QUADRATICALLY CONVERGENT SELF-CONSISTENT FIELD (QC-SCF) ORBITAL OPTIMIZATION

[1,2]SHUOYANG WANG, [2]PAUL W. AYERS

Abstract. This project focuses on developing an orbital optimization algorithm using the quadratically convergent self-consistent field (QC-SCF) algorithm in conjunction with the `PyCI`[11] library for quantum chemistry configuration interaction (CI). The QC-SCF algorithm primarily provides a gradient-based optimization method for Hartree-Fock (HF) calculations, while configuration interaction singles and doubles (CISD) are utilized as the CI instance for orbital optimization. This algorithm is flexible and can be extended to accommodate other electronic structure methods, allowing it to adapt to various calculation contexts.

1. Introduction

The molecular electronic problem essentially addresses the internal structure of molecules, which is an important sub-branch in molecular physics and quantum chemistry. For a molecular system with N electrons and M nuclei under the Born-Oppenheimer approximation, the electronic non-relativistic time-independent Schrödinger equation can be written as

$$\left[-\frac{1}{2}\sum_{i=1}^{N}\Delta_i - \sum_{i=1}^{N}\sum_{A=1}^{M}\frac{Z_A}{\|\mathbf{r}_i-\mathbf{R}_A\|} + \sum_{i=1}^{N-1}\sum_{j=i+1}^{N}\frac{1}{\|\mathbf{r}_i-\mathbf{r}_j\|}\right]|\psi\rangle = E|\psi\rangle, \tag{1}$$

where the Hamiltonian is the sum of the electron kinetic energy, electron-nucleus attraction, and electron-electron Coulomb repulsion terms. The N-electron Schrödinger equation is a nonlinear partial differential equation (PDE), and this nonlinearity arises from the Coulomb repulsions between electrons. Since this PDE cannot be solved analytically, the variational method is required to solve the problem. This involves providing an orthonormal trial function guess $|\psi\rangle$. By minimizing the variational energy

$$E_0 \leq \min_{\psi}\langle\psi|\hat{H}|\psi\rangle, \tag{2}$$

an upper bound to the ground state energy E_0 is obtained, and the wave function $|\psi\rangle$ is optimized to be closest to the exact solution.

To propose an appropriate trial function, for the electron system with N electrons $\{\mathbf{x}_i\}_{i=1}^{N}$ and K ($K \geq N$) orthonormalized spin state/orbital basis $\{|\varphi_j\rangle\}_{j=1}^{K}$, the N-electron wave function can be constructed according to the Pauli exclusion principle as

$$|\psi(\mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_N)\rangle = \frac{1}{\sqrt{N!}}\det\begin{bmatrix}|\varphi_1(\mathbf{x}_1)\rangle & |\varphi_2(\mathbf{x}_1)\rangle & \cdots & |\varphi_N(\mathbf{x}_1)\rangle\\ |\varphi_1(\mathbf{x}_2)\rangle & |\varphi_2(\mathbf{x}_2)\rangle & \cdots & |\varphi_N(\mathbf{x}_2)\rangle\\ \vdots & \vdots & \ddots & \vdots\\ |\varphi_1(\mathbf{x}_N)\rangle & |\varphi_2(\mathbf{x}_N)\rangle & \cdots & |\varphi_N(\mathbf{x}_N)\rangle\end{bmatrix}. \tag{3}$$

This antisymmetric non-correlated electron wave function is called a Slater determinant. To appropriately represent the spin orbitals, each spin-orbital $|\psi_j\rangle$ for the i-th occupied electron can be written in the series expansion form of a set of basis functions $\{|\phi_\eta\rangle\}_{\eta=1}^{L}$ as

$$|\psi_j(\mathbf{x}_i)\rangle = \sum_{\eta=1}^{L}C_{\eta j}|\phi_\eta(\mathbf{x}_i)\rangle. \tag{4}$$

Date: April 22, 2024.

[1]Department of Physics and Astronomy, McMaster University, Hamilton, Ontario, L8S 4L8.
[2]Department of Chemistry and Chemical Biology, McMaster University, Hamilton, Ontario, L8S 4L8.

Conventionally, the basis functions $\{|\phi_\eta\rangle\}_{\eta=1}^{L}$ are square-integrable exponential functions in the form of

$$|\phi_\eta(\mathbf{x}_i)\rangle = Ae^{B(\mathbf{x}_i)}, \tag{5}$$

where A is the normalization constant, and the exponent function B commonly takes two forms:

$$B(\mathbf{x}_i) = \begin{cases} -\alpha r^l Y_l^m \|\mathbf{x}_i\| & \text{(Slater type)} \\ -\beta r^l Y_l^m \|\mathbf{x}_i\|^2 & \text{(Gaussian type)} \end{cases}, \qquad \alpha, \beta \in \mathbb{R}. \tag{6}$$

Since Gaussian type orbitals are more computationally efficient for post Hartree-Fock (HF) methods such as configuration interaction (CI), they are primarily used as the basis set (e.g. 6-31G).

For a system with N electrons and K orbitals ($K > N$), the electrons can be arranged in

$$\binom{K}{N} = \frac{K!}{N!(K-N)!} \tag{7}$$

different configurations, including both the ground and excited state configurations, referred to as configuration states. Without loss of generality, consider a non-degenerate system, and the trial wave function is a linear combination of Slater determinants in all possible configuration states, such that

$$|\psi\rangle = c_0|\psi_0\rangle + \sum_{a<r} c_a^r|\psi_a^r\rangle + \sum_{a<r}\sum_{b<s} c_{ab}^{rs}|\psi_{ab}^{rs}\rangle + \dots \tag{8}$$

where $a, b, \dots$ denote the occupied orbitals and $r, s, \dots$ denote the unoccupied orbitals. The first term is the ground state single Slater determinant. If $|\psi_0\rangle$ is solely used as a trial function, it is the HF method. When the second, third, and higher order terms are considered, it is the CI method. For example, if only single and double excitations are considered, it is referred to as configuration interaction singles and doubles (CISD). When all the excited configurations are considered with respect to spin orbitals $\{|\varphi_j\rangle\}_{j=1}^{K}$, this is referred to as the exact solution, called full configuration interaction (FCI) [1, 2, 3].

The quadratically convergent self-consistent field (QC-SCF) algorithm converts the variational problem into a quadratic function optimization problem. This method was first proposed by G. B. Backsay [4, 5, 6], and it was motivated by its theoretical convergence when solving an N-electron system as opposed to the poor convergence of matrix algebra approaches for solving Roothaan equations [7]. A few extensions were recently implemented, such as using the trust-radius optimization scheme for both restricted and unrestricted HF methods [8]. The implemented algorithm in this project is gradient-based, and the optimization primarily makes use of the `scipy.optimize.minimize` function [9].

The orbital optimization process essentially refines the geometry of a molecule to mitigate effects such as reference-dependent CI wave function constructions. This refinement constitutes a further variational step aimed at reducing energy and approaching Full Configuration Interaction (FCI) accuracy [10, 11, 16]. Previous studies have explored and developed methods grounded in orbital space activity [12]. Given that modern methods typically share a similar formal setup, orbital optimization can be seamlessly integrated as a complementary enhancement to approaches like SCI+PT [17] and FANCI [13, 14, 15].

In this project, orbital optimization predominantly leverages the primary optimization of Hartree-Fock (HF) (QC-SCF) to construct the CI wave function using `PyCI` [18], thereby facilitating further energy minimization through the inclusion of excited-state configurations. Since employing HF as a reference for the CI wave function does not invariably yield the optimal orbitals for CI, this construction process can be further refined by iteratively modifying the initial trial function. This iterative refinement resembles optimizing an optimizer and enables further energy reduction in a variational manner.

