# Quantum computing library for quantum chemistry applications

**K M Makushin**[1,2,*], **M D Sapova**[1], **and A K Fedorov**[1,†]

[1] Russian Quantum Center, Skolkovo, Moscow 121205, Russia
[2] Department of Theoretical Physics, Kazan Federal University, Kazan 420008, Russia

E-mail: *`k.makushin@rqc.ru`; †`akf@rqc.ru`

**Abstract.** Quantum computing is aimed to solve tasks, which are believed to be exponentially hard to existing computational devices and tools. A prominent example of such classically hard problems is simulating complex quantum many-body systems, in particular, for quantum chemistry. However, solving realistic quantum chemistry problems with quantum computers encounters various difficulties, which are related, first, to limited computational capabilities of existing quantum devices and, second, to the efficiency of algorithmic approaches. In the present work, we address the algorithmic side of quantum chemistry applications by introducing a Python 3 code library, whose primary objective is to speed up the development of variational quantum algorithms for electronic structure problems. We describe the various features and capabilities of this library, including its ease in constructing customized versions of variational quantum algorithms. We elucidate how the developed library allows one to design quantum circuits and enable for the efficient execution of quantum algorithms. Furthermore, the library facilitates the integration of classical and quantum algorithms for hybrid computations and helps to realize the cross-verification of data with traditional computational methods, thereby enhancing the overall reliability of quantum chemistry simulations.

## 1. Background

One of the motivations to build a quantum computer is the idea of solving complex quantum problems, which are considered to be hard for existing classical computing approaches. In particular, utilizing computational tools to predict the mechanisms and energetics of chemical reactions is of significant interest for both research and industrial applications [1, 2]. Quantum chemistry, among these tools, is especially valuable for predicting material properties because it predominantly relies upon quantum mechanics principles rather than empirical data. This ensures high precision in quantum chemical simulations but can also demand extensive computational resources [3]. Despite significant progress made in classical computing techniques in recent decades [4, 5], there are still limits that cannot be overcome in classical computations. The high complexity of high-precision *ab initio* methods of quantum chemistry makes them impractical for addressing industrially relevant problems. At the same time, the widely utilized density functional theory (DFT) often fails in accurately predicting chemical reactions involving transition metals, crucial components in catalytic processes [6]. Therefore, the development of new scalable, high-precision methods in computational chemistry continues to pose a significant challenge.

A promising approach lies in the use of quantum computers to simulate chemical systems.

The substantial entanglement observed in molecular wavefunctions suggests that quantum chemical simulations could benefit from such quantum computational capacities, although there is no proven quantum advantage yet [7]. However, the capabilities of the currently available generation of quantum processors are not yet enough to solve such problems at the relevant scale. First of all, in order to make them more powerful not only the number of qubits should be increased, but also the quality of quantum operations is needed to be improved. In addition to that, an important task is to design resource-efficient quantum algorithms. Quantum chemistry algorithms based on the use of quantum computing can be divided into two large classes: (i) methods for fault-tolerant quantum computers and (ii) approaches for noisy intermediate-scale quantum (NISQ) devices [8]. Algorithms for NISQ devices take into account the limitations of the existing quantum processors, so they require low circuit depths that allow executing within the limited coherence time of the device.

In recent years, variational quantum algorithms (VQAs) [9, 10] have received a significant attention due to their flexibility and potential applications with the NISQ devices [8]. Although it remains uncertain whether quantum advantage can be generally achieved with VQAs [9], the development and testing new, efficient strategies is crucial for identifying potential applications of quantum computers. The variational principle [10], which states that the energy of the trial wavefunction $\phi(\theta)$ always provides an upper bound to the true ground state energy, is a key idea of variational quantum computations:

$$\langle\phi(\theta)|H|\phi(\theta)\rangle = \langle\Psi_{gs}|H|\Psi_{\text{gs}}\rangle \geq E_{\text{gs}}. \tag{1}$$

Following the variational principle, one can find an approximate eigenstate of the system by determining the optimal parameters of the trial wavefunction that minimize the energy. The form of an efficient trial wavefunction depends on the problem at hand and is referred to as an 'ansatz'. VQAs leverage quantum computers to prepare the ansatz as a parameterized quantum circuit, this procedure can be denoted as $U(\theta)|0\rangle$. The energy (or more complex cost function) of the ansatz is then measured, and the circuit parameters are updated using classical optimization algorithms. This process of state preparation, measurement, and parameter updating is repeated until convergence criteria are met. The flexibility of VQAs arises from the ability to select different quantum circuits, cost functions, and optimization algorithms for the same problem. Therefore, a user-friendly and straightforward tool is needed, one that enables researchers to implement customized versions of VQAs, thereby accelerating the development process.

