # with solvers
        self.assign_params(params)
        output = 0.0
        for sd in self.pspace:
            output += self.ham.integrate_sd_wfn(sd, self.wfn) / self.wfn.get_overlap(sd)
        return output

    def gradient(self, params):
        self.assign_params(params)
        # note that gradient of the objective is also flattened (1-dimensional)
        output = np.zeros(params.size)
        for sd in self.pspace:
            # indices of the wavefunction parameters that are active
            wfn_inds_component = self.indices_component_params[self.wfn]
            if wfn_inds_component.size > 0:
                # indices of the objective parameters that correspond to the
                # wavefunction
                wfn_inds_objective = self.indices_objective_params[self.wfn]

                # differentiate local energy with respect to wavefunction parameters
                output[wfn_inds_objective] += (
                    self.ham.integrate_sd_wfn(sd, self.wfn, wfn_deriv=wfn_inds_component)
                    / self.wfn.get_overlap(sd)
                )
                output[wfn_inds_objective] -= (
                    self.ham.integrate_sd_wfn(sd, self.wfn)
```

```
                    * self.wfn.get_overlap(sd, deriv=wfn_inds_component)
                    / self.wfn.get_overlap(sd) ** 2
                )
            # indices of the Hamiltonian parameters that are active
            # Used when hamiltonian has parameters to optimize (e.g. orbital
            # optimization)
            ham_inds_component = self.indices_component_params[self.ham]
            if ham_inds_component.size > 0:
                # indices of the objective parameters that correspond to the
                # hamiltonian
                ham_inds_objective = self.indices_objective_params[self.ham]

                # differentiate local energy with respect to Hamiltonian parameters
                output[ham_inds_objective] += (
                    self.ham.integrate_sd_wfn(sd, self.wfn, ham_deriv=ham_inds_component)
                    / self.wfn.get_overlap(sd)
                )
        return output
```

Though it is not required, providing the indices in the gradient ensures that users can specify the parameters that are active during the optimization via the attribute `param_selection`.

6 Frequently Asked Questions

Who is `Fanpy` for? `Fanpy` was designed to be used by developers of post-HF methods, especially those interested in new multireference wavefunction ansätze. To use our library, extensive programming experience is not necessary, because: `Fanpy`'s modular design and extensive documentation make it easy to understand and extend the existing methods and base classes. The base classes serve as templates to help ensure that the developed method fits together with the rest of `Fanpy` seamlessly. Developers with programming experience but a limited background in post-HF methods should have an easier time understanding the code because the methods are documented with the corresponding equations (and their derivations) and are implemented in a simple and straight-forward fashion.

What is the mission of `Fanpy`? Our goal is to develop a platform where developers of new *ab initio* methods can quickly implement and test their ideas. We hope to make it easier for researchers—whether they are seasoned professors or new graduate students—to test their ideas without being burdened by undocumented code conventions, untested source code, mysterious equations, or cumbersome installation processes.

What does `Fanpy` do? As elaborated in Sections 4 and 5, `Fanpy` provides independent modules that facilitate the development of new multideterminant wavefunctions, Hamiltonians, representations of the Schrödinger equation (objective functions), and optimization algorithms. We designed these modules to be compatible with one another so that researchers can easily customize their calculations and experiment with different combinations of methods and algorithms.

What are the limits of `Fanpy`? At present, `Fanpy` is not designed for high performance. In fact, its performance was often deliberately compromised to prioritize ease of use and development. For accessibility, `Fanpy` was written in pure Python even though other languages, such as C++ and Julia, may be better suited for high-performance parallel computing. Moreover, while `Fanpy`'s modular design is important for its extendibility and customizability, it prevents some types of algorithmic improvements. Since a method in its early stages of development is often intractably expensive, calculations in `Fanpy` are often limited to small model systems with small basis sets. Some of the more efficient methods (e.g. AP1roG, which could be extended to thousands of electrons in an efficient implementation) are limited to about 100 electrons in `Fanpy`. Consistent with the overall mission of `HORTON3`, therefore, `Fanpy` should be viewed as a research tool that allows developers to quickly implement and test their ideas, rather than a comprehensive quantum chemistry suite that can simulate large chemical systems. The intention is that after a researcher establishes that a method is of practical utility, a more efficient implementation can be developed.