2. Method

2.1. **Energy, Gradient, and Hessian.** The general N-electron Schrödinger equation (1) can be represented as an N-electron molecular Hamiltonian in second quantization:

$$\hat{H} = \sum_{pq} h_{pq}\hat{a}_p^\dagger \hat{a}_q + \frac{1}{2}\sum_{pqrs} v_{pqrs}\hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_r, \qquad \hat{H}|\psi\rangle = E|\psi\rangle, \tag{9}$$

where $\hat{a}_\sigma^\dagger, \hat{a}_\rho$ are creation and annihilation operators, and the one-electron and two-electron integrals h_{pq} and v_{pqrs} are defined as:

$$h_{pq} = \left\langle \psi_p(\mathbf{r}_i) \left| -\frac{1}{2}\Delta_i - \sum_{A=1}^{M} \frac{Z_A}{\|\mathbf{r}_i - \mathbf{R}_A\|} \right| \psi_q(\mathbf{r}_i) \right\rangle, \quad v_{pqrs} = \left\langle \psi_p(\mathbf{r}_i)\psi_q(\mathbf{r}_j) \left| \frac{1}{\|\mathbf{r}_i - \mathbf{r}_j\|} \right| \psi_r(\mathbf{r}_i)\psi_s(\mathbf{r}_j) \right\rangle. \tag{10}$$

Applying the variational method by imposing the trial function $|\psi\rangle$, and introducing the unitary operator $\hat{U}$, the energy of the system can be written as:

$$E = \left\langle \psi \middle| \hat{H} \middle| \psi \right\rangle = \left\langle \psi \middle| \hat{U}^\dagger \hat{H} \hat{U} \middle| \psi \right\rangle, \tag{11}$$

where the unitary transformation can be exponentially parametrized by the skew-Hermitian operator $\hat{L} - \hat{L}^\dagger$ for an arbitrary coefficient operator $\hat{L}$ with matrix elements l_{ij}:

$$\hat{U} = e^{\hat{L}-\hat{L}^\dagger}, \quad \hat{L} - \hat{L}^\dagger = \sum_{ij} l_{ij}\hat{a}_i^\dagger \hat{a}_j - \sum_{ij} l_{ji}^* \hat{a}_j^\dagger \hat{a}_i. \tag{12}$$

For a sufficiently small transformation $\hat{L}-\hat{L}^\dagger$, using the second-order truncation of the Baker–Campbell–Hausdorff (BCH) formula, the energy can be estimated as:

$$E \approx \left\langle \psi \middle| \hat{H} \middle| \psi \right\rangle + \left\langle \psi \middle| \left[\hat{L} - \hat{L}^\dagger, \hat{H}\right] \middle| \psi \right\rangle + \frac{1}{2}\left\langle \psi \middle| \left[\hat{L} - \hat{L}^\dagger, \left[\hat{L} - \hat{L}^\dagger, \hat{H}\right]\right] \middle| \psi \right\rangle := \mathcal{E} + \mathcal{J} + \mathcal{H}, \tag{13}$$

where the first term $\mathcal{E}$ is the core energy, the second term $\mathcal{J}$ is the gradient component, and the third term $\mathcal{H}$ is the Hessian component. To avoid the abuse of notation, we use the one and two reduced density matrices (1-RDM, 2-RDM) to denote the elements:

$$\gamma_{ab} = \left\langle \psi \middle| \hat{a}_a^\dagger \hat{a}_b \middle| \psi \right\rangle, \quad \Gamma_{abcd} = \left\langle \psi \middle| \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_d \hat{a}_c \middle| \psi \right\rangle. \tag{14}$$

The code for each term expression can be found below.

2.1.1. *Core Energy.* The core energy is approximated at zeroth order:

$$\mathcal{E} = \left\langle \psi \middle| \hat{H} \middle| \psi \right\rangle = \sum_{pq} h_{pq}\gamma_{pq} + \frac{1}{2}\sum_{pqrs} v_{pqrs}\Gamma_{pqrs}. \tag{15}$$

```
### energy function in class OrbEnergy
    """
    (2) compute energy
    """

    def get_energy(self):
        """Compute the energy of the wavefunction.

        Parameters
        ----------
        (N/A)

        Equations
        ----------
        energy = \sum_{pq} h_{pq} \gamma_{pq} + \sum_{pqrs} v_{pqrs} \Gamma_{pqrs}
        Returns
        ----------
        energy : float

        """

        energy = np.einsum("pq,pq", self.h, self.rdm1, optimize=True)
        energy += 0.5 * np.einsum("pqrs,pqrs", self.v, self.rdm2, optimize=True)
        return energy
```

2.1.2. *Gradient Term.* The gradient term serves as the first-order approximation:

$$\mathcal{J} = \left\langle \psi \left| \left[\hat{L} - \hat{L}^{\dagger}, \hat{H} \right] \right| \psi \right\rangle = \sum_{ij} \left(l_{ij} G_{ij} - l_{ji}^{*} G_{ji} \right), \tag{16}$$

where the gradient elements are defined as:

$$G_{ij} = \sum_{p} h_{jq}\gamma_{iq} - \sum_{pq} h_{pi}\gamma_{pj} + \frac{1}{2}\left(\sum_{qrs} v_{jqrs}\Gamma_{iqrs} - \sum_{prs} v_{pjrs}\Gamma_{iprs} - \sum_{pqs} v_{pqis}\Gamma_{pqjs} + \sum_{pqr} v_{pqri}\Gamma_{pqjr} \right). \tag{17}$$

```
### gradient function in class OrbGradient

    """
    (2) compute gradient
    """

    def get_gradient(self,l):
        """

        Parameters
        ----------
        l : ndarray (n,n)
            coefficient matrix (step)

        Equations
        ----------
        g1 = \sum_{pq} \delta_{pj} h_{pq} \gamma_{iq} - \sum_{pq} \delta_{iq} h_{pq} \gamma_{pj}
        g2 = \sum_{pqrs} \delta_{pj} v_{pqrs} \Gamma_{iqrs} - \sum_{pqrs} \delta_{qj} v_{pqrs} \
        ↪ Gamma_{iprs}
           - \sum_{pqrs} \delta_{ir} v_{pqrs} \Gamma_{pqjs} + \sum_{pqrs} \delta_{is} v_{pqrs} \
            ↪ Gamma_{pqjr}.

        Returns
        ----------
        gradient : ndarray (n,n)

        """

        self.l = l
        self.n = self.l.shape[0]
        I = np.eye(self.n, dtype=self.h.dtype)


        # g1
        g1 = np.einsum("pj,pq,iq->ij", I, self.h, self.rdm1, optimize=True)
        g1 -= np.einsum("iq,pq,pj->ij", I, self.h, self.rdm1, optimize=True)
        # g2
        g2 = np.einsum('pj,pqrs,iqrs->ij', I, self.v, self.rdm2, optimize=True)
        g2 -= np.einsum('qj,pqrs,iprs->ij', I, self.v, self.rdm2, optimize=True)
        g2 -= np.einsum('ir,pqrs,pqjs->ij', I, self.v, self.rdm2, optimize=True)
        g2 += np.einsum('is,pqrs,pqjr->ij', I, self.v, self.rdm2, optimize=True)

        Gradient = 2 * (g1 + 0.5*g2)

        return Gradient
```

2.1.3. *Hessian Term.* The Hessian term is given by the second-order approximation:

$$\mathcal{H} = \frac{1}{2}\sum_{ij}\sum_{kl}\left(l_{ij}l_{kl}\text{Hess}_{ijkl} - l_{ij}l^*_{lk}\text{Hess}_{ijlk} - l^*_{ji}l_{kl}\text{Hess}_{jikl} + l^*_{ji}l^*_{lk}\text{Hess}_{jilk}\right), \tag{18}$$

where the Hessian elements are defined as:

$$\begin{aligned}\text{Hess}_{ijkl} = {}& \delta_{kj}\sum_p h_{lp}\gamma_{ip} - h_{li}\gamma_{kj} - h_{jk}\gamma_{il} + \delta_{il}\sum_p h_{pk}\gamma_{pj} \\ & + \frac{1}{2}\Bigg(\delta_{kj}\sum_{qrs} v_{lqrs}\Gamma_{iqrs} - \sum_{rs} v_{ljrs}\Gamma_{ikrs} - \sum_{qs} v_{lqis}\Gamma_{kqjs} + \sum_{qr} v_{lqri}\Gamma_{kqjr} \\ & -\delta_{kj}\sum_{prs} v_{plrs}\Gamma_{iprs} + \sum_{rs} v_{jlrs}\Gamma_{ikrs} + \sum_{ps} v_{plis}\Gamma_{kpjs} - \sum_{pr} v_{plri}\Gamma_{kpjr} \\ & -\sum_{qs} v_{jqks}\Gamma_{iqls} + \sum_{ps} v_{pjks}\Gamma_{ipls} + \delta_{il}\sum_{pqs} v_{pqks}\Gamma_{pqjs} - \sum_{pq} v_{pqki}\Gamma_{pqjl} \\ & +\sum_{qr} v_{jqrk}\Gamma_{iqlr} - \sum_{pr} v_{pjrk}\Gamma_{iplr} - \delta_{il}\sum_{pqr} v_{pqrk}\Gamma_{pqjr} + \sum_{pq} v_{pqik}\Gamma_{pqjl}\Bigg).\end{aligned} \tag{19}$$

Since the implemented QC-SCF algorithm primarily uses the energy and gradient functions, the code for the Hessian term is not included.

