GQCP: The Ghent Quantum Chemistry Package

I. INTRODUCTION

Software in the computational molecular sciences community is currently undergoing transformations in order to manage its ever-increasing complexity and to prepare itself for the advent of exascale hardware architectures.\textsuperscript{1,2} In an attempt to solve both problems at once, modularity has emerged as a software paradigm. Underlying this paradigm shift is the engineering principle of the \textit{separation of concerns},\textsuperscript{3} in which the full software stack is composed of a set of modular components that each perform a specialized task. This modular design allows for developments to be made in each module independently, which greatly decreases maintenance efforts,\textsuperscript{4} and targeting new kinds of hardware can be offloaded to specialized adapter modules instead of requiring large-scale rewrites at the level of the solution of the quantum chemical problem itself.\textsuperscript{5}

Many software libraries are currently targeting modularity,\textsuperscript{4,6–13} but with it comes the requirement of interoperability between different modules. In an attempt to overcome this obstacle, there has been a recent focus on providing programmatic communication through APIs (application programming interfaces) that specify instructions and transmit data,\textsuperscript{6,13–21} and some libraries have adopted support for custom-written plug-ins.\textsuperscript{14,18,20,21} Another recent trend is allowing access to lower-level intermediates, which are typically the matrix representations of one- and two-electron integrals.\textsuperscript{11,14,18,20–23}

Even though many general-purpose as well as specific niche software packages exist, there is still room for software that specializes in the rapid prototyping of new proof of principles, especially if it incorporates features that are not yet omnipresent in the current electronic structure software ecosystem. Among others, such features are as follows: the possibility to modify the algorithmic structure of quantum chemical solvers without requiring the re-compilation of the whole project, the ability to work with real and complex orbitals, and the flexibility to use restricted, unrestricted or generalized\textsuperscript{24,25} orbitals.

In the open-source Ghent Quantum Chemistry Package (GQCP), we have implemented computational objects that bear semantic correspondence with the concepts that appear in the electronic structure theory, while keeping the aforementioned features in mind. Rather than offering highly optimized and specialized routines, our goal is to deliver intuitive and reusable software components (both in C++ and Python) in a domain-specific language, in order to enable method developers to focus on the high-level formulation of prototypes for new computational procedures. By implementing core “building blocks,” our design embodies the hypothesis that new research can emerge from the combination of carefully thought-out high-level software objects. Furthermore, we offer the flexibility of allowing run-time structural modifications of quantum chemical solvers and we provide access to all lower-level intermediates. It is this focus on expressiveness that renders our package complementary to other related initiatives such as Psi4,\textsuperscript{20} PySCF,\textsuperscript{21} and ChronusQ.\textsuperscript{26}

In order to elucidate the two-component concepts that lie at the core of GQCP, we first provide a didactical introduction to
the theoretical framework behind spinor bases (Sec. II). We then
give an overview of how we have implemented the underlying con-
cepts, how we have targeted flexible algorithmic design, and provide
some illustrative applications that highlight the capabilities of GQCP
(Sec. III). We also detail how we adhere to the current best practices
in quantum chemical software engineering (Sec. IV).

II. THEORETICAL CONCEPTS

In electronic structure theory, solving the Schrödinger equation
\[ \hat{H} |\Psi\rangle = E |\Psi\rangle, \]
amounts to finding the eigenvectors $|\Psi\rangle$ of the Hamiltonian of
the system, $\hat{H}$. Unfortunately, only for the smallest systems the
Hamiltonian can be diagonalized, and thus, approximate methods
are required. In such methods, the wave function is parameterized
by a set of model parameters, which define the approximate
ground state $|\Psi(p)\rangle$ (and possibly approximate excited states) after
an optimization procedure.27

Lykos and Pratt28 report how Löwdin considers that approximate
methods pose a dilemma with respect to the symmetry properties
imposed by the constants of motion $\hat{\Lambda}$ (with $[\hat{H}, \hat{\Lambda}] = 0$) of the
system. Indeed, as the approximations $|\Psi(p)\rangle$ are not necessarily
eigenfunctions of the Hamiltonian, imposing those symmetries on
approximate wave function models implies placing constraints that
will restrict their variational freedom and hence increase the energy.
On the other hand, breaking such symmetry requirements leads to
lower energy solutions that are mixtures of different symmetry types,
which among others may lead to worse performance for properties
other than the energy.29–33

As the total projected spin $\hat{S}_z$ is a symmetry of the spinless
field-free molecular Hamiltonian, wave function models are usually
chosen to be eigenvectors of $\hat{S}_z$. In order to accomplish this, the one-
electron functions that are used in the wave function model are then
chosen to be eigenfunctions of individual electron projected spin.
These one-electron functions are often called spin-orbitals,27

\[
\begin{align*}
\phi_{\alpha}(r) &= \phi_{\alpha}^\uparrow(r) \alpha(\omega) \\
\phi_{\beta}(r) &= \phi_{\beta}^\downarrow(r) \beta(\omega),
\end{align*}
\]
and solely consist of a space part multiplied by a spin part. Here, the
space–spin coordinate $x = (r, \omega)$ collects the space and spin coor-
dinates $r$ and $\omega$, respectively. Inspired by Löwdin’s argument,29 the
most general admissible form for the one-electron functions would
however be one that represents an expansion in the complete spin
basis. Such one-electron functions are spinors,24,25,27

\[
\phi(r) = \phi_{\alpha}^\uparrow(r) \alpha(\omega) + \phi_{\beta}^\downarrow(r) \beta(\omega),
\]
which simultaneously hold spin-$\alpha$ and spin-$\beta$ character. Such
spinors arise naturally when the Hamiltonian is no longer spin-
free, for example, when the spin–orbit coupling interaction is included,23–36 or when the presence of an external magnetic field explicitly
requires the treatment of electron spin;35,37–39 although, for uniform magnetic fields applied along the $z$-axis, spin-orbitals are sufficient. Spinors are furthermore used in so-called
broken-symmetry theories and their application has led to lower
energies,40–45 but the lower energy does come at the cost of the loss of
true quantum numbers. As this may turn out to lead to poor chem-
ical properties other than the energy, this conundrum has sparked
recent approaches toward symmetry restoration.46–51