How do I install `Fanpy`? The `Fanpy` library can be installed using Python package-management systems *pip* and *conda* or directly from its source code. Since `Fanpy` is purely Python and depends mainly on common Python libraries (NumPy and SciPy), it can be installed by simply copying the source code onto the desired directory (though this is not recommended). Detailed instructions on how to install `Fanpy` are available on the `Fanpy` website.

What is the future direction of `Fanpy`? In addition to developing additional methods relevant to our scientific interests, the next iteration of `Fanpy` will mainly focus on improving its performance. The computationally critical components will be outsourced to highly optimized libraries, such as our in-house CI software, PyCI. Some of the performance bottlenecks will be removed by reimplementing some features in Cython or C++. As these improvements may cause problems for some users in terms of ease of use and installation, the pure Python implementation of `Fanpy` will continue to be available. In terms of new features, modules for (arbitrary-order) perturbation theory, equations-of-motion, and quantum-mechanical embedding are currently in various stages of development.

7 Summary

This brief paper introduces `Fanpy` as a library for developing new post-HF *ab initio* methods. `Fanpy`'s goal is to help researchers quickly test their ideas for new correlated electronic structure theory methods and, to achieve this goal, `Fanpy` contains many methods that can be used in countless combinations with one another. These methods are of intrinsic interest but, moreover, they serve as examples to be extended upon. Base classes are available as templates to help users develop a structure that is compatible with the rest of `Fanpy`.

Acknowledgements

We wish to acknowledge various refinements to `Fanpy` library from Matthew Chan, Cristina E. González-Espinoza, Xiaotian D. Yang, Stijn Fias, Caitlin Lanssens, and the QC-Devs Community. P.W.A. and F.H.Z. acknowledge Natural Sciences and Engineering Research Council (NSERC) of Canada, Compute Canada, and CANARIE for financial and computational support. P.W.A. acknowledges support from the Canada Research Chairs. T.V. acknowledges support from the Research Board of Ghent University. R.A.M.Q. acknowledges financial support from the University of Florida in the form of a start-up grant.

8 References

(1) Kim, T. D.; Miranda-Quintana, R. A.; Richer, M.; Ayers, P. W. *Computational and Theoretical Chemistry* **2021**, *1202*, 113187.

(2) Johnson, P.; Ayers, P.; Limacher, P.; De Baerdemacker, S.; Van Neck, D.; Bultinck, P. *Computational and Theoretical Chemistry* **2013**, *1003*, 101–13.

(3) Fecteau, C.-É.; Berthiaume, F.; Khalfoun, M.; Johnson, P. A. *Journal of Mathematical Chemistry* **2020**, 1–13.

(4) Frisch, M. J. et al. Gaussian 16 Revision C.01, Gaussian Inc. Wallingford CT, 2016.

(5) Werner, H.-J.; Knowles, P. J.; Knizia, G.; Manby, F. R.; Schütz, M. *Wiley Interdisciplinary Reviews: Computational Molecular Science* **2012**, *2*, 242–253.

(6) Smith, D. G.; Burns, L. A.; Sirianni, D. A.; Nascimento, D. R.; Kumar, A.; James, A. M.; Schriber, J. B.; Zhang, T.; Zhang, B.; Abbott, A. S., et al. *Journal of chemical theory and computation* **2018**, *14*, 3504–3511.

(7) Smith, D. G.; Burns, L. A.; Simmonett, A. C.; Parrish, R. M.; Schieber, M. C.; Galvelis, R.; Kraus, P.; Kruse, H.; Di Remigio, R.; Alenaizan, A., et al. *The Journal of Chemical Physics* **2020**, *152*, 184108.

(8) Lemmens, L.; Vriendt, X. D.; Tolstykh, D.; Huysentruyt, T.; Bultinck, P.; Acke, G. *The Journal of Chemical Physics* **2021**, *155*, 084802.

(9) Verstraelen, T.; Tecmer, P.; Heidar-Zadeh, F.; Boguslawski, K.; Chan, M.; Zhao, Y.; Kim, T. D.; Vandenbrande, S.; Yang, D.; González-Espinoza, C. E.; Fias, S.; Limacher, P. A.; Berrocal, D.; Malek, A.; Ayers, P. W. HORTON, version 2.0.1, 2015.