2.2. **Initialization.** The QC-SCF method necessitates the input of one and two-electron integrals, and for this primary computation, the Gbasis [19] package is utilized. As atomic orbital (AO) bases are provided by default in the Gbasis [19], the Löwdin transformation [2] is employed to convert them to the molecular orbital (MO) basis. The module for computing integrals is provided below.

```
### Compute 1- and 2- electron integrals from gbasis

from turtle import st
from meanfield import energy
import scipy
import numpy as np
import pyci

from gbasis.integrals.kinetic_energy import kinetic_energy_integral
from gbasis.integrals.nuclear_electron_attraction import nuclear_electron_attraction_integral
from gbasis.integrals.electron_repulsion import electron_repulsion_integral
from meanfield.utils import spinize

def compute_integrals(basis, atcoords, atnums, transform=None):
    """
    Construct 1-electron integrals.

    Parameters
    ----------
    basis : list/tuple of GeneralizedContractionShell
```

```
        Basis set object.
    atcoords : np.ndarray(N, 3)
        Atomic coordinates.
    atnums : np.ndarray(N)
        Atomic numbers.
    transform : np.ndarray(K_orbs, K_cont)
        Transformation matrix from the basis set in the given coordinate system (e.g. AO) to linear
        combinations of contractions (e.g. MO).
        Transformation is applied to the left, i.e. the sum is over the index 1 of 'transform'
        and index 0 of the array for contractions.
        Default is no transformation.

    Returns
    -------
    tuple : np.ndarray(K_orbs, K_orbs), np.ndarray(K_orbs, K_orbs, K_orbs, K_orbs)
        nuclear attraction One and two electron energy integrals of the given basis set.
        If keyword argument 'transform' is provided, then the integrals will be transformed
        accordingly.
        Dimensions 0 and 1 of the arrays are associated with the basis functions in the basis set.
        'K_orbs' is the total number of basis functions in the basis set.
    """

    # compute 1 electron integrals
    k_1e_int = kinetic_energy_integral(basis, transform=transform)
    v_1e_int = nuclear_electron_attraction_integral(basis, atcoords, atnums, transform=transform)

    # compute 2 electron integrals
    v_2e_int = electron_repulsion_integral(basis, transform=transform)

    return k_1e_int + v_1e_int, v_2e_int
```

2.3. **QC-SCF Algorithm.** The energy surface is locally approximated by a quadratic surface, making a gradient-based optimization process sufficient for minimizing the energy and optimizing the orbital. To ensure that the step direction points towards the largest descent direction, a unitary transformation $\mathbf{U} = e^{\mathbf{L}-\mathbf{L}^\dagger}$ is constructed based on the coefficient matrix/step $\mathbf{L}$. This transformation rotates the one and two-electron integrals $\mathbf{h}$ and $\mathbf{v}$ to compute the new gradient, given by

$$h_{PQ} = \sum_{pq} h_{pq} U_{pP} U_{qQ}, \qquad v_{PQRS} = \sum_{pqrs} v_{pqrs} U_{pP} U_{qQ} U_{rR} U_{sS}. \tag{20}$$

The optimization with respect to the coefficient matrix $\mathbf{L}$ is performed using the `scipy.optimize.minimize` function (refer to Figure 2). One example employs the limited-memory Broyden-Fletcher-Goldfarb-Shanno with Box Constraints (L-BFGS-B) method [9] to optimize energy, provided the gradient. The general algorithm is outlined in Figure 3 for reference.

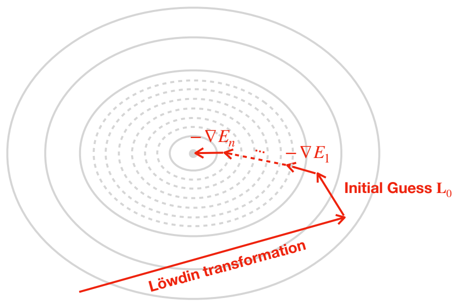

FIGURE 1. a schematic diagram of gradient descent using `scipy.optimize.minimize`

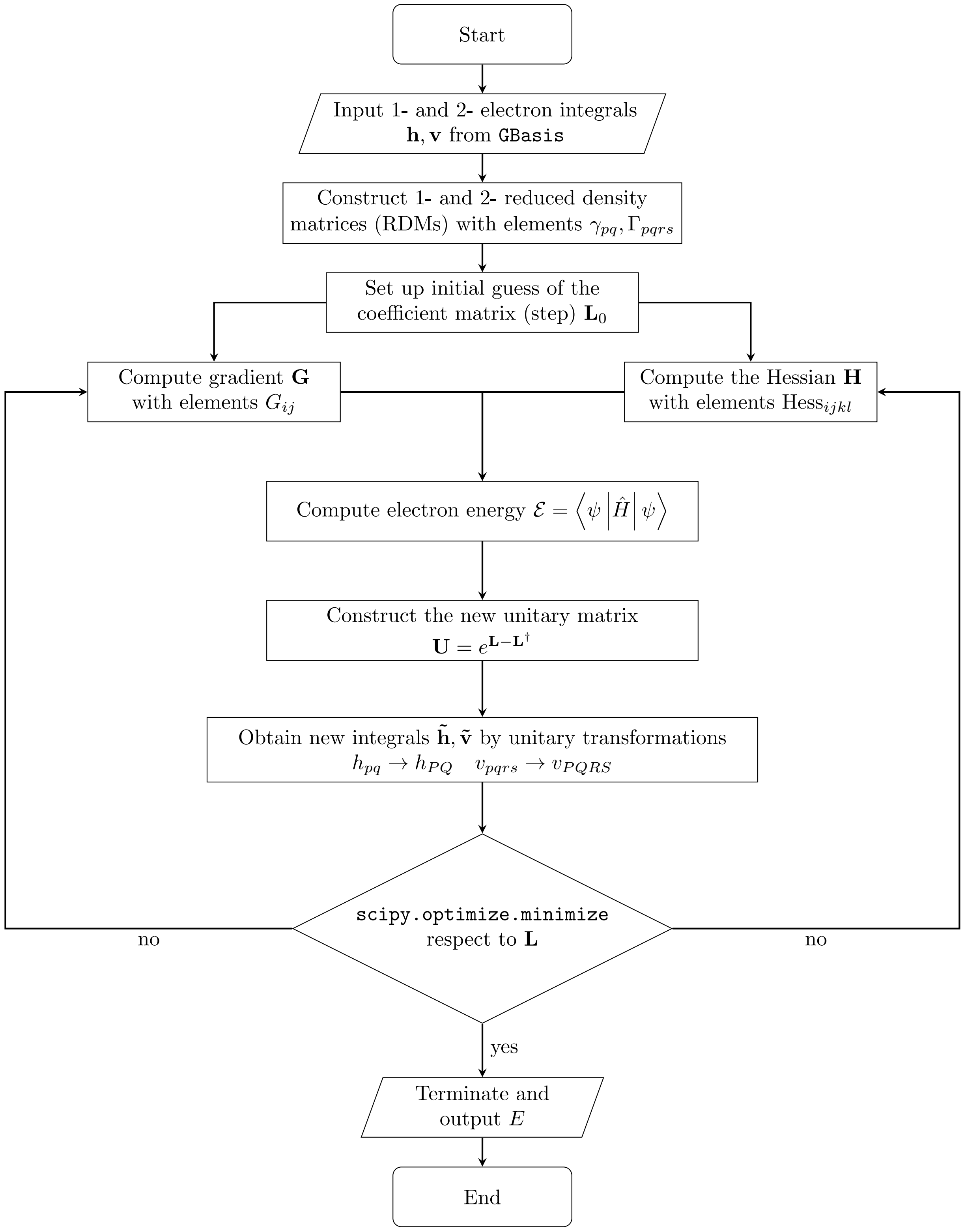
Start
Input 1- and 2- electron integrals $\mathbf{h}, \mathbf{v}$ from GBasis
Construct 1- and 2- reduced density matrices (RDMs) with elements $\gamma_{pq}, \Gamma_{pqrs}$
Set up initial guess of the coefficient matrix (step) $\mathbf{L}_0$
Compute gradient $\mathbf{G}$ with elements G_{ij}
Compute the Hessian $\mathbf{H}$ with elements Hess_{ijkl}
Compute electron energy $\mathcal{E} = \left\langle \psi \middle| \hat{H} \middle| \psi \right\rangle$
Construct the new unitary matrix $\mathbf{U} = e^{\mathbf{L}-\mathbf{L}^\dagger}$
Obtain new integrals $\tilde{\mathbf{h}}, \tilde{\mathbf{v}}$ by unitary transformations $h_{pq} \to h_{PQ} \quad v_{pqrs} \to v_{PQRS}$
scipy.optimize.minimize respect to $\mathbf{L}$
no
no
yes
Terminate and output E
End

FIGURE 2. QC-SCF optimization algorithm

```
### scipy optimizer

# initial guess of coefficient matrix l
l0 = np.zeros(n ** 2) + np.random.rand(n ** 2) * 0.1
l = l0.reshape(n ** 2,1)
l0 = l0 * 1e-2
l = l * 1e-2

# Define Energy, Jacobian functions
def Energy(l, h, v, rdm1, rdm2):
    return Orbscipy_opt(h, v, rdm1, rdm2).Energy(T)
def Jacobian(l,h,v,rdm1,rdm2):
    return Orbscipy_opt(h, v, rdm1, rdm2).Jacobian(T)

# Optimization
Optimization = minimize(Energy, l0, args=(h, v, rdm1, rdm2), method='L-BFGS-B', jac=Jacobian)
```

2.4. **Orbital Optimization.** The implemented algorithm utilizes CISD as a post-HF method, with the `PyCI` [18] package serving as a wrapper around the QC-SCF optimizer for this purpose. The process commences by constructing the CISD wave function using HF-optimized orbitals, which facilitates a decrease in the energy minimum due to the inclusion of excited state configurations. Subsequently, an iterative procedure is employed to adjust the orbital coefficients, thereby minimizing the CISD wave function. Through the variational principle, this optimization enhances the orbitals, yielding a lower energy that closely approximates the exact solution. Schematic diagrams illustrating this process is provided in Figure 4 and Figure 5f. The connection between QC-SCF and `PyCI` is facilitated through a wrapper, with further details available in the appendix for reference.