In the following sections, we explore the theory of second quanti-
zation in terms of spinors and contrast it with the simplifications
that arise when using a spin-orbital basis. The distinctions that appear in these formalisms are discussed at the level of the orbitals,
one- and two-electron operators and density matrices, in order to
prepare the discussion of the semantic design of GQCP’s reusable
software objects in Sec. III.

A. Spinors are two-component orbitals

In the spin basis, we can alternatively represent the spinor (4)
as the two-component function,

\[
\vec{\phi}_\alpha(r) = \begin{pmatrix} \phi_{\alpha}^\uparrow(r) \\ \phi_{\alpha}^\downarrow(r) \end{pmatrix}
\]
The scalar product of two spinors can be calculated using the
multiplication of the row vector $\vec{\phi}_\alpha$ and the column vector $\vec{\phi}_\beta$,

\[
\int dr \phi_{\alpha}^\dagger(r) \phi_{\beta}(r) = \int dr \phi_{\alpha}^\dagger(r) \phi_{\beta}^\downarrow(r) + \int dr \phi_{\alpha}^\dagger(r) \phi_{\beta}^\uparrow(r),
\]
equivalent to the scalar product

\[
\int dx \phi_{\alpha}^\dagger(x) \phi_{\beta}(x) = \int dx \phi_{\alpha}^\dagger(x) \phi_{\beta}^\downarrow(x) + \int dx \phi_{\alpha}^\dagger(x) \phi_{\beta}^\uparrow(x),
\]
where we have used the orthonormality of the spin functions $\alpha$ and $\beta$. In an orthonormal basis of $M$ spinors $\{\phi_p(r)|p = 1, \ldots, M\}$, the
corresponding elementary creation and annihilation operators, $\hat{a}_p^\dagger$ and $\hat{a}_p$, obey the usual fermionic anticommutation relations52

\[
[\hat{a}_p, \hat{a}_q^\dagger] = \delta_{pq}
\]
and

\[
[\hat{a}_p, \hat{a}_q] = 0.
\]
The associated fermionic field operators

\[
\hat{\phi}(r) = \sum_p \phi_p(r) \hat{a}_p^\dagger
\]
and

\[
\hat{\phi}^\dagger(r) = \sum_p \phi_p(r) \hat{a}_p
\]
can be used to express one- and two-electron operators in a
spinor basis, which is a procedure that is commonly referred to as
the quantization53,54,55 of the one- and two-electron operators.
The coordinate representation (indicated by the superscript \( c \)) of a one-electron operator \( f^c \) for a system of \( N \) electrons is given by

\[
f^c(r_1, \ldots, r_N) = \sum_{i=1}^{N} f^c(r_i),
\]

where each individual one-electron operator \( f^c(r_i) \) is mathematically represented by a \((2 \times 2)\) matrix operator,

\[
f^c(r_i) = \begin{pmatrix} f^{c\alpha\alpha}(r_i) & f^{c\alpha\beta}(r_i) \\ f^{c\beta\alpha}(r_i) & f^{c\beta\beta}(r_i) \end{pmatrix}.
\]

Each underlying one-electron operator \( f^{c\alpha\alpha}(r_i) \) is a scalar operator, which means that it only acts on one of the components of a spinor and not on the two-component spinors themselves. Using the field operators (10) and (11), the second-quantized analogue of (12) becomes

\[
\hat{f} = \int \text{d}r \hat{\phi}^\dagger(r) f^c(r) \hat{\phi}(r) = \sum_{PQ} f_{PQ} \hat{d}_P^\dagger \hat{d}_Q,
\]

where the parameters \( f_{PQ} \) are the one-electron integrals,

\[
f_{PQ} = \int \text{d}r_1 \text{d}r_2 \hat{\phi}_1^\dagger(r_1) f^c(r_1) \hat{\phi}_2(r_2) \hat{\phi}_1(r_1).
\]

Analogously, a two-electron operator

\[
g^c(r_1, \ldots, r_N) = \frac{1}{2} \sum_{i<j}^N g^c(r_i, r_j)
\]

can be represented as

\[
g = \frac{1}{2} \int \text{d}r_1 \text{d}r_2 \hat{\phi}_1^\dagger(r_1) f^c(r_1, r_2) \hat{\phi}_2(r_2) \hat{\phi}_1(r_1) = \frac{1}{2} \sum_{PQRS} g_{PQRS} \hat{d}_P^\dagger \hat{d}_Q^\dagger \hat{\phi}_P \hat{\phi}_Q,
\]

where the two-electron parameters for the Coulomb repulsion operator are given by

\[
g_{PQRS} = \sum_{\sigma\tau} (P \sigma Q \tau | R \tau S \sigma),
\]

with \((P \sigma Q \tau | R \tau S \sigma)\) a Coulomb integral over the spinor components in Mulliken notation,

\[
(P \sigma Q \tau | R \tau S \sigma) = \int \text{d}r_1 \text{d}r_2 \phi^{\sigma \tau}_1(r_1) \phi^{\tau \sigma}_2(r_2) \times \frac{1}{|r_1 - r_2|} \phi^{\alpha \beta}_3(r_2) \phi^{\beta \alpha}(r_2).
\]

Here, we have used \( \sigma \) and \( \tau \) as generic labels for \( \alpha \) and \( \beta \).