In the present work, we address the algorithmic side of quantum chemistry applications. We introduce a Python 3 code library, whose primary objective is to speed up the development of variational quantum algorithms for the resolution of electronic structure problems. We describe the various features and capabilities of this library, including its ease in constructing customized versions of variational quantum algorithms. We elucidate how it allows one to design quantum circuits and enable efficient execution of quantum computations. Furthermore, the library facilitates the integration of classical and quantum algorithms for hybrid computations and helps to realize the cross-verification of data with traditional computational methods, thereby enhancing the overall reliability of simulations.

## 2. Quantum chemistry problem statement

One of the key applications, in which quantum computers could make a significant impact is in the realm of non-relativistic quantum chemical simulations, typically performed within the Born-Oppenheimer approximation. This approximation assumes the separation of electronic and nuclear motion due to mass difference of three orders of magnitude. As a result, the molecular wavefunction is expressed as a product of the electronic wavefunction, which depends on the nuclear coordinates $\mathbf{R}$ only parametrically, and the nuclear wavefunction [11, 12]:

$$|\Psi\rangle = |\Phi_{\text{elec}}(\mathbf{r}, \mathbf{R})\rangle|\chi_{\text{nuc}}(\mathbf{R})\rangle. \tag{2}$$

The Schrödinger equation is thus split into the electronic (3) and nuclear (4) equations, the latter requires the electronic potential calculated from the first equation:

$$H_{\text{elec}}|\Phi_{\text{elec}}(\mathbf{r},\mathbf{R})\rangle = E_{\text{elec}}|\Phi_{\text{elec}}(\mathbf{r},\mathbf{R})\rangle, \tag{3}$$

$$(T_N(\mathbf{R}) + E_{\text{elec}}(\mathbf{R}))|\chi_{\text{nuc}}(\mathbf{R})\rangle = E_{\text{tot}}|\chi_{\text{nuc}}(\mathbf{R})\rangle, \tag{4}$$

Solving the electronic Schrödinger equation is the central problem in quantum chemistry and it is then of the greatest interest in the context of quantum computing [13, 14]. In this paper, we focus on the electronic structure problem (electronic equation).

It is important to bear in mind, however, that despite the efficiency and simplicity of the Born-Oppenheimer approximation in simplifying quantum chemical calculations, it is not always an accurate representation of molecular systems. This is especially true when studying chemical reactions that involve transitions between electronic states, i.e., photochemical reactions, electron or charge transfer [15].

## 3. Library

Our software is written in Python 3 and relies on its several libraries: NumPy for numerical computations, PySCF for calculating molecular integrals, Qiskit Terra for quantum circuit construction, and SciPy for classical optimization [16]. The current version of our software is compatible with the *AerSimulator* from Qiskit and the *default.qubit* simulator provided by PennyLane [17]. A key feature of our software is its ability to easily construct custom versions of variational quantum algorithms. The main steps of this procedure are as follows [9, 18]:

- **Construction of the problem Hamiltonian in the qubit representation**. The current version supports built-in construction of molecular Hamiltonians for the selected active space in both second-quantized and qubit representations [19]. Alternatively, users can manually define matrix, second-quantized, or qubit Hamiltonians.
- **Construction of the ansatz**. Users are able to employ built-in ansatzes (such as unitary coupled cluster [20] and hardware-efficient ansatzes), or they can create a custom ansatz (for more details, see Subsec. 3.3).
- **Construction of the cost function**. The cost function can be defined as a weighted sum of the expected values of operators and overlaps [21, 22, 23].
- **Construction of the classical optimizer**. The current version supports only SciPy's built-in optimizer.

In the following sections, we describe the main building blocks of the library and demonstrate common use cases.

### *3.1. Models*

To automatically construct a problem qubit Hamiltonian, one can utilize the built-in models. The current version of the library includes only a molecular model (represented by the `Molecule` class), which can be used to calculate basic properties of molecular systems. In future versions, we plan to expand the set of models to include the Fermi-Hubbard model.