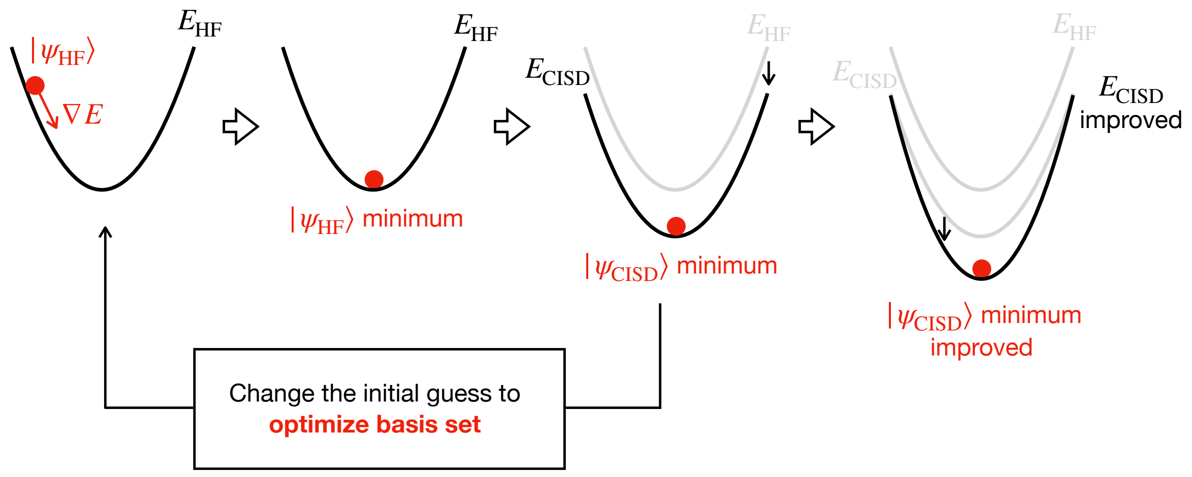
E_{HF}
$|\psi_{HF}\rangle$
∇E
E_{HF}
$|\psi_{HF}\rangle$ minimum
E_{CISD}
E_{HF}
$|\psi_{CISD}\rangle$ minimum
E_{CISD}
E_{HF}
E_{CISD} improved
$|\psi_{CISD}\rangle$ minimum improved
Change the initial guess to optimize basis set

FIGURE 3. Schematic diagram of orbital optimization algorithm (QC-SCF + `PyCI`)

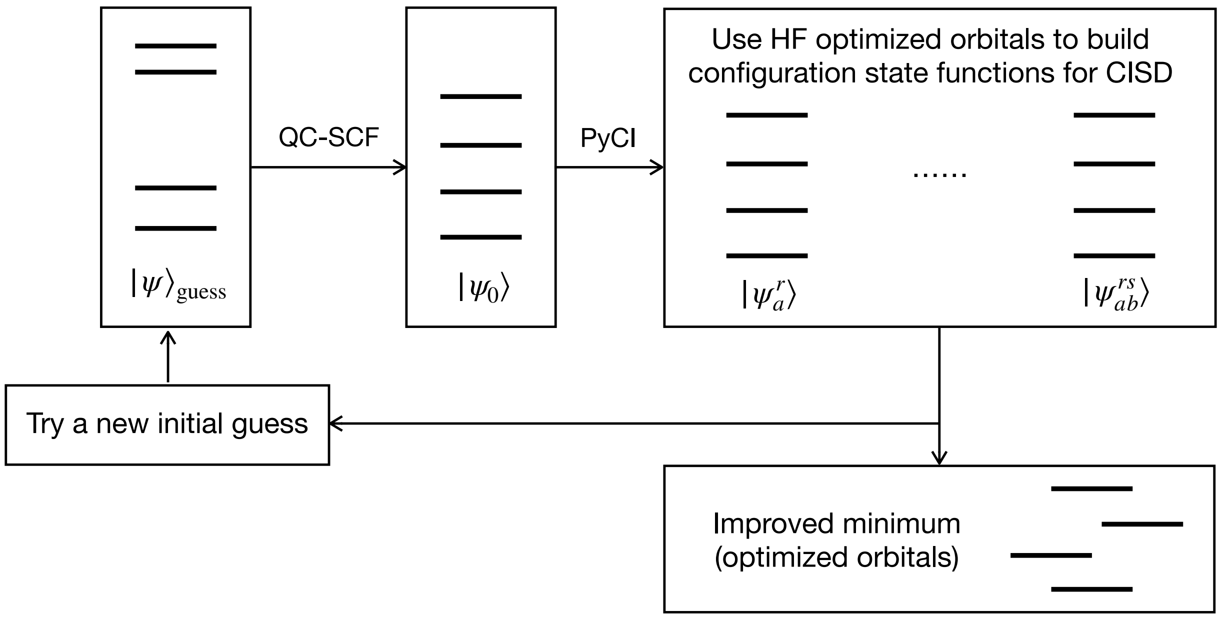
$|\psi\rangle_{\text{guess}}$
QC-SCF
$|\psi_0\rangle$
PyCI
Use HF optimized orbitals to build configuration state functions for CISD
$|\psi_a^r\rangle$
......
$|\psi_{ab}^{rs}\rangle$
Try a new initial guess
Improved minimum (optimized orbitals)

FIGURE 4. Procedure of QC-SCF + PyCI orbital optimization

```
### PyCI orbital optimization

import numpy as np
import os
from meanfield.utils import hartreefock_rdms

# Load starting (HF) MOs and other data
data_file = 'nh3_6-31g_hf.npz'
in_data_path = os.getcwd() + '/result/' + data_file
data = np.load(in_data_path, allow_pickle=True)

occs = data['occs']
guess_mo = data['opt_mos']
atcoords = data['atcoords']
atnums = data['atnums']
ao_basis = list(data['ao_basis'])

nbasis = guess_mo.shape[0]

# get rdms in the MO basis
rdm1, rdm2 = hartreefock_rdms(guess_mo.shape[0], *occs)

# set random starting rotation matrix
l0 = np.zeros(nbasis**2)

import iodata
import numpy as np
import pyci

from meanfield.calculators import compute_gradient_sw
from meanfield.pyci_wrapper import *
```

```


### Compute CISD energy at Hartree-Fock orbitals

# Create a CISD wave function
excitations = (0, 1, 2) # excitations to include (0 = reference, 1 = single, 2 = double)
wfn2 = pyci.fullci_wfn(nbasis, *occs)
pyci.add_excitations(wfn2, *excitations, ref=None)

# compute energy CISD at Hartree-Fock orbitals
cisd_0 = compute_energy(l0, ao_basis, atcoords, atnums, wfn2, guess_mo)
print(f"CISD energy at Hartree-Fock orbitals: {cisd_0}")


### optimize orbitals using PyCI wrapper

import scipy

def compute_eg(l, ao_basis, atcoords, atnums, guess_mo, trace=False):
    e, rdm1, rdm2 = compute_e_rdms(l, ao_basis, atcoords, atnums, wfn2, guess_mo)
    g = compute_gradient_sw(l, ao_basis, atcoords, atnums, rdm1, rdm2, guess_mo)
    if trace:
        print(f'Energy: {e} Gradient norm: {np.linalg.norm(g)}')
    return e, g


args = (ao_basis, atcoords, atnums, guess_mo, True)
result = scipy.optimize.minimize(compute_eg, l0, args = args, method='BFGS', options={'disp': True,
    ↪  'gtol': 1e-8}, jac=True)
```

3. Results and Discussions

3.1. **QC-SCF.** The QC-SCF algorithm, while not the most efficient due to its computational costs being on the order of $\mathcal{O}(n^5)$, is highly adaptable to other methods. Its gradient-based nature makes it naturally suitable for the orbital optimization process, ensuring theoretical convergence, particularly for small systems.