Given a (normalized) wave function \( |\Psi\rangle \), we define the one-electron density matrix (1-DM) as

\[
D_{PQ} = \langle \Psi | \hat{d}_P^\dagger \hat{d}_Q | \Psi \rangle
\]

and the two-electron density matrix (2-DM) as

\[
d_{PQRS} = \langle \Psi | \hat{d}_P^\dagger \hat{d}_Q^\dagger \hat{\phi}_P \hat{\phi}_Q | \Psi \rangle.
\]

With these definitions, the expectation value of a one-electron operator can be written as

\[
\langle \Psi | f | \Psi \rangle = \sum_{PQ} f_{PQ} D_{PQ}
\]

and the expectation value of a two-electron operator as

\[
\langle \Psi | \hat{g} | \Psi \rangle = \frac{1}{2} \sum_{PQRS} g_{PQRS} d_{PQRS}.
\]

### B. Imposing spin symmetries leads to spin-orbital bases

When both components of a spinor are non-zero, it is not an eigenvector of the Pauli matrix \( \sigma_i \). However, if \( K_\alpha \) spinors only have a non-zero \( \alpha \)-component and \( K_\beta \) spinors (with \( M = K_\alpha + K_\beta \)) only have a non-zero \( \beta \)-component, we can rearrange them as

\[
\begin{pmatrix}
\phi_1^{\alpha}(r) \\
\phi_2^{\alpha}(r) \\
0 \\
\vdots \\
0 \\
\phi_{K_\alpha+1}^{\beta}(r) \\
\phi_{K_\beta}(r)
\end{pmatrix}
\]

and relabel them according to the label of their non-zero component,

\[
\phi_p(r) \rightarrow \phi_{po}(r) =
\begin{pmatrix}
\phi_p^{\alpha}(r) \\
0 \\
0 \\
\vdots \\
0 \\
\phi_p^{\beta}(r)
\end{pmatrix}
\quad 1 \leq p \leq K_\alpha
\]

\[
\phi_p(r) \rightarrow \phi_{po}(r) =
\begin{pmatrix}
\phi_p^{\alpha}(r) \\
0 \\
0 \\
\vdots \\
0 \\
\phi_p^{\beta}(r)
\end{pmatrix}
\quad K_\alpha < p \leq M
\quad 1 \leq p \leq K_\beta.
\]

We note in passing that Eqs. (27) and (2) are alternative representations of a spin-orbital, in the same way that Eqs. (5) and (4) are alternative representations of a spinor.

The resulting compound lowercase indices \( po \) are used as labels for spin-orbitals. Since these spin-orbitals have either of their components set to zero, they are eigenvectors of the Pauli matrix \( \sigma_i \).

In a spin-orbital basis, the field operators (10) and (11) can be rewritten as

\[
\hat{\phi}^\dagger(r) = \sum_p \phi_{po}^\dagger(r) \hat{d}_{po}
\]

and

\[
\hat{\phi}(r) = \sum_p \phi_{po}(r) \hat{d}_{po},
\]

with the associated elementary anticommutation relations

\[
[\hat{d}_{po}, \hat{d}_{pq}^\dagger] = \delta_{pq}\delta_{op}.
\]
and
\[ [d_{pq}, d_{qr}]_{\alpha} = 0. \] (31)

The expressions for the second-quantized representations of the one- and two-electron operators (12) and (17) then reduce to
\[ \hat{f} = \sum_{pq} f_{pq,qr} d_{pq}^{\dagger} d_{qr} \] (32)
and
\[ \hat{g} = \frac{1}{2} \sum_{pqrs} (p_{pq} q_{rs}) d_{pq}^{\dagger} d_{rs} d_{qr}. \] (33)

Similarly, 1- and 2-DMs reduce to
\[ D_{pq}^{\alpha} = \langle \Psi | d_{pq}^{\dagger} d_{qr} | \Psi \rangle \] (34)
and
\[ d_{pqrs}^{pert} = \langle \Psi | d_{pq}^{\dagger} d_{rs} d_{qr} | \Psi \rangle, \] (35)

which can be used in the following expectation values:
\[ \langle \Psi | \hat{f} | \Psi \rangle = \sum_{pq} f_{pq,qr} D_{pq}^{\alpha} \] (36)
and
\[ \langle \Psi | \hat{g} | \Psi \rangle = \frac{1}{2} \sum_{pqrs} (p_{pq} q_{rs}) d_{pqrs}^{pert}. \] (37)

In the special case that the one-electron operator (13) is zero in its off-diagonal elements and equal in its diagonal elements, i.e., \( f_{pq,qr}^{\text{off}} = f_{pq,qr}^{\text{diag}} \), and the components of the \( \alpha \)-spin-orbitals are equal to those of the \( \beta \)-spin-orbitals, its second-quantized representation can be written as
\[ \hat{f} = \sum_{pq} f_{pq} \hat{E}_{pq}, \] (38)
where we have introduced the one-electron singlet excitation operator\(^{27,47}\)
\[ \hat{E}_{pq} = \sum_{\sigma} d_{pq}^{\sigma} d_{qr}. \] (39)