(10) Verstraelen, T.; Adams, W.; Pujal, L.; Tehrani, A.; Kelly, B. D.; Macaya, L.; Meng, F.; Richer, M.; Hernández-Esparza, R.; Yang, X. D., et al. *Journal of Computational Chemistry* **2021**, *42*, 458–464.

(11) Boguslawski, K.; Leszczyk, A.; Nowak, A.; Brzęk, F.; Żuchowski, P. S.; Kędziera, D.; Tecmer, P. *Computer Physics Communications* **2021**, *264*, 107933.

(12) Pople, J.; Seeger, R.; Krishnan, R. *International Journal of Quantum Chemistry: Quantum Chemistry Symposium* **1977**, *11*, 149–163.

(13) Bytautas, L.; Henderson, T.; Jiménez-Hoyos, C.; Ellis, J.; Scuseria, G. *Journal of Chemical Physics* **2011**, *135.*

(14) Alcoba, D.; Torre, A.; Luis, L.; Oña, O.; Capuzzi, P.; Van Raemdonck, M.; Bultinck, P.; Van Neck, D. *Journal of Chemical Physics* **2014**, *141.*

(15) Cook, D. *Molecular Physics* **1975**, *30*, 733–743.

(16) Weinhold, F.; Wilson Jr, E. *The Journal of Chemical Physics* **1967**, *46*, 2752–58.

(17) Boys, S. *Proceedings of the Royal Society of London A* **1950**, *200*, 542–554.

(18) Hurley, A.; Lennard-Jones, J.; Pople, J. *A theory of paired-electrons in polyatomic molecules Proceedings of the Royal Society of London Series A* **1953**, *220*, 446–455.

(19) Parr, R.; Ellison, F.; Lykos, P. *Journal of Chemical Physics* **1956**, *24*, 1106.

(20) Parks, J.; Parr, R. *Journal of Chemical Physics* **1958**, *28*, 335–345.

(21) McWeeny, R.; Sutcliffe, B. *Proceedings of the Royal Society of London Series A* **1963**, *273*, 103–116.

(22) Surjan, P. In *Correlation and Localization*, Surjan, P., Ed., 1999, pp 63–88.

(23) Allen, T.; Shull, H. *Journal of Physical Chemistry* **1962**, *66*, 2281–2283.

(24) Tecmer, P.; Boguslawski, K.; Johnson, P.; Limacher, P.; Chan, M.; Verstraelen, T.; Ayers, P. *Journal of Physical Chemistry A* **2014**, *118*, 9058–9068.

(25) Paldus, J.; Cizek, J.; Sengupta, S. *Journal of Chemical Physics* **1971**, *55*, 2452–2462.

(26) Paldus, J.; Sengupta, S.; Cizek, J. *Journal of Chemical Physics* **1972**, *57*, 652–666.

(27) Shull, H. *The Journal of Chemical Physics* **1959**, *30*, 1405–13.

(28) Kutzelnigg, W. *The Journal of Chemical Physics* **1964**, *40*, 3640–47.

(29) Johnson, P.; Limacher, P.; Kim, T.; Richer, M.; Miranda-Quintana, R.; Heidar-Zadeh, F.; Ayers, P.; Bultinck, P.; De Baerdemacker, S.; Van Neck, D. *Computational and Theoretical Chemistry* **2017**, *1116*, 207–219.

(30) Surjan, P.; Szabados, Á.; Jeszenszki, P.; Zoboki, T. *Journal of Mathematical Chemistry* **2012**, *50*, 534–551.

(31) Rassolov, V. *Journal of Chemical Physics* **2002**, *117*, 5978–5987.

(32) Rassolov, V.; Xu, F.; Garashchuk, S. *Journal of Chemical Physics* **2004**, *120*, 10385–10394.

(33) Rassolov, V.; Xu, F. *Journal of Chemical Physics* **2007**, *126*, 234112.

(34) Cassam-Chenaï, P. *Journal of Chemical Physics* **2006**, *124*, 194109.

(35) Cassam-Chenaï, P.; Rassolov, V. *Chemical Physics Letters* **2010**, *487*, 147–152.