3.2. **Orbital Optimization.** The orbital optimization is applied to several molecules, and their HF, CISD (unoptimized), and CISD (optimized) energies are compared. As depicted in Figures 1 and 4, the optimization begins by optimizing the single Slater determinant for HF before transitioning to the construction of the CISD wave function. These two steps constitute the orbital optimization process. Consequently, the energy follows the relationship:

$$E_{\text{Exact}} < E_{\text{CISD optimized}} < E_{\text{CISD unoptimized}} < E_{\text{HF}} \tag{21}$$

This relationship indicates that the orbitals are optimized to approach the exact solution based on the variational principle. The optimization process is applied to several molecules, and the validity of the energy relationship is confirmed in Table 1. Additionally, the percentage energy difference is visualized in Figure 5. For small systems such as H_2 and LiH, the energy difference between HF and unoptimized CISD is much larger than the difference between unoptimized and optimized CISD. This discrepancy arises primarily because the orbitals are already well-suited, resulting in minimal changes during orbital optimization.

Table 1. Energy values (in Hartree) for different molecules

Molecule	**HF Energy**	**CISD Energy**	**CISD Optimized Energy**
H_2	-1.840458	-1.865408	-1.865408
Ne	-128.473877	-128.586270	-128.586723
LiH	-8.971516	-8.990539	-8.990543
H_2O	-85.653328	-85.779408	-85.779569
NH_3	-76.882577	-76.963038	-76.963319
CH_4	-63.371173	-63.440603	-63.440846

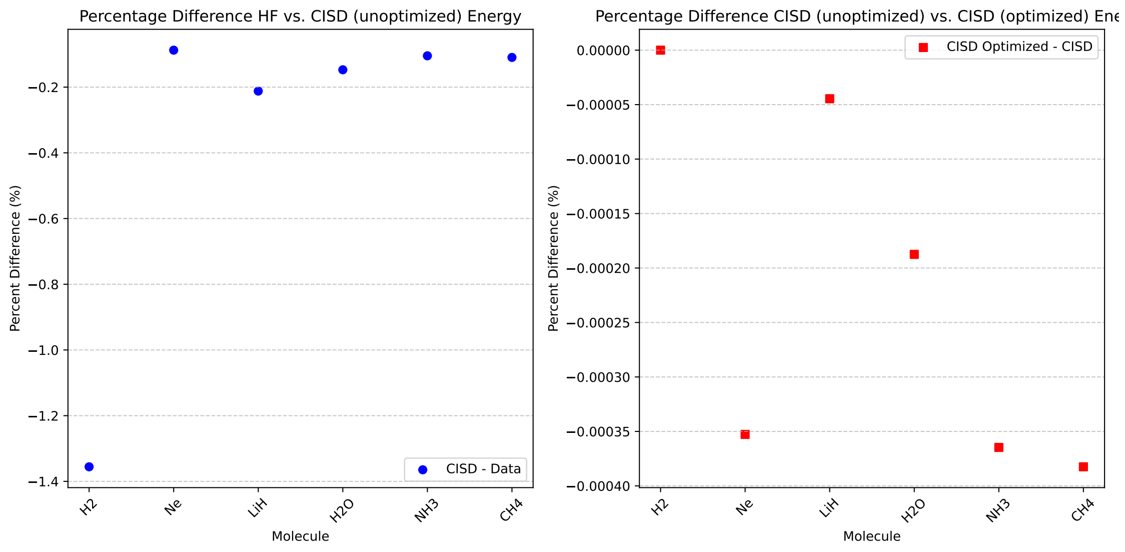

Figure 5. Energy Difference Comparison: HF, CISD (unoptimized), CISD (optimized)

4. Conclusion & Future Direction

In this project, an orbital optimization algorithm is implemented by coupling a QC-SCF algorithm with `PyCI` [18]. The QC-SCF employs second-order BCH truncation to approximate the quadratic energy surface and generate a gradient-based optimization scheme used for primary HF calculations. `PyCI` [18] is utilized to construct CISD wave functions and iteratively cooperate with QC-SCF to find optimized orbitals for the system. The algorithm is planned to be generalized and adapted to other electronic structure methods, and efforts are needed for a complete implementation of the Hessian component.

5. Acknowledgement

I would like to express my gratitude to my supervisor, Dr. Paul W. Ayers, for his invaluable support and guidance throughout the project over the last year. Additionally, I extend my thanks to the postdoc, Dr. Marco Martinez Gonzalez, for his assistance in `PyCI` [18] wrapper and implementing orbital optimization.

6. Appendix

The appendix contains a more detailed derivation of the equations included in the method section, along with the important pieces of code provided for comprehensive reference.

6.1. **Derivation.** A detailed derivation of gradient and Hessian terms are provided as a reference.

6.1.1. *Gradient.* The gradient term (16) from can be expanded and simplified as

$$\begin{aligned}\mathcal{J} = \left\langle \psi \middle| \left[\hat{L} - \hat{L}^\dagger, \hat{H}\right] \middle| \psi \right\rangle &= \sum_{ij} l_{ij} \left\langle \psi \middle| \left[\hat{a}_i^\dagger \hat{a}_j, \hat{H}\right] \middle| \psi \right\rangle - \sum_{ij} l_{ji}^* \left\langle \psi \middle| \left[\hat{a}_j^\dagger \hat{a}_i, \hat{H}\right] \middle| \psi \right\rangle \\ &= \sum_{ij} \left(l_{ij} G_{ij} - l_{ji}^* G_{ji} \right)\end{aligned} \tag{22}$$

where l_{ij}, l_{ij}^* are transformation coefficients, and the gradient elements are

$$G_{ij} = \sum_{pq} h_{pq} \left\langle \psi \middle| \left[\hat{a}_i^\dagger \hat{a}_j, \hat{a}_p^\dagger \hat{a}_q\right] \middle| \psi \right\rangle + \sum_{pqrs} v_{pqrs} \left\langle \psi \middle| \left[\hat{a}_i^\dagger \hat{a}_j, \hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_r\right] \middle| \psi \right\rangle . \tag{23}$$

Evaluate the commutators

$$\left[\hat{a}_i^\dagger \hat{a}_j, \hat{a}_p^\dagger \hat{a}_q\right] = \delta_{pj} (\hat{a}_i^\dagger \hat{a}_q) - \delta_{iq} (\hat{a}_p^\dagger \hat{a}_j), \tag{24}$$

and

$$\left[\hat{a}_i^\dagger \hat{a}_j, \hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_r\right] = \delta_{pj} (\hat{a}_i^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_r) - \delta_{qj} (\hat{a}_i^\dagger \hat{a}_p^\dagger \hat{a}_s \hat{a}_r) - \delta_{ir} (\hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_j) + \delta_{is} (\hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_r \hat{a}_j). \tag{25}$$

Then, the equation (11) can be simplified and represented using 1- and 2- RDMs (14) as following.

$$\begin{aligned}G_{ij} &= \sum_{pq} \delta_{pj} h_{pq} \gamma_{iq} - \sum_{pq} \delta_{iq} h_{pq} \gamma_{pj} + \sum_{pqrs} \delta_{pj} v_{pqrs} \Gamma_{iqrs} - \sum_{pqrs} \delta_{qj} v_{pqrs} \Gamma_{iprs} \\ &\quad - \sum_{pqrs} \delta_{ir} v_{pqrs} \Gamma_{pqjs} + \sum_{pqrs} \delta_{is} v_{pqrs} \Gamma_{pqjr} \\ &= \sum_{p} h_{jq} \gamma_{iq} - \sum_{pq} h_{pi} \gamma_{pj} + \sum_{qrs} v_{jqrs} \Gamma_{iqrs} - \sum_{prs} v_{pjrs} \Gamma_{iprs} - \sum_{pqs} v_{pqis} \Gamma_{pqjs} + \sum_{pqr} v_{pqri} \Gamma_{pqjr}\end{aligned} \tag{26}$$