Similarly, in this case, the two-electron Coulomb operator can be represented by
\[ \hat{g} = \frac{1}{2} \sum_{pqrs} g_{pqrs} \hat{E}_{pqrs}, \] (40)
where we have introduced the two-electron singlet excitation operator\(^{27}\)
\[ \hat{E}_{pqrs} = \sum_{\sigma} d_{pq}^{\sigma} d_{rs} d_{qr}. \] (41)

In this (restricted) basis, we define the orbital 1- and 2-DMs\(^{27}\)
\[ D_{pq} = \sum_{\sigma} D_{pq}^{\sigma} \] (42)
and
\[ d_{pqrs} = \sum_{\sigma} d_{pqrs}^{pert}, \] (43)

which are used in the following expectation values:
\[ \langle \Psi | \hat{f} | \Psi \rangle = \sum_{pq} f_{pq} D_{pq} \] (44)
and
\[ \langle \Psi | \hat{g} | \Psi \rangle = \frac{1}{2} \sum_{pqrs} (p_{pq} q_{rs}) d_{pqrs}. \] (45)

C. Expanding spinors in scalar basis functions leads to coefficient matrices

By introducing a basis of \( K \) scalar basis functions \( \{ \chi_{\mu}(r) \mid \mu = 1, \ldots, K \} \), we can express each of the components of the spinors as a linear combination of these basis functions,
\[ \phi_{\mu}^{\alpha}(r) = \sum_{\mu} \chi_{\mu}(r) C_{\mu \mu}^{\alpha}. \] (46)

Since we use the same scalar basis for both components, the expansion coefficients \( C_{\mu \mu}^{\alpha \beta} \) can be arranged into the square \( (M \times M) \)-coefficient matrix \( C \) (with \( M = 2K \)),
\[ C = \begin{pmatrix} C_{11} & \cdots & C_{1K} & C_{1(K+1)} & \cdots & C_{1M} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ C_{K1} & \cdots & C_{KK} & C_{K(K+1)} & \cdots & C_{KM} \end{pmatrix}, \] (47)
in which every column \( P \) collects the expansion coefficients of the spinor \( \phi_{\mu}^{\alpha} \). In a spin-orbital basis, the coefficient matrix thus has a blocked diagonal form,
\[ C = \begin{pmatrix} C^{\alpha} & 0 \\ 0 & C^{\beta} \end{pmatrix}, \] (48)
and with the further restriction that the components of the \( \alpha \)-spin-orbitals are equal to those of the \( \beta \)-spin-orbitals, the spinor coefficient matrix can be written as
\[ C = \begin{pmatrix} C' & 0 \\ 0 & C' \end{pmatrix}, \] (49)
with \( C' = C^{\alpha} = C^{\beta} \).

Using the expansion (46), we can calculate the one-electron integrals (16) as
\[ f_{pq} = \sum_{\mu \nu} C_{\mu \nu}^{\alpha} f_{\mu \nu}^{\alpha} C_{\nu \mu}^{\alpha}, \] (50)
where we have introduced the one-electron integrals in the underlying scalar basis as
\[ f_{\mu \nu}^{\alpha} = \int dr \chi_{\nu}^{\ast}(r) f_{\mu \nu}^{\alpha}(r) \chi_{\nu}(r). \] (51)
We should note that Eq. (50) is equivalent to the matrix multiplication

\[
f = C^\dagger \left( f^{\alpha\beta} f^{\rho\sigma} \right) C,
\]

where \( f^{\alpha\beta} \) is the matrix representation of the scalar operator \( f^{\alpha\beta} \) in terms of the underlying scalar basis. The two-electron Coulomb integrals (20) can be calculated using the expansion (46) as

\[
g_{\text{QCP}} = \sum_{\sigma\tau} \sum_{\mu\rho} C_\sigma^\mu C_{\tau\rho}^* C_\mu^\sigma C_\rho^\tau (\mu\nu|\rho\lambda), \tag{53}
\]

in which \((\mu|\nu|\rho|\lambda)\) is a Coulomb integral over scalar basis functions,

\[
(\mu|\nu|\rho|\lambda) = \int d\mathbf{r}_1 d\mathbf{r}_2 \chi_\mu^*(\mathbf{r}_1) \chi_\nu(\mathbf{r}_1) \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} \chi_\nu^*(\mathbf{r}_2) \chi_\lambda(\mathbf{r}_2). \tag{54}
\]

III. DESIGN AND APPLICATIONS

Before highlighting some of GQCP's illustrative capabilities later on in this section, we first detail GQCP's overarching design choices. We refer the reader to our documentation\(^{19}\) for a more comprehensive and continuously updated overview of the APIs.

GQCP offers interfaces both in C++ (in its core library) and Python (through the use of PyBind11).\(^{18}\) In general, we refer to real-valued (double) or complex-valued (complex<double>) types with the subscripts \_d and \_cd in our Python bindings. For example, the real-valued C++ type ScalarRSQOneElectronOperator<double> (cf. Sec. III A) is bound to the Python class ScalarRSQOneElectronOperator_d. In order to make this manuscript as readable as possible, we have opted not to pollute the text with these template parameters or subscripts, but the code examples do reflect the actual use of the APIs. Note that the code listings use GQCP's Python interface and implicitly assume that the gqcpy module has been imported:

```python
import gqcpy
```

Listing 1: Importing the gqcpy Python module.