(36) Cassam-Chenaï, P.; Ilmane, A. *Journal of Mathematical Chemistry* **2012**, *50*, 652–667.

(37) Stein, T.; Henderson, T.; Scuseria, G. *Journal of Chemical Physics* **2014**, *140*, 214113.

(38) Henderson, T.; Scuseria, G.; Dukelsky, J.; Signoracci, A.; Duguet, T. *Physical Review C* **2014**, *89*, 054305.

(39) Henderson, T.; Bulik, I.; Stein, T.; Scuseria, G. *Journal of Chemical Physics* **2014**, *141*, 244104.

(40) Bulik, I.; Henderson, T.; Scuseria, G. *Journal of Chemical Theory and Computation* **2015**, *11*, 3171–3179.

(41) Cullen, J. *Chemical Physics* **1996**, *202*, 217–229.

(42) Miller, K.; Ruedenberg, K. *Journal of Chemical Physics* **1965**, *43*, S88–S90.

(43) Miller, K.; Ruedenberg, K. *Journal of Chemical Physics* **1968**, *48*, 3414–3443.

(44) Silver, D.; Mehler, E.; Ruedenberg, K. *Journal of Chemical Physics* **1970**, *52*, 1174–1180.

(45) Mehler, E.; Ruedenberg, K.; Silver, D. *Journal of Chemical Physics* **1970**, *52*, 1181–1205.

(46) Silver, D.; Ruedenberg, K.; Mehler, E. *Journal of Chemical Physics* **1970**, *52*, 1206–1227.

(47) Coleman, A. *Journal of Mathematical Physics* **1965**, *6*, 1425–1431.

(48) Coleman, A. *International Journal of Quantum Chemistry* **1997**, *63*, 23–30.

(49) Bajdich, M.; Drobný, G.; Wagner, L.; Schmidt, K. *Physical Review Letters* **2006**, *96*, 130201.

(50) Bajdich, M.; Mitas, L.; Wagner, L.; Schmidt, K. *Physical Review B* **2008**, *77*, 115112.

(51) Pernal, K. *Journal of Chemical Theory and Computation* **2014**, *10*, 4332–4341.

(52) Pastorczak, E.; Pernal, K. *Physical Chemistry Chemical Physics* **2015**, *17*, 8622–8626.

(53) Limacher, P.; Kim, T.; Ayers, P.; Johnson, P.; De Baerdemacker, S.; Van Neck, D. *Molecular Physics* **2014**, *112*, 853–862.

(54) Limacher, P. *Journal of Chemical Physics* **2016**, *145*, 194102.

(55) Boguslawski, K.; Tecmer, P.; Ayers, P.; Bultinck, P.; De Baerdemacker, S.; Van Neck, D. *Physical Review B* **2014**, *89*, 201106.

(56) Boguslawski, K.; Tecmer, P.; Bultinck, P.; De Baerdemacker, S.; Van Neck, D.; Ayers, P. *Journal of Chemical Theory and Computation* **2014**, *10*, 4873–4882.

(57) Silver, D. *The Journal of Chemical Physics* **1969**, *50*, 5108–16.

(58) Limacher, P.; Ayers, P.; Johnson, P.; De Baerdemacker, S.; Van Neck, D.; Bultinck, P. *Journal of Chemical Theory and Computation* **2013**, *9*, 1394–1401.

(59) Schollwöck, U. *Annals of Physics* **2011**, *326*, 96–192.

(60) Paldus, J.; Li, X. In *Advances in Chemical Physics*, Prigogine, I., Rice, S., Eds., 1999; Vol. 110, pp 1–175.

(61) Cizek, J. *Journal of Chemical Physics* **1966**, *45*, 4256–4266.

(62) Shavitt, I.; Bartlett, R., *Many-body methods in chemistry and physics: MBPT and coupled-cluster theory*; Cambridge: Cambridge, 2009.

(63) Bartlett, R.; Musiał, M. *Reviews of Modern Physics* **2007**, *79*, 291–352.

(64) Evangelista, F. A.; Chan, G. K. L.; Scuseria, G. E. *Journal of Chemical Physics* **2019**, *151*, 244112.

(65) Evangelista, F. A. *Journal of Chemical Physics* **2011**, *134*, 224102.