For the real coefficients $l_{ij} = l_{ji}^*$, it has $G_{ij} = -G_{ji}$ and it simplifies to

$$\mathcal{J} = 2 \sum_{ij} l_{ij} G_{ij} \tag{27}$$

6.1.2. *Hessian.* The Hessian term (18) can be expanded and simplified as

$$
\begin{aligned}
\mathcal{H} &= \frac{1}{2}\left\langle \psi \middle| \left[\hat{L}-\hat{L}^\dagger, \left[\hat{L}-\hat{L}^\dagger, \hat{H}\right]\right] \middle| \psi \right\rangle \\
&= \frac{1}{2}\left(\sum_{ij}\sum_{kl} l_{ij}l_{kl} \left\langle \psi \middle| \left[\hat{a}_i^\dagger \hat{a}_j, \left[\hat{a}_k^\dagger \hat{a}_l, \hat{H}\right]\right] \middle| \psi \right\rangle - \sum_{ij}\sum_{kl} l_{ij}l_{lk}^* \left\langle \psi \middle| \left[\hat{a}_i^\dagger \hat{a}_j, \left[\hat{a}_l^\dagger \hat{a}_k, \hat{H}\right]\right] \middle| \psi \right\rangle \right. \\
&\quad \left. - \sum_{ij}\sum_{kl} l_{ji}^* l_{kl} \left\langle \psi \middle| \left[\hat{a}_j^\dagger \hat{a}_i, \left[\hat{a}_k^\dagger \hat{a}_l, \hat{H}\right]\right] \middle| \psi \right\rangle + \sum_{ij}\sum_{kl} l_{ji}^* l_{lk}^* \left\langle \psi \middle| \left[\hat{a}_j^\dagger \hat{a}_i, \left[\hat{a}_l^\dagger \hat{a}_k, \hat{H}\right]\right] \middle| \psi \right\rangle \right). \\
&= \frac{1}{2}\sum_{ij}\sum_{kl}\left(l_{ij}l_{kl}\mathrm{Hess}_{ijkl} - l_{ij}l_{lk}^*\mathrm{Hess}_{ijlk} - l_{ji}^* l_{kl}\mathrm{Hess}_{jikl} + l_{ji}^* l_{lk}^*\mathrm{Hess}_{jilk}\right),
\end{aligned} \tag{28}
$$

where the Hessian elements are

$$
\mathrm{Hess}_{ijkl} = \sum_{pq} h_{pq} \left\langle \Psi \middle| \left[\hat{a}_i^\dagger \hat{a}_j, \left[\hat{a}_k^\dagger \hat{a}_l, \hat{a}_p^\dagger \hat{a}_q\right]\right] \middle| \Psi \right\rangle + \sum_{pqrs} v_{pqrs} \left\langle \Psi \middle| \left[\hat{a}_i^\dagger \hat{a}_j, \left[\hat{a}_k^\dagger \hat{a}_l, \hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_r\right]\right] \middle| \Psi \right\rangle. \tag{29}
$$

Evaluate the commutators

$$
\left[\hat{a}_i^\dagger \hat{a}_j, \left[\hat{a}_k^\dagger \hat{a}_l, \hat{a}_p^\dagger \hat{a}_q\right]\right] = \delta_{pl}\delta_{kj}(\hat{a}_i^\dagger \hat{a}_q) - \delta_{pl}\delta_{iq}(\hat{a}_k^\dagger \hat{a}_j) - \delta_{kq}\delta_{pj}(\hat{a}_i^\dagger \hat{a}_l) + \delta_{kq}\delta_{il}(\hat{a}_p^\dagger \hat{a}_j), \tag{30}
$$

and

$$
\begin{aligned}
\left[\hat{a}_i^\dagger \hat{a}_j, \left[\hat{a}_k^\dagger \hat{a}_l, \hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_r\right]\right] &= \delta_{pl}\delta_{kj}(\hat{a}_i^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_r) - \delta_{pl}\delta_{qj}(\hat{a}_i^\dagger \hat{a}_k^\dagger \hat{a}_s \hat{a}_r) - \delta_{pl}\delta_{ir}(\hat{a}_k^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_j) + \delta_{pl}\delta_{is}(\hat{a}_k^\dagger \hat{a}_q^\dagger \hat{a}_r \hat{a}_j) \\
&\quad - \delta_{ql}\delta_{kj}(\hat{a}_i^\dagger \hat{a}_p^\dagger \hat{a}_s \hat{a}_r) + \delta_{ql}\delta_{pj}(\hat{a}_i^\dagger \hat{a}_k^\dagger \hat{a}_s \hat{a}_r) + \delta_{ql}\delta_{ir}(\hat{a}_k^\dagger \hat{a}_p^\dagger \hat{a}_s \hat{a}_j) - \delta_{ql}\delta_{is}(\hat{a}_k^\dagger \hat{a}_p^\dagger \hat{a}_r \hat{a}_j) \\
&\quad - \delta_{kr}\delta_{pj}(\hat{a}_i^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_l) + \delta_{kr}\delta_{qj}(\hat{a}_i^\dagger \hat{a}_p^\dagger \hat{a}_s \hat{a}_l) + \delta_{kr}\delta_{il}(\hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_j) - \delta_{kr}\delta_{is}(\hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_l \hat{a}_j) \\
&\quad + \delta_{ks}\delta_{pj}(\hat{a}_i^\dagger \hat{a}_q^\dagger \hat{a}_r \hat{a}_l) - \delta_{ks}\delta_{qj}(\hat{a}_i^\dagger \hat{a}_p^\dagger \hat{a}_r \hat{a}_l) - \delta_{ks}\delta_{il}(\hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_r \hat{a}_j) + \delta_{ks}\delta_{ir}(\hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_l \hat{a}_j).
\end{aligned} \tag{31}
$$

Thus, consider the Hessian as the sum of one electron term $(\mathrm{Hess}_{ijkl})_1$ and two electron terms $(\mathrm{Hess}_{ijkl})_2$ such that

$$
\begin{aligned}
(\mathrm{Hess}_{ijkl})_1 &= \sum_{pq}\delta_{pl}\delta_{kj}h_{pq}\gamma_{iq} - \sum_{pq}\delta_{pl}\delta_{iq}h_{pq}\gamma_{kj} - \sum_{pq}\delta_{kq}\delta_{pj}h_{pq}\gamma_{il} + \sum_{pq}\delta_{kq}\delta_{il}h_{pq}\gamma_{pj} \\
&= \delta_{kj}\sum_{p} h_{lp}\gamma_{ip} - h_{li}\gamma_{kj} - h_{jk}\gamma_{il} + \delta_{il}\sum_{p} h_{pk}\gamma_{pj}
\end{aligned} \tag{32}
$$

and
(33)

$$\begin{aligned}
(\mathrm{Hess}_{ijkl})_2 = &\sum_{pqrs} \delta_{pl}\delta_{kj} v_{pqrs}\Gamma_{iqrs} - \sum_{pqrs} \delta_{pl}\delta_{qj} v_{pqrs}\Gamma_{ikrs} - \sum_{pqrs} \delta_{pl}\delta_{ir} v_{pqrs}\Gamma_{kqjs} + \sum_{pqrs} \delta_{pl}\delta_{is} v_{pqrs}\Gamma_{kqjr} \\
&- \sum_{pqrs} \delta_{ql}\delta_{kj} v_{pqrs}\Gamma_{iprs} + \sum_{pqrs} \delta_{ql}\delta_{pj} v_{pqrs}\Gamma_{ikrs} + \sum_{pqrs} \delta_{ql}\delta_{ir} v_{pqrs}\Gamma_{kpjs} - \sum_{pqrs} \delta_{ql}\delta_{is} v_{pqrs}\Gamma_{kpjr} \\
&- \sum_{pqrs} \delta_{kr}\delta_{pj} v_{pqrs}\Gamma_{iqls} + \sum_{pqrs} \delta_{kr}\delta_{qj} v_{pqrs}\Gamma_{ipls} + \sum_{pqrs} \delta_{kr}\delta_{il} v_{pqrs}\Gamma_{pqjs} - \sum_{pqrs} \delta_{kr}\delta_{is} v_{pqrs}\Gamma_{pqjl} \\
&+ \sum_{pqrs} \delta_{ks}\delta_{pj} v_{pqrs}\Gamma_{iqlr} - \sum_{pqrs} \delta_{ks}\delta_{qj} v_{pqrs}\Gamma_{iplr} - \sum_{pqrs} \delta_{ks}\delta_{il} v_{pqrs}\Gamma_{pqjr} + \sum_{pqrs} \delta_{ks}\delta_{ir} v_{pqrs}\Gamma_{pqjl} \\
= &\,\delta_{kj}\sum_{qrs} v_{lqrs}\Gamma_{iqrs} - \sum_{rs} v_{ljrs}\Gamma_{ikrs} - \sum_{qs} v_{lqis}\Gamma_{kqjs} + \sum_{qr} v_{lqri}\Gamma_{kqjr} \\
&- \delta_{kj}\sum_{prs} v_{plrs}\Gamma_{iprs} + \sum_{rs} v_{jlrs}\Gamma_{ikrs} + \sum_{ps} v_{plis}\Gamma_{kpjs} - \sum_{pr} v_{plri}\Gamma_{kpjr} \\
&- \sum_{qs} v_{jqks}\Gamma_{iqls} + \sum_{ps} v_{pjks}\Gamma_{ipls} + \delta_{il}\sum_{pqs} v_{pqks}\Gamma_{pqjs} - \sum_{pq} v_{pqki}\Gamma_{pqjl} \\
&+ \sum_{qr} v_{jqrk}\Gamma_{iqlr} - \sum_{pr} v_{pjrk}\Gamma_{iplr} - \delta_{il}\sum_{pqr} v_{pqrk}\Gamma_{pqjr} + \sum_{pq} v_{pqik}\Gamma_{pqjl}.
\end{aligned}$$