A. Orbital bases, operators, density matrices, and transformations

Based on the theoretical foundations covered in Sec. II, we have distinguished and implemented three distinct orbital bases. If the spinor coefficient matrix (47) is fully dense, we use the terminology generalized spinor basis (with the corresponding class GSpinorBasis). If the spinor coefficient matrix contains off-diagonal zero blocks, we distinguish two kinds of spin-orbital bases. When the diagonal blocks are equal, the spin-orbital basis is considered to be restricted and the corresponding type is RSpinOrbitalBasis; and when they are not (necessarily), the spin-orbital basis is said to be unrestricted and is represented by an instance of USpinOrbitalBasis. Note that we have borrowed this nomenclature from three spin symmetry-related formulations of Hartree–Fock theory: RHF, UHF, and GHF.\(^{49,50}\) We note that the restricted, unrestricted, and generalized distinctions are propagated throughout our APIs through the prefix R/U/G.

An orbital basis object serves as the starting point for the second-quantized representation of one- and two-electron operators. The result of such a quantization procedure is an instance of XYSQOneElectronOperator or XYSQTTwoElectronOperator, where SQ is used as an abbreviation of SecondQuantized. The prefix X=Scalar, Vector, Tensor is used in order to accommodate the tensorial structure of the operators. For example, second-quantized scalar operators such as the kinetic energy operator or the nuclear attraction operator can be represented by ScalarYSQOneElectronOperator. Vector operators, such as the dipole operator, the electronic spin operator, or the linear momentum operator can be represented by VectorYSQOneElectronOperator. Higher-order tensor operators, such as the quadrupole operator, can finally be represented by TensorYSQOneElectronOperator. Most of the methods that are usually only available for scalar operators are thus made available for higher-order tensor operators, such as basis transformations, the calculation of expectation values, and the evaluation in a Fock (sub)space.

In GQCP, the Hamiltonian is regarded as a distinct operator. For most quantum chemical applications, it suffices to represent it as a sum of one- and two-electron operators. To this end, GQCP provides the three types of YSQHamiltonian depending on which orbital basis the Hamiltonian is expressed in, allowing the user to choose the most applicable type in their routines. We also support the specialized representation of model Hamiltonians such as the Hubbard Hamiltonian.

Generally, quantum chemical routines optimize a set of model parameters (such as the amplitudes in coupled cluster theories or the expansion coefficients in configuration interaction theories), given a specification of the orbital basis and the one- and two-electron operators that define the Hamiltonian of the system. From these model parameters, the one- and two-electron density matrices (1- and 2-DMs) are used to calculate expectation values of second-quantized operators. GQCP provides G1DM and G2DM as types to represent the 1- and 2-DMs [cf. Eqs. (22) and (23)] expressed in a generalized spinor basis. Following the nomenclature as used by Helgaker et al.,\(^{17}\) the restricted density matrices are referred to as Orbital1DM and Orbital2DM. To designate density matrices with possibly different components related to \(\alpha\)– and \(\beta\)-spin, we use the terminology SpinResolved1DM and SpinResolved2DM.

The types discussed above are dependent on the orbital basis in which they are expressed. In order to change their representation to that related to another orbital basis, GQCP provides the API transformed(T), with T a suitable YTransformation. In this way, a YTransformation can be used to basis-transform generalized objects, but not unrestricted or restricted objects.

A schematic overview of the concepts introduced in this section and their most important relations is given in Fig. 1. Code listing 2 combines them in a simple example calculation of the GHF dipole moment. Note that in order to make this example as conceptually simple as possible, we assume that the user has calculated the GHF model parameters in the variable gfh parameters, for example as a result of the calculation in code listing 4.
FIG. 1. Four high-level classes that GQCP offers and their quintessential relations. SpinorBases are linked to SQOperators through a quantize call, while SQOperators and DensityMatrices may be combined through calculate ExpectationValue. All four types are basis-transformable through the use of a Transformation.

Listing 2: Having optimized the GHF wave function model in the variable `ghf_parameters` (omitted in this example), a user can use this code snippet to calculate the GHF dipole moment.

```
# Set up the molecular system and spinor basis.
molecule = gqcp.Molecule.ReadXYZ("CO.xyz", charge=0)
spinor_basis = gqcp.GSpinorBasis_d(molecule, "6-31G**")

# (The optimization of the GHF wave function in the variable `ghf_parameters` is omitted.)

# Calculate the electronic dipole moment in the AO basis.
dipole_op = spinor_basis.quantize(gqcp.ElectronicDipoleOperator())  # 'op'
    for 'operator'.
D = ghf_parameters.calculateScalarBasis1DM()  # The 1-DM in AO basis.
dipole_moment = dipole_op.calculateExpectationValue(D)
```

The optimized GHF coefficients can be used to transform the operator and density matrix objects so they are expressed with respect to the GHF MOs. Code example 3 shows this behavior and calculates the GHF dipole moment, but now uses the density matrix and dipole operator expressed with respect to the GHF MOs. Note that the results obtained after executing code snippet 3 are equal to those obtained after executing code snippet 2.

```
# Transform the dipole operator and density matrix from their AO representation to an MO representation, and calculate the dipole moment in the MO basis.
```
Listing 3: Instead of calculating the GHF dipole moment through quantities expressed in the AO basis, users can calculate the GHF dipole moment using the dipole operator and density matrix in their MO representation. Note that listings 2 and 3 produce the same result.

Since the design of the computational objects reflects the theoretical framework behind spinor bases, GQCP can be used as a tool to explore electronic structure concepts in education,
\[11,51\] both from a user’s point of view and from a method developer’s perspective.

B. The algorithm framework

Another way that GQCP targets development flexibility is by focusing on modifiable algorithms. Since almost all computational methods use an interactive scheme to solve a set of equations, being able to intervene in these iterative cycles both at compile-time and at run-time drastically increases the sustainability, reproducibility, and the speed of the development of new computational methods. 