The `Molecule` class obtains information about electronic one- and two-body integrals, molecular symmetry, and molecular energies from the PySCF library [24]. Integrals are computed with the selected atomic basis (the full list of available basis sets can be found in the PySCF documentation [25]).

Now we describe the key features of the `Molecule` class:

- **Manual definition of frozen molecular orbital indices**. Selecting active orbitals and freezing the occupation number of other orbitals (2 for core orbitals, 0 for virtual) for electron correlated methods is a widely use technique in quantum chemical simulation. This technique allows one to reduce the computational cost of the simulation, which is especially important for current quantum simulations. Our library includes the ability to manually define the indices of the frozen molecular orbitals, giving researchers greater control over the active space.
- **Restricted and unrestricted Hartree-Fock (HF) orbitals**. Molecular Hamiltonian can be build with the restriction for the spatial component of corresponding $\alpha$ and $\beta$ orbitals to be the same (restricted is RHF) or without it (unrestricted is UHF).
- **Reference energies from classical methods**. To validate the accuracy of the quantum algorithm it is common to compute energies with Hartree-Fock and electron correlated methods [coupled cluster (CC) and configuration interaction (CI)]. Built-in launch of PySCF's HF, CCSD (singles doubles), CASCI (complete active space) computations for both spin-restricted and unrestricted HF orbitals in Molecule provides a reference point.
- **Two spin-orbital notations (qiskit style, openfermion style)**. There are two widely used conventions for representing spin-orbitals in quantum chemistry simulations: (i) the Qiskit convention ($\alpha$ spin-orbitals first, $\beta$ spin-orbitals second: $\alpha ... \alpha \beta ... \beta$) and (ii) the OpenFermion convention ($\alpha\beta\alpha\beta ... \alpha\beta$).

Below we demonstrate some examples. Input parameters of a molecule are defined in config file (file.yaml), which follows PySCF notation:

```yaml
model: molecule
molecule_info:
  geometry: [["Li", [0., 0., 0.]], ["H", [0., 0., 1.5]]]
  basis_set: sto-3g
  spin: 0
  symmetry: True
  abelian: True
  charge: 0
  method: rhf
  frozen_orbs: [null, [0], [0, 5]]
```

**Listing 1.** Example of LiH molecule configuration in .yaml file format.

In the above example, *abelian* is an optional parameter that should be used together with qubit tapering procedure to reduce the symmetry of the molecule to abelian subgroup. Parameter *frozen_orbs* determines the indices of frozen molecular orbitals (MOs). If multiple simulations with different active spaces is required, it is convenient to define multiple sets of frozen orbitals simultaneously. In the aforementioned example for LiH, we define three different sets of frozen orbitals, which result in 6, 5, and 4 MOs in the active space, respectively.

The `Molecule` class can be used further to construct the electronic molecular Hamiltonian in the selected active space, which is expressed in the qubit representation (see the information about qubit operators below).

```python
from src.models import Molecule
from omegaconf import OmegaConf
from src.operators import FermionicOperator

config_path = "src/configs/molecules/lih_rhf_sto3g_eq.yaml"
config = OmegaConf.load(config_path)
config._set_item_impl ("path", config_path)
mol = Molecule(config)
mapping = "jordan_wigner"
```

```
frozen_inds = frozen_orbs[0]
hamiltonian = mol.build_hamiltonian(frozen_inds=frozen_inds, notation=Notation
  .QISKIT, mapping=mapping)
```

**Listing 2.** Getting a qubit Hamiltonian using a `Molecule` class

Additionally, one can obtain results of classical computations, such as Hartree-Fock, CCSD, and CASCI [26]. The latter two requires to define the indices of frozen orbitals following PySCF MO notation.

```
from src.models import Molecule
from omegaconf import OmegaConf
from src.operators import FermionicOperator

config_path = "src/configs/molecules/lih_rhf_sto3g_eq.yaml"
config = OmegaConf.load(config_path)
config._set_item_impl ("path", config_path)
mol = Molecule(config)
freeze_inds = config.molecule_info.frozen_orbs[-1]

print("FREEZE INDS", freeze_inds)
print("HARTREE-FOCK ENERGY:", mol.hf.e_tot)
print("CCSD ENERGY:", mol.ccsd(freeze_inds).e_tot)
print("CASCI ENERGY:", mol.fci(freeze_inds).e_tot)

OUTPUT:
FREEZE INDS None
HARTREE-FOCK ENERGY: -1.1161514489386022
CCSD ENERGY: -1.1371172451631746
CASCI ENERGY: -1.1371170673457307
```