6.2. **Initial Set-Up.** The initialization of the classes `OrbEnergy, OrbGradient` in gradient and energy module shares the same set-up function `__init__` as following.

```
### Initialization of classes OrbEnergy, OrbGradient

def __init__(self, h, v, rdm1, rdm2, rdm2_construction = 'HF', functional=None, restrict=None):
    """

    Parameters
    ----------
    h : ndarray (n,n)
        One-electron integrals, spinized
    v : ndarray (n,n,n,n)
        Two-electron integrals, spinized but not antisymmetrized
    rdm1 : ndarray (n,n)
        One-electron reduced density matrix (1-rdm)
    rdm2 : ndarray (n,n,n,n)
        Two-electron reduced density matrix (2-rdm)
    functional : str
        if DFT is used, functional specification
    restrict : dict
        Indicates whether the calculation is restricted, unrestricted, or generalized. (default
            ↪ generalized)
        Indicates whether the coefficients ar real or complex. (default complex)
        Lists orbitals that should be frozen. (default none)
    """

    self.h = h
    self.v = v
    self.rdm1 = rdm1
    self.rdm2 = rdm2
    self.restrict = restrict
    self.functional = functional
    self.rdm2_construction = rdm2_construction
    # Check to be sure all matrices have appropriate dimensions and symmetry.
    n = self.h.shape[0]
    if self.h.shape != (n,n): # check dimension of h
        raise ValueError("h should be a n*n matrix")
    if self.v.shape != (n,n,n,n): # check dimension of v
        raise ValueError("v should be a n*n block matrix with each component matrix being n*n")
    if self.rdm1.shape != (n,n): # check dimension of rdm1
        raise ValueError("rdm1 should be a n*n matrix")
    if not np.allclose(self.h,self.h.conj().T): # check symmetry/Hermitian of h
        raise ValueError("h should be Hermitian")
    if not np.allclose(self.rdm1,self.rdm1.conj().T): # check symmetry/Hermitian of rdm1
        raise ValueError("rdm1 should be Hermitian")

    # If 2DM is not passed, construct it from the 1DM in a sensible way.
    def function_dict_dm2(x): # a function addapt to different constructions of rdm2
        def HF(x):
            return x
        def Hartree(x):
            return np.zeros_like(x)
        def Muller(x):
            return np.sqrt(x)
        def Power(x,n):
            return np.power(x,n)

        function_dict_dm2 = {
            'HF': HF,
            'Hartree': Hartree,
            'Muller': Muller,
            'Power': Power,
        }
        return(function_dict_dm2[x])

    if self.rdm2 is None:
        n = self.rdm1.shape[0]
        self.rdm2 = np.einsum('pr, qs -> pqrs', self.rdm1, self.rdm1)
        self.rdm2 -= function_dict_dm2(self.rdm2_construction)(np.einsum('ps, qr -> pqrs', self.rdm1
            ↪ , self.rdm1))

    if not np.allclose(self.rdm2,self.rdm2.conj().transpose(1,0,3,2)): # check symmetry/Hermitian
        ↪ of rdm2
        raise ValueError("v should be Hermitian")

    if functional is None:
        functional = "None"
```

6.3. **Reference Checks.** The optimized HF energies through QC-SCF algorithm are checked against the HF method in PySCF [20] before passing to the construction of CI wave functions. One example of LiH is shown below.

```
### Reference Check using PySCF

from pyscf import gto, scf

# Define the molecule
mol = gto.Mole()
mol.build(
    verbose=0,
    atom='H 0 0 0; Li 0 0 3.0236',
    basis='sto-6g',
    spin=0,
    charge=0,
    unit = 'Bohr'
)

# Set up and run Hartree-Fock calculation
mf = scf.RHF(mol)
mf.conv_tol = 1e-6 # convergence tolerance
energy = mf.kernel() # run SCF and get total energy

# Output results
print('Total energy (including nuclear repulsion):', energy)
print('Pure electronic energy:', energy - mf.energy_nuc())
```

6.4. `PyCI` **wrapper.** The `PyCI` wrapper bridges the QC-SCF and `PyCI` [18], facilitating orbital optimization. Several fundamental functions are adapted for universal application.

```
 ### Module for PyCI wrapper

 from turtle import st
from meanfield import energy
import scipy
import numpy as np
import pyci

from gbasis.integrals.kinetic_energy import kinetic_energy_integral
from gbasis.integrals.nuclear_electron_attraction import nuclear_electron_attraction_integral
from gbasis.integrals.electron_repulsion import electron_repulsion_integral
from meanfield.utils import spinize

__all__ = ["compute_e_rdms", "compute_energy", "compute_gradient", "update_mo", "compute_integrals"
    ↪ ]

def compute_integrals(basis, atcoords, atnums, transform=None):
    """
    Construct 1-electron integrals.

    Parameters
    ----------
    basis : list/tuple of GeneralizedContractionShell
        Basis set object.
    atcoords : np.ndarray(N, 3)
        Atomic coordinates.
    atnums : np.ndarray(N)
        Atomic numbers.
    transform : np.ndarray(K_orbs, K_cont)
        Transformation matrix from the basis set in the given coordinate system (e.g. AO) to linear
        combinations of contractions (e.g. MO).
        Transformation is applied to the left, i.e. the sum is over the index 1 of ‘transform‘
        and index 0 of the array for contractions.
        Default is no transformation.

    Returns
    -------
    tuple : np.ndarray(K_orbs, K_orbs), np.ndarray(K_orbs, K_orbs, K_orbs, K_orbs)
        nuclear attraction One and two electron energy integrals of the given basis set.
        If keyword argument ‘transform‘ is provided, then the integrals will be transformed
        accordingly.
        Dimensions 0 and 1 of the arrays are associated with the basis functions in the basis set.
        `K_orbs` is the total number of basis functions in the basis set.
    """

    # compute 1 electron integrals
    k_1e_int = kinetic_energy_integral(basis, transform=transform)
    v_1e_int = nuclear_electron_attraction_integral(basis, atcoords, atnums, transform=transform)

    # compute 2 electron integrals
    v_2e_int = electron_repulsion_integral(basis, transform=transform)

    return k_1e_int + v_1e_int, v_2e_int


def update_mo(step, init_mo=None):
    """Update MOs

    Update the MOs given the initial MOs and the step vector.

    Parameters
    ----------
    step : np.ndarray(N_MO * N_AO)
        Step vector
    init_mo : np.ndarray(N_MO, N_AO)
        Initial MOs

    Returns
    -------
    mo : np.ndarray(N_MO, N_AO)
        Updated MOs
    """
    # rehape l to square matrix TODO: check length of basis
    step_matrix = step.reshape(int(len(step) ** 0.5), int(len(step) ** 0.5))

    # compute transformation matrix
    step_matrix_transform = scipy.linalg.expm(step_matrix - step_matrix.conj().T)
    # compute new AO to MO transformation matrix
    if init_mo is None:
        transform_matrix = step_matrix_transform
    else:
        transform_matrix = step_matrix_transform @ init_mo
    return transform_matrix


def compute_e_rdms(l, basis, atcoords, atnums, wfn, guess_mo_coeffs=None):
    """Compute energy for PyCI wavefunction

    Compute the energy, 1rdm and 2rdm for a PyCI wavefunction given the step vector l,
    the basis set, the atomic coordinates, the atomic numbers, and the initial transformation
    matrix (e.g. From AOs to MOs).

    Parameters
    ----------
    l : np.ndarray
        Step vector for the optimization
    basis : list/tuple of GeneralizedContractionShell
        Contracted Cartesian Gaussians (of the same shell) that will be used to construct an array.
    atcoords : np.ndarray(N_nuc, 3)
        Coordinates of each atom.
    atnums : np.ndarray
        Atomic numbers
    wfn : wavefunction
        PyCI wavefunction object
    guess_mo_coeffs : np.ndarray(N_MO, N_AO) or None
        Initial matrix coefficients for the MOs

    Returns
    -------
    energy : float
        Energy of the wavefunction
    rdm1 : np.ndarray
        One-electron reduced density matrix
    rdm2 : np.ndarray
        Two-electron reduced density matrix
    """

    # update MO coefficients with step vector
    new_mo_coeffs = update_mo(l, guess_mo_coeffs)

    # compute integrals for the new MOs
    h, v = compute_integrals(basis, atcoords, atnums, transform=new_mo_coeffs)

    # create pyci hamiltonian
    ham = pyci.secondquant_op(0.0, h, v)

    # declare sparse operator using the hamiltonian and the wavefunction
    op = pyci.sparse_op(ham, wfn)

    # solve sparse operator
    e_vals, e_vecs = op.solve(n=1, tol=1.0e-9)

    # compute rdms
    d1, d2 = pyci.compute_rdms(wfn, e_vecs[0])

    rdm1, rdm2 = pyci.spinize_rdms(d1, d2) # TODO: check if this is needed

    return e_vals, rdm1, rdm2


def compute_energy(l, basis, atcoords, atnums, wfn, guess_mo_coeffs=None, trace=False):
    """Compute energy for PyCI wavefunction

    Compute the energy for a PyCI wavefunction given the step vector l, the basis set, the atomic
    coordinates, the atomic numbers, and the initial transformation matrix (e.g. From AOs to MOs).

    Parameters
    ----------
    l : np.ndarray
        Step vector for the optimization
    basis : list/tuple of GeneralizedContractionShell
        Contracted Cartesian Gaussians (of the same shell) that will be used to construct an array.
    atcoords : np.ndarray(N_nuc, 3)
        Coordinates of each atom.
    atnums : np.ndarray
        Atomic numbers
    wfn : wavefunction
        PyCI wavefunction object
    guess_mo_coeffs : np.ndarray(N_MO, N_AO) or None
        Initial matrix coefficients for the MOs
            trace : bool
        If True, print the energy at each step of the optimization. Default is False.

    Returns
    -------
    energy : float
        Energy of the wavefunction
    """
    energy, _, _ = compute_e_rdms(l, basis, atcoords, atnums, wfn, guess_mo_coeffs)
    if trace:
        print(f"Energy: {energy}")
    return energy


def compute_gradient(l, basis, atcoords, atnums, wfn, guess_mo_coeffs=None):
    """Compute gradient for PyCI wavefunction

    Compute the energy for a PyCI wavefunction given the step vector l, the basis set, the atomic
    coordinates, the atomic numbers, and the initial transformation matrix (e.g. From AOs to MOs).

    Parameters
    ----------
    l : np.ndarray
        Step vector for the optimization
    basis : list/tuple of GeneralizedContractionShell
        Contracted Cartesian Gaussians (of the same shell) that will be used to construct an array.
    atcoords : np.ndarray(N_nuc, 3)
        Coordinates of each atom.
    atnums : np.ndarray
        Atomic numbers
    wfn : wavefunction
        PyCI wavefunction object
    guess_mo_coeffs : np.ndarray(N_MO, N_AO) or None
        Initial matrix coefficients for the MOs

    Returns
    -------
    gradient : np.ndarray(N)
        Gradient of the wavefunction
    """
    # update MO coefficients with step vector
    new_mo_coeffs = update_mo(l, guess_mo_coeffs)

    # compute rdms
    _, rdm1, rdm2 = compute_e_rdms(l, basis, atcoords, atnums, wfn, guess_mo_coeffs)

    # compute integrals
    h, v = compute_integrals(basis, atcoords, atnums, transform=new_mo_coeffs)

    h = spinize(h)
    v = spinize(v)

    nbasis = h.shape[0]
    # construct identity matrix
    I = np.eye(nbasis)
    # g1
    g1 = np.einsum("pj,pq,iq->ij", I, h, rdm1, optimize=True)
    g1 -= np.einsum("iq,pq,pj->ij", I, h, rdm1, optimize=True)

    # g2
    g2 = np.einsum("pj,pqrs,iqrs->ij", I, v, rdm2, optimize=True)
    g2 -= np.einsum("qj,pqrs,iprs->ij", I, v, rdm2, optimize=True)
    g2 -= np.einsum("ir,pqrs,pqjs->ij", I, v, rdm2, optimize=True)
    g2 += np.einsum("is,pqrs,pqjr->ij", I, v, rdm2, optimize=True)

    Gradient = 2 * (g1 + 0.5 * g2)

    # TODO: this is a temporary crazy test
    # take only the diagonal blocks of the gradient and average them
    gradient = (Gradient[: nbasis // 2, : nbasis // 2] + Gradient[nbasis // 2 :, nbasis // 2 :]) /
        ↪ 2
    gradient = gradient.ravel()
    # print(f"Gradient norm: {np.linalg.norm(gradient)}")

    return gradient
```