In contrast to only being able to modify heuristic parameters such as particular thresholds, subspace dimensions, and damping factors, GQCP allows for an intuitive way to modify the structure of algorithms and solvers, opening up otherwise black-box quantum chemical methods.

In GQCP, algorithms are formulated as sequential applications of their constituting steps that can each modify a so-called environment, which acts as a calculation “cache” and collects the previously calculated intermediates. Pictorially, this is represented in Fig. 2. This reformulation not only allows for a more rapid implementation of modifications to particular solvers in the C++ core library language, but also provides the means for users to define their custom algorithmic steps in Python and modify algorithms at run-time by injecting them in the basic algorithm. We emphasize that such modifications do not require a re-compilation of the whole project. For an illustrative application of this design, we refer the reader to Sec. III C.

SDE’s (Software Development Environment)\[14\] module design, where each module performs a particular task related to a quantum chemical method (such as calculating the energy, calculating the Fock matrix, etc.) is similar to what GQCP offers in its Algorithm framework: modules each perform their own functionality and have access to state, similarly to how different Steps may access an Environment and sequentially build up an IterativeAlgorithm.

The main difference is that in GQCP, algorithms are not designed based on inheritance, but as subsequent applications of their underlying steps. As an example, in SDE, one would override the behavior of a plain FockBuilder with DIIS acceleration, while in GQCP, one would insert the DIIS step in the plain SCF solver to allow the acceleration to take place. Similarly, a density-damped solver may be engineered by inserting the dampening step in a plain SCF solver.

As an example of how solvers and environments are typically combined in a quantum chemical method, we discuss the calculation of the GHF energy and spinor expansion coefficients. The corresponding code example is found in listing 4. In the first section, the set-up is performed by instantiating a molecule, orbital basis, and Hamiltonian. Then, a solver and an environment (with an initial guess that solves the general diagonalization problem for the core Hamiltonian) are prepared and are provided to the GHF QCMethod, whose optimize call returns a QCStructure. This quantum chemical structure encapsulates (energy, optimized parameters)-pairs and allows for a general representation of ground-state and excited state optimized parameters. The code example concludes with the extraction of the ground state (electronic) energy and the corresponding optimized GHF model parameters from the computed QCStructure.

```
C = ghf_parameters.expansion()  # The GHF spinor expansion coefficients.
dipole_op.transform(C)
D.transform(C)
dipole_moment = dipole_op.calculateExpectationValue(D)
```

FIG. 2. In GQCP, algorithms are formulated as (iterative) applications of its constituent steps, each with read-write capabilities to a calculation environment that encapsulates the algorithm’s state.
Listing 4: A complete code example that calculates the GHF energy and optimized model parameters for CO in a 6-31G∗∗ basis set. This code example can be modified to use complex orbitals by replacing the occurrences of _d with _cd.

C. ROHF through constrained UHF

A typical application of the flexibility of GQCP's algorithmic framework is found in the formulation of restricted open-shell Hartree–Fock (ROHF) as constrained unrestricted Hartree–Fock (CUHF). In this formalism, Tsuchimochi and Scuseria have reproduced ROHF energies through a constrained UHF procedure by replacing the α- and β-Fock matrices according to the ROHF constraint after each UHF SCF cycle.
Listing 5: Example code that performs a CUHF calculation on triplet molecular oxygen in a 6-31G basis set.

As far as other Hartree–Fock methods are concerned, GQCP provides real- and complex-valued restricted, unrestricted, and generalized Hartree–Fock (R/U/GHF) with internal and external stability analyses.\textsuperscript{49,50} Plain, DIIS\textsuperscript{53,54} and dampening solvers have been implemented in such a way that they can easily be modified or extended by the user as explained in Sec. III B.

D. CCSD with a GHF reference

Consider the scripting code in listing 6 that calculates the CCSD correlation energy using a GHF reference determinant. Note that, in order to simplify the code listings, we have opted not to include the code that performs the optimization of the HF spinors or spin-orbitals. For RHF, UHF and GHF code examples, we refer the reader to our website.\textsuperscript{82} After having transformed the quantities from the atomic to the molecular orbital basis, the CCSD environment is initialized by supplying an OrbitalSpace that encapsulates occupied and virtual orbital indices. Finally, the optimization of the CCSD wave function model and the retrieval of the correlation energy are performed in the usual manner through the optimize call.

Listing 6: Example code that calculates the CCSD correlation energy based on a GHF reference determinant.
Only minor modifications are required in order to calculate the CCSD correlation energy with a UHF determinant as a reference (cf. listing 7) or with an RHF determinant as a reference (cf. listing 8). One of the modifications is related to the re-formulation of the spin-orbital bases as generalized spinor bases, while the other required change corresponds to a different kind of ordering of the occupied and virtual orbitals.