(66) Richardson, R. *Physical Review* **1967**, *159*, 792–805.

(67) Pariser, R.; Parr, R. G. *The Journal of Chemical Physics* **1953**, *21*, 466–471.

(68) Pariser, R.; Parr, R. G. *The Journal of Chemical Physics* **1953**, *21*, 767–776.

(69) Pople, J. A. *Transactions of the Faraday Society* **1953**, *49*, 1375–1385.

(70) Surján, P. R., *Second quantized approach to quantum chemistry: an elementary introduction*; Springer Science & Business Media: 2012.

(71) Hubbard, J. *Proceedings of the Royal Society of London A* **1963**, *276*, 238–257.

(72) Planelles, J.; Zicovich-Wilson, C.; Jaskolski, W.; Corma, A. *International journal of quantum chemistry* **1996**, *60*, 971–981.

(73) Ising, E. *Zeitschrift für Physik* **1925**, *31*, 253–258.

(74) Capelle, K.; Campo Jr, V. L. *Physics Reports* **2013**, *528*, 91–159.

(75) Heisenberg, W. In *Original Scientific Papers Wissenschaftliche Originalarbeiten*; Springer: 1985, pp 580–597.

(76) Richardson, R. **1963**.

(77) Dukelsky, J.; Pittel, S.; Sierra, G. *Reviews of modern physics* **2004**, *76*, 643.

(78) Chuiko, V.; Adams, W.; Richards, A.; Sanchez-Diaz, G.; Richer, M.; Zhao, Y.; Heidar-Zadeh, F.; Ayers, P. W. ModelHamiltonian, version 0.0.0, 2022.

(79) Piela, L., *Ideas of quantum chemistry*; Elsevier: 2013.

(80) Jensen, F., *Introduction to computational chemistry*; John wiley & sons: 2017.

(81) Cramer, C. J., *Essentials of computational chemistry: theories and models*; John Wiley & Sons: 2013.

(82) Helgaker, T.; Jørgensen, P.; Olsen, J., *Modern electronic structure theory*; Wiley: Chichester, 2000.

(83) Szabo, A.; Ostlund, N., *Modern Quantum Chemistry - Introduction to Advanced Electronic Structure Theory*; McGraw-Hill Inc.: 1989, pp 43–107.

(84) Nightingale, M. P.; Umrigar, C. J., *Quantum Monte Carlo methods in physics and chemistry*; 525; Springer Science & Business Media: 1998.

(85) Umrigar, C. *The Journal of chemical physics* **2015**, *143*, 164105.

(86) Sabzevari, I.; Sharma, S. *Journal of chemical theory and computation* **2018**, *14*, 6276–6286.

(87) Neuscamman, E. *The Journal of chemical physics* **2013**, *139*, 194105.

(88) Neuscamman, E. *Journal of chemical theory and computation* **2016**, *12*, 3149–3159.

(89) Kurita, M.; Yamaji, Y.; Morita, S.; Imada, M. *Physical Review B* **2015**, *92*, 035122.

(90) Virtanen, P.; Gommers, R.; Oliphant, T. E.; Haberland, M.; Reddy, T.; Cournapeau, D.; Burovski, E.; Peterson, P.; Weckesser, W.; Bright, J., et al. *Nature methods* **2020**, *17*, 261–272.

(91) Hansen, N.; Auger, A. In *Theory and Principled Methods for the Design of Metaheuristics*, Borenstein, Y., Moraglio, A., Eds.; Springer Berlin Heidelberg: Berlin, Heidelberg, 2014, pp 145–180.

(92) Hansen, N.; Akimoto, Y.; Baudis, P. CMA-ES/pycma on Github, Zenodo, DOI:10.5281/zenodo.2559634, 2019.

(93) Head, T. et al. scikit-optimize/scikit-optimize: v0.5.2, version v0.5.2, 2018.

(94) Sun, Q.; Berkelbach, T. C.; Blunt, N. S.; Booth, G. H.; Guo, S.; Li, Z.; Liu, J.; McClain, J. D.; Sayfutyarova, E. R.; Sharma, S., et al. *Wiley Interdisciplinary Reviews: Computational Molecular Science* **2018**, *8*, e1340.