**Listing 3.** The results for the $H_2$ molecule obtained using classical methods.

### *3.2. Operators*

The quantum simulation of electronic systems requires a specific encoding of the wavefunction and operators within the qubit space. Moreover, such encoding is needed to preserve the properties of the simulated system, which in the case of fermions includes antisymmetrization. The developed encodings for fermionic problems can be categorized into first-quantized and second-quantized encodings. The latter, despite having worse scalability, require significantly fewer qubits for small- and mid-sized systems, making them suitable for NISQ devices. Among the second-quantized encodings, Jordan-Wigner [27], Bravyi-Kitaev [28], and parity are the most commonly used. In our code, we use molecular integrals from PySCF to construct a second-quantized representation of fermionic operators, and then perform the Jordan-Wigner transformation to obtain qubit operators.

The `Operator` class is the building block of our code. It handles second-quantized fermionic operators and qubit operators. Operators can be added, subtracted, or multiplied.

*Fermionic operators* The `FermionicOperator` class represents operators in fermionic systems within the second-quantized formalism [13], specifically, in terms of creation and annihilation operators. The creation operator $a_i^\dagger$ creates a fermion at the $i$th mode, while the annihilation operator $a_i$ destroys a fermion at the $i$th mode. Each interaction in the fermionic system can be expressed in the tensor product basis of creation and annihilation operators.

Fermionic operators obey standard the anti-commutation relations:

$$\begin{aligned} \{a_i, a_j\} = \{a_i^\dagger, a_j^\dagger\} = 0, \\ \{a_i^\dagger, a_j\} = a_i^\dagger a_j + a_j a_i^\dagger = \delta_{ij}, \end{aligned} \tag{5}$$

where $i$ and $j$ are indices of the single-particle state (or simply spin-orbital), and $\delta_{ij}$ is the Kronecker delta, which is 1 if $i = j$, and 0 otherwise.

In the current version of the library fermionic operators can be build from molecular integrals using method *from_integrals()*:

```
from src.models import Molecule
from omegaconf import OmegaConf
from src.operators import FermionicOperator

config_path = "src/configs/molecules/h2_rhf_sto3g_eq.yaml"
config = OmegaConf.load(config_path)
config._set_item_impl("path", config_path)
mol = Molecule(config)
freeze_inds = config.molecule_info.frozen_orbs[-1]
integrals = mol.transform_mo2so(freeze_inds)
fermionic_op = FermionicOperator.from_integrals(integrals)
print("FERMIONIC OPERATOR", fermionic_op.inp_dict)

OUTPUT:
FERMIONIC OPERATOR defaultdict(<class 'complex'>, {'+0 -0':
  (-1.2472845052236152+0j), '+1 -1': (-0.48127293109598346+0j), '+2 -2':
  (-1.2472845052236152+0j), '+3 -3': (-0.48127293109598346+0j), '+0 +1 -0 -1':
   (0.09088576828865244+0j), '+0 +1 -1 -0': (0.33098862973957277+0j), '+0 +2
  -2 -0': (0.33642397347431435+0j), '+0 +2 -3 -1': (0.09088576828865239+0j), '
  +0 +3 -2 -1': (0.09088576828865244+0j), '+0 +3 -3 -0':
  (0.33098862973957277+0j), '+1 +0 -0 -1': (0.330988629739573+0j), '+1 +0 -1
  -0': (0.09088576828865237+0j), '+1 +2 -2 -1': (0.330988629739573+0j), '+1 +2
   -3 -0': (0.09088576828865237+0j), '+1 +3 -2 -0': (0.09088576828865245+0j),
  '+1 +3 -3 -1': (0.3479075755298825+0j), '+2 +0 -0 -2':
  (0.33642397347431435+0j), '+2 +0 -1 -3': (0.09088576828865239+0j), '+2 +1 -0
   -3': (0.09088576828865244+0j), '+2 +1 -1 -2': (0.33098862973957277+0j), '+2
   +3 -2 -3': (0.09088576828865244+0j), '+2 +3 -3 -2': (0.33098862973957277+0j
  ), '+3 +0 -0 -3': (0.330988629739573+0j), '+3 +0 -1 -2':
  (0.09088576828865237+0j), '+3 +1 -0 -2': (0.09088576828865245+0j), '+3 +1 -1
   -3': (0.3479075755298825+0j), '+3 +2 -2 -3': (0.330988629739573+0j), '+3 +2
   -3 -2': (0.09088576828865237+0j)})
```