In the GHF method formulation, the MOs are sorted with increasing orbital energy, while in UHF (and RHF), they are separated with respect to their spin. Therefore, the correct instance of OrbitalSpace is initialized through the GHF occupation number vector (ONV). In the GCCSD example, the GHF ONV would correspond to a set of 1s (denoting the occupied orbitals) followed by a set of 0s (denoting the virtual orbitals), while in the RCCSD and UCCSD examples, the GHF ONV corresponds to a set of 1s and a set of 0s, once related to the alpha spin-orbitals and once to the beta spin-orbitals. As an example, after a GHF calculation, the GHF ONV would be represented as 110000 for two electrons in six orbitals and as 100100 after an UHF or RHF calculation, where the alpha and beta orbitals are conceptually separated by the vertical bar |

```
# Transform from AO to MO basis.
C = uhf.parameters.expansion()
spin_orbital_basis.transform(C)

# Re-formulations related to generalized spinor bases.
spiner_basis = gqcpy.GSSpinorBasis_d.FromUnrestricted(spin_orbital_basis)
hamiltonian = spiner_basis.quantize(gqcpy.FQMolecularHamiltonian(molecule))

# Prepare the solver and environment.
solver = gqcpy.CCSDSolver_d.Plain()
M = spiner_basis.numberOfSpinors()
N = molecule.numberOfElectrons()
ghf_onv = gqcpy.SpinUnresolvedONV.GHF(M, N, uhf_parameters.
   spinOrbitalEnergiesBlocked()) # First the alpha orbital energies, then
   the beta orbital energies.
environment = gqcpy.CCSDEnvironment_d.PertubativeCCSD(hamiltonian, ghf_onv
   .orbitalSpace()) # Initialize the CCSD amplitudes from MP2 theory.

# Optimize and retrieve the CCSD correlation energy.
cccsd_qc_structure = gqcpy.CCSD_d.optimize(solver, environment)
cccsd_correlation_energy = cccsd_qc_structure.groundStateEnergy()
```

Listing 7: Example code that calculates the CCSD correlation energy with a UHF reference determinant.
Since the change of reference determinant requires the transformation of the operators and spin-orbital basis to their generalized equivalents, the GCCSD, UCCSD, and RCCSD methods formulated in this manner have exactly the same computational scaling since no simplifications are used with respect to spin symmetry. However, with GQCP’s aim in mind, the increase in flexibility and decrease in development time outweigh the minor decrease in execution speed.

As far as coupled-cluster (CC) methods are concerned, GQCP has implemented CCD (CC doubles) and CCSD (CC singles and doubles) both for RHF, UHF, and GHF reference determinants. Furthermore, we have implemented pair coupled-cluster doubles (pCCD) \(^{55,56}\) and orbital-optimized pCCD (OO-pCCD) using the dense orbital Hessian.

IV. SOFTWARE DEVELOPMENT IN GQCP

We strictly adhere to scientific software best practices as promoted by the institutions MolSSI, \(^{5,57,58}\) CECAM, \(^{13,59}\) and MaX. \(^{60}\) We spent considerable time refactoring our code, making sure that the APIs are semantically as clear as possible and that technical debt does not accumulate beyond reasonability.

GQCP’s focus has been to identify reusable software objects and, as a consequence, we have not attempted any forms of deliberate (i.e., hand-coded) parallelization, or targeted heterogeneous architectures specifically. By designing components that specifically separate the model (i.e., the abstract formulation of a solution) from computation (i.e., how the specifics are implemented under the hood), \(^{4,17}\) we focus on the implementation of the high-level characteristics of the problem at hand, while delegating the optimization and hardware acceleration to experts in their respective fields. \(^{4,17,61}\) Currently, we rely on underlying libraries such as Eigen \(^{62}\) (with interfaces to BLAS implementations) or Libint2 \(^{22}\) (for molecular integral calculations) for achieving parallelization in particular sub-calculations. By relying on lower-level adapter-like modules, \(^{61,63–70}\) our APIs can, in principle, proceed to scale to the exascale regime.

As GQCP’s focus is to provide useful generalizations, GQCP could serve as an initiative to further improve inter-module communication between the modules in the current electronic structure software ecosystem. \(^{11,13,14,18,20,21,71,72}\) Due to GQCP’s Python bindings, a prototype can typically be developed in a Jupyter Notebook, \(^{73}\) which allows for a flexible and high-level discussion between developers. Upon careful consideration related to possible generalizations, the actual functionality of the prototype is implemented in the C++ library, accompanied by corresponding Python bindings and relevant unit tests. Adhering to the Gitflow workflow, \(^{74}\) this work is typically done in a so-called feature branch and is merged with the develop branch only after an approving review by peer developers (who typically formulate feedback on the code style, documentation, etc.) and if the continuous integration processes do not report any failures for the full set of unit tests.

On a push to the develop or master branch, the project’s GitHub Actions \(^{83}\) automate both the creation of a new Docker

Listing 8: Example code that calculates the CCSD correlation energy with an RHF reference determinant.

```python
spinor_basis = gqcpy.GSpinorBasis_d.FromRestricted(spin_orbital_basis)
hamiltonian = spinor_basis.quantize(gqcpy.FQMolecularHamiltonian(molecule))

# Prepare the solver and environment.
solver = gqcpy.CCSDSolver_d Plain()

M = spinor_basis.numberOfSpinors()
N = molecule.numberOfElectrons()

ghf_env = gqcpy.SpinUnresolvedONV.GHF(M, N, rhf_parameters)
spinOrbitalEnergiesBlocked() # First the alpha orbital energies, then the beta orbital energies.

environment = gqcpy.CCSDEnvironment_d.PerturbativeCCSD(hamiltonian, ghf_env.orbitalSpace()) # Initialize the CCSD amplitudes from MP2 theory.

# Optimize and retrieve the CCSD correlation energy.
ccsd_qc_structure = gqcpy.CCSD_d_optimize(solver, environment)
ccsd_correlation_energy = ccsd_qc_structure.groundStateEnergy()
```
container image and a new conda installer, made public through Docker Hub\textsuperscript{75,76} and Anaconda,\textsuperscript{77,78} respectively. Existing users can then proceed to use newly implemented functionalities simply by rebuilding the Docker image on their infrastructure, or re-installing the conda package. Users who require HPC systems for their calculations can install GQCP through the EasyBuild\textsuperscript{79} installation framework, or can alternatively convert the Docker container to a Singularity\textsuperscript{80} container. As far as provisioning is concerned for developers, we provide a specific Docker container image for development, which covers the proper inclusion and linkage of all our dependencies, so that they can immediately focus on development rather than manually provisioning their systems. We note that we more than welcome feature requests, bug reports, and pull requests from the community through our GitHub repository.\textsuperscript{81}