References

[1] Helgaker, T.J. (2014) Molecular electronic-structure theory. London, UK: Wiley.

[2] Attila, S. and Ostlund, N.S. (1989) Modern Quantum Chemistry. New York, USA: Dover.

[3] Lehtola, S., Blockhuys, F. and Van Alsenoy, C. (2020) 'An overview of self-consistent field calculations within finite basis sets', Molecules, 25(5), p. 1218. doi:10.3390/molecules25051218.

[4] Pulay, P. (1982) 'Improved scf convergence acceleration', Journal of Computational Chemistry, 3(4), pp. 556–560. doi:10.1002/jcc.540030413.

[5] Bacskay, G.B. (1981) 'A quadratically convergent Hartree—Fock (QC-SCF) method. application to closed Shell Systems', Chemical Physics, 61(3), pp. 385–404. doi:10.1016/0301-0104(81)85156-7.

[6] Bacskay, G.B. (1982) 'A quadritically convergent Hartree-Fock (QC-SCF) method. application to open shell orbital optimization and coupled perturbed Hartree-Fock calculations', Chemical Physics, 65(3), pp. 383–396. doi:10.1016/0301-0104(82)85211-7.

[7] Roothaan, C.C. (1951) 'New developments in molecular orbital theory', Reviews of Modern Physics, 23(2), pp. 69–89. doi:10.1103/revmodphys.23.69.

[8] Nottoli, T., Gauss, J. and Lipparini, F. (2021) 'A black-box, general purpose quadratic self-consistent field code with and without cholesky decomposition of the two-electron integrals', Molecular Physics, 119(21–22). doi:10.1080/00268976.2021.1974590.

[9] Zhu, C. et al. (1997) 'Algorithm 778: L-BFGS-B', ACM Transactions on Mathematical Software, 23(4), pp. 550–560. doi:10.1145/279232.279236.

[10] Roos, B.O., Taylor, P.R. and Sigbahn, P.E.M. (1980) 'A complete active space SCF method (CASSCF) using a density matrix formulated super-CI approach', Chemical Physics, 48(2), pp. 157–173. doi:10.1016/0301-0104(80)80045-0.

[11] Roos, B.O. (2009) 'The complete active space SCF method in a fock-matrix-based super-CI formulation', International Journal of Quantum Chemistry, 18(S14), pp. 175–189. doi:10.1002/qua.560180822.

[12] Limacher, P. A.; Kim, T. D.; Ayers, P. W.; Johnson, P. A.; De Baerdemacker, S.; Van Neck, D.; Bultinck, P. The Influence of Orbital Rotation on the Energy of Closed-Shell Wavefunctions. Molecular Physics 2014, 112 (5), 853–862.

[13] Kim, T. D.; Miranda-Quintana, R. A.; Richer, M.; Ayers, P. W. Flexible Ansatz for N-Body Configuration Interaction. Computational and Theoretical Chemistry 2021, 1202, 113187.

[14] Johnson, P. A.; Limacher, P. A.; Kim, T. D.; Richer, M.; Alain Miranda-Quintana, R.; Heidar-Zadeh, F.; Ayers, P. W.; Bultinck, P.; De Baerdemacker, S.; Van Neck, D. Strategies for Extending Geminal-Based Wavefunctions: Open Shells and Beyond. Computational and Theoretical Chemistry 2017, 1116, 207–219. https://doi.org/10.1016/j.comptc.2017.05.010.

[15] Kim, T. D.; Richer, M.; Sánchez-Díaz, G.; Miranda-Quintana, R. A.; Verstraelen, T.; Heidar-Zadeh, F.; Ayers, P. W. Fanpy: A Python Library for Prototyping Multideterminant Methods in Ab Initio Quantum Chemistry. Journal of Computational Chemistry 2023, 44 (5), 697–709. https://doi.org/10.1002/jcc.27034.

[16] Yaffe, L.G. and Goddard, W.A. (1976) 'Orbital optimization in electronic wave functions; equations for quadratic and cubic convergence of general multiconfiguration wave functions', Physical Review A, 13(5), pp. 1682–1691. doi:10.1103/physreva.13.1682.

[17] Yao, Y. and Umrigar, C.J. (2021) 'Orbital optimization in selected Configuration Interaction Methods', Journal of Chemical Theory and Computation, 17(7), pp. 4183–4194. doi:10.1021/acs.jctc.1c00385.

[18] QC-Devs Community (2020) Theochem/pyci: A flexible AB-initio quantum chemistry library for (parameterized) configuration interaction calculations., GitHub. Available at: https://github.com/theochem/PyCI (Accessed: 22 April 2024).

[19] QC-Devs Community (2019) Theochem/gbasis: Python Library for Analytical Evaluation and integration of gaussian-type basis functions and related quantities., GitHub. Available at: https://github.com/theochem/gbasis?tab=coc-ov-file (Accessed: 22 April 2024).

[20] Sun, Q. et al. (2017) 'PySCF: The python-based simulations of Chemistry Framework', WIREs Computational Molecular Science, 8(1). doi:10.1002/wcms.1340.

Email address: `wangs455@mcmaster.ca, ayers@mcmaster.ca`