**Listing 4.** Building a `FermionicOperator` using method *from_integrals()* and config for $H_2$ molecule

or directly by providing a `defaultdict` dictionary class:

```
from src.operators import FermionicOperator
from collections import defaultdict
fop = FermionicOperator(defaultdict(complex, {"+1 +2 -3 -4": 1.29 + 0.4j, "+2
  -4": -0.05 - 0.12j}), num_qubits=5)
print("FERMIONIC OPERATOR", fop.inp_dict)

OUTPUT:
FERMIONIC OPERATOR defaultdict(<class 'complex'>, {'+1 +2 -3 -4': (1.29+0.4j),
   '+2 -4': (-0.05-0.12j)})
```

**Listing 5.** Building a `FermionicOperator` using a `defaultdict` dictionary.

For further application of the Jordan-Wigner transformation, the `FermionicOperator` object must be transformed into the normal-ordered form. In other words, the product of creation and annihilation operators must be rearranged as follows:

(i) all creation operators $a_i^\dagger$ are placed to the left of all annihilation operators $a_j$;

(ii) within each group of creation or annihilation operators, the operators are ordered according to their indices with operators corresponding to lower indices appearing first.

*Qubit operators* The `QubitOperator` class represents an operator acting in multi-qubit space. For NISQ applications qubit operators are represented in the Pauli tensor product basis [29]. Within the library, one can build qubit operators from fermionic operators or a matrix using two methods: *from_fermionic_operator()* and *from_matrix()*. We note that the matrix is needed to be Hermitian:

```
from src.operators import QubitOperator, FermionicOperator
from collections import defaultdict
fop = FermionicOperator(defaultdict(complex, {"+1 -2 +4 -3": 1.29 + 0.4j, "-2
  +4": -0.05 - 0.12j}), num_qubits=5).operator_to_normal_order()
qop = QubitOperator.from_fermionic_operator(fop)

print("FERMIONIC OPERATOR: ", fop.inp_dict)
print("QUBIT OPERATOR: ", qop.inp_dict)

OUTPUT:
FERMIONIC OPERATOR:  defaultdict(<class 'complex'>, {'+1 +4 -2 -3': (-1.29-0.4
  j), '+4 -2': (0.05+0.12j)})
QUBIT OPERATOR:  defaultdict(<class 'complex'>, {'IXXXX': (0.080625+0.025j), '
  IYYYY': (0.080625+0.025j), 'IXXYX': (-0.025+0.080625j), 'IXYXX':
  (-0.025+0.080625j), 'IXXXY': (0.025-0.080625j), 'IYXXX': (0.025-0.080625j),
  'IXYYY': (-0.025+0.080625j), 'IYYYX': (-0.025+0.080625j), 'IYXYY':
  (0.025-0.080625j), 'IYYXY': (0.025-0.080625j), 'IXXYY': (0.080625+0.025j), '
  IXYXY': (0.080625+0.025j), 'IYXYX': (0.080625+0.025j), 'IYYXX':
  (0.080625+0.025j), 'IYXXY': (-0.080625-0.025j), 'IXYYX': (-0.080625-0.025j),
   'IIYZX': (0.03-0.0125j), 'IIYZY': (0.0125+0.03j), 'IIXZX': (0.0125+0.03j),
  'IIXZY': (-0.03+0.0125j)})
```

**Listing 6.** Buiding a `QubitOperator` instance from a `FermionicOperator` class object

Here. the method *operator_to_normal_order()* is used to make `FermionicOperator` normal ordered. Also a `QubitOperator` can be built from a `defaultdict` dictionary just like we did before for a `FermionicOperator`:

```
qop = QubitOperator(defaultdict(complex, {"XZZY": 1.29 + 0.4j, "YZZX": -0.05 -
   0.12j, "ZZZZ": 0.25}), num_qubits=4)
```

**Listing 7.** Buiding a `QubitOperator` from a `defaultdict` dictionary

For commuting terms in the set of qubit operators, a basis exists in which these terms can be measured simultaneously using a single quantum circuit. Reducing the number of measured circuits is extremely important to accelerate computation and reduce its cost, especially since molecular qubit Hamiltonians can include hundreds of terms even for the smallest molecules. In our code, terms within `QubitOperator` class instances can be grouped based on qubit-wise or general commutativity [30] (note that general commutativity is not available in the current version of the library):

```
qop.group_terms_and_measurements(group_type="qubit_wise")
print("GROUPED OPERATORS", qop.term_dict_cache)

OUTPUT:
{'qubit_wise': {('XZZY',): (([0, 1, 2, 3],), ('ZZZZ',), <qiskit.circuit.
  quantumcircuit.QuantumCircuit object at 0x7fc74949d7f0>), ('YZZX',): (([0,
  1, 2, 3],), ('ZZZZ',), <qiskit.circuit.quantumcircuit.QuantumCircuit object
```

```
    at 0x7fc748c31be0>), ('ZZZZ',): (([0, 1, 2, 3],), ('ZZZZ',), <qiskit.circuit
    .quantumcircuit.QuantumCircuit object at 0x7fc748c60a00>)}}
```

**Listing 8.** An example of using qubit-wise Pauli operators method of grouping.

In the presence of symmetries, the amount of qubits necessary to simulate a molecular Hamiltonian can be decreased. The library offers methods for eliminating qubits from Hamiltonians that include numerous $Z_2$ symmetries (qubit tapering [31]):

```
from src.models import Molecule
from omegaconf import OmegaConf
from src.operators import FermionicOperator, QubitOperator

config_path = "src/configs/molecules/h2_rhf_sto3g_eq.yaml"
config = OmegaConf.load(config_path)
config._set_item_impl("path", config_path)
mol = Molecule(config)
freeze_inds = config.molecule_info.frozen_orbs[-1]

qubit_op = mol.build_hamiltonian(freeze_inds)
print("HAMILTONIAN:", qubit_op.inp_dict)
occ_orbs, virt_orbs = mol.occupied_virtual_orbs(freeze_inds=freeze_inds)

# Define active electrons to get tapered operator
active_electrons = mol.active_electrons(freeze_inds)
print("ACTIVE ELECTRONS", active_electrons)
tap_op = qubit_op.taper(active_electrons)
print("TAPERED HAMILTONIAN:", tap_op.inp_dict)

OUTPUT:
HAMILTONIAN: defaultdict(<class 'complex'>, {'IIII': (-0.8153001706270068+0j),
    'ZIII': (0.16988452027940376+0j), 'IZII': (-0.2188630678121962+0j), 'IIZI':
    (0.16988452027940373+0j), 'IIIZ': (-0.2188630678121962+0j), 'ZZII':
   (0.12005143072546023+0j), 'ZIZI': (0.16821198673715718+0j), 'YYYY':
   (0.04544288414432621+0j), 'YYXX': (0.04544288414432621+0j), 'XXYY':
   (0.04544288414432621+0j), 'XXXX': (0.04544288414432621+0j), 'ZIIZ':
   (0.16549431486978644+0j), 'IZZI': (0.16549431486978644+0j), 'IZIZ':
   (0.17395378776494125+0j), 'IIZZ': (0.12005143072546023+0j)})
ACTIVE ELECTRONS (1, 1)
TAPERED HAMILTONIAN: defaultdict(<class 'complex'>, {'I':
   (-1.0442258873154011+0j), 'Z': (0.7774951761831996+0j), 'X':
   (0.18177153657730474+0j)})
```

**Listing 9.** An example of using qubit tapering procedure for $H_2$ molecule

In the Listing 9 example, one can see that three qubits can be removed from the original operators, so simulating the new Hamiltonian requires less computational resources.

For computing the expectation value of the qubit operator on the given state (circuit) $\langle 0|U(\theta)^\dagger HU(\theta)|0\rangle$ one can use *compute_expectation()* method:

```
from qiskit import QuantumCircuit
from src.operators import QubitOperator
from collections import defaultdict

num_qubits = 4
qop = QubitOperator(defaultdict(complex, {"XZZY": 1.29, "YZZX": -0.05, "ZZZZ":
    0.25}), num_qubits=num_qubits)
qc = QuantumCircuit(num_qubits, num_qubits)
qc.ry(0.2, 0)
qc.h(0)
```

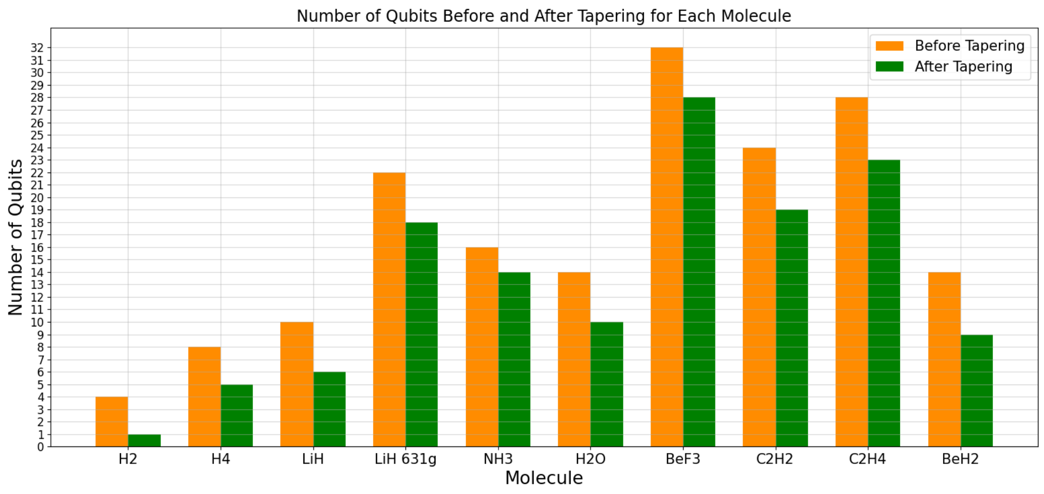

**Figure 1.** The change in the number of qubits after applying the tapering procedure for different molecules.

```
qc.cnot(0, 1)

expectation_value = qop.compute_expectation(qc)
print("EXPECTATION VALUE:", expectation_value[0])

OUTPUT:
EXPECTATION VALUE: 0.226953125
```

**Listing 10.** Computation of the expectation value of a `QubitOperator` on a given state

## 3.3. *Ansatzes*

The ansatz consists of a parameterized quantum circuit, the parameters of which can be fine-tuned to obtain the most appropriate solution. The choice of the ansatz is critical in determining the success of the quantum algorithm as it influences the speed of convergence, the accuracy of the solution, and the ability to scale to larger problem sizes. Thus, finding the optimal ansatz typically involves an iterative process that requires both expertise and intuition. Usually, the ansatz is selected based on prior knowledge of the problem and trial-and-error techniques. Currently, there are four classes of ansatzes available:

- `CustomAnsatz` class. This class requires electron excitations or a set of fermion operators or a set of qubit operators. It transforms the input data in the following sequence: Electron Excitation → Fermionic Operators → Qubit operators. The circuit is constructed by sequentially adding a circuit block to the end of the circuit, where each block is the gate decomposition of the exponent of the qubit operator.

```
from src.ansatzes import CustomAnsatz

ansatz = CustomAnsatz(excitations=[((1, ), (3, )), ((2, 3), (0, 1))],
  num_qubits=4)
qc = ansatz.build_circuit()
qc.draw(output='mpl')
print("FERMIONIC OPERATORS", ansatz.fermionic_operators[1].inp_dict)
print("QUBIT OPERATORS", ansatz.qubit_operators[1].inp_dict)

OUTPUT:
```

```
FERMIONIC OPERATORS defaultdict(<class 'complex'>, {'+0 +1 -2 -3': (1+0j),
   '+2 +3 -0 -1': (-1+0j)})
QUBIT OPERATORS defaultdict(<class 'complex'>, {'XXXY': -0.125j, 'XXYX':
  -0.125j, 'XYXX': 0.125j, 'YXXX': 0.125j, 'XYYY': -0.125j, 'YXYY': -0.125
  j, 'YYXY': 0.125j, 'YYYX': 0.125j})
```

**Listing 11.** `CustomAnsatz