V. SUMMARY

The Ghent Quantum Chemistry Package (GQCP) is an open-source electronic structure library that focuses on providing computational objects that correspond to the theoretical concepts behind two-component spinor bases. The resulting electronic structure building blocks can be used through both a C++ and a Python interface.

GQCP’s main area of focus is to allow rapid prototyping of novel computational methods by providing a flexible software framework that allows method developers to focus on the high-level formulation of the problem at hand. Furthermore, GQCP allows structural modifications of quantum chemical solvers at run-time and allows access to all lower-level intermediates and raw data types.

As the design of the objects itself semantically resembles the underlying theoretical concepts, GQCP is a helpful educational tool for exploring electronic structure concepts and can serve as an initiative to improve programmatic communication in-between modules of the current electronic structure software ecosystem.

AUTHORS’ CONTRIBUTIONS

L.L. is the software lead and visualization lead; he wrote the original draft and equally contributed in editing the draft. X.D.V. and D.T. equally contributed in the software development, reviewed the manuscript, and supported in editing. T.H. equally contributed in the software development. P.B. is the funding acquisition lead; he reviewed the manuscript and equally contributed in editing. G.A. is the editing lead; he equally contributed in the funding acquisition and software development and reviewed the manuscript.

ACKNOWLEDGMENTS

L.L. acknowledges support from an FWO Ph.D. fellowship (Grant No. 1126619N). Parts of this research were also funded by the FWO (Research Project No. G031820N). The computational resources (Stevin Supercomputer Infrastructure) and services used in this work were provided by the VSC (Flemish Supercomputer Center), funded by Ghent University, FWO, and the Flemish Government—Department EWI.

APPENDIX: THE CUHF REPLACEMENT FUNCTION

```python
def cuhf_function(environment):
    # Read the relevant intermediates from the environment.
    F = environment.fock_matrices[-1]
    D = environment.density_matrices[-1]
    N = environment.N # Number of alpha and beta electrons.

    # Form the closed-shell Fock matrix and the UHF modification in the AO basis.
    F_cs = (F.alpha + F.beta) / 2.0
    Delta_UHF = (F.alpha - F.beta) / 2.0

    # Form the modified Fock matrices.
    F_aa = F_cs + Delta_UHF
    F_bb = F_cs - Delta_UHF
```
# Find the natural occupation numbers and vectors by diagonalizing the
# charge-density matrix in an orthonormal basis. For the orthonormal
# basis, we use the Löwdin basis of the alpha basis functions.
X = spin_orbital.basis.lowdinOrthonormalization().alpha  # X transforms
# from the AO basis to the Löwdin basis.
P_AO = (D.alpha + D.beta) / 2.0
P_MO = P_AO.transformed(X)
natural_occupation_numbers, V = np.linalg.eigh(P_MO.matrix())  # Use
# NumPy for the diagonalization.

# Construct the Langrange multipliers to add them to the 'constrained'
# Fock matrices. They should be constructed in the basis of the
# natural occupations.
V = gqcpy.UTransformationComponent_d(V)  # V transforms from the Löwdin
# basis to the natural occupations.
Delta_UHF_NO = Delta_UHF.transformed(X).transformed(V)  # In the natural
# occupation (NO) basis.
Delta_UHF_NO = Delta_UHF_NO.parameters()
Lambda_NO = np.zeros(np.shape(Delta_UHF_NO))
Lambda_NO[N.beta, N.alpha] = -Delta_UHF_NO[N.beta, N.alpha]
Lambda_NO[N.alpha, N.beta] = -Delta_UHF_NO[N.alpha, N.beta]
Lambda_NO = gqcpy.ScalarUSQOneElectronOperatorComponent_d(Lambda_NO)
Lambda_AO = Lambda_NO.transformed(V.inverse()).transformed(X.inverse())
# In the AO basis.

# Overwrite the most recent UHF Fock matrix with the CUHF modification.
F_alpha_constrained = F_aa + Lambda_AO
F_beta_constrained = F_bb - Lambda_AO
F_constrained = gqcpy.ScalarUSQOneElectronOperator_d(F_alpha_constrained, F_beta_constrained)
environment.replace_current_fock_matrix(F_constrained)
Listing 9: A Python function that modifies the UHF Fock matrices according to the ROHF constraint and its subsequent insertion as an algorithmic step into a plain UHF SCF solver.

DATA AVAILABILITY

Data sharing is not applicable to this article as no new data were created or analyzed in this study.

REFERENCES

of Chemical Physics
The Journal

37. See https://www.cecam.org for CECAM (Centre Europeen de Calcul Atomique et Moleculaire - European Center of Atomic and Molecular Calculations); accessed 23 June 2021.


See https://github.com/GQCG/GQCP for the GQCP GitHub repository; accessed 23 June 2021.


See https://github.com/GQCG/GQCP for the GQCP GitHub repository; accessed 23 June 2021.


See https://github.com/features/actions for GitHub Actions: Automate your workflow from idea to production; accessed 23 June 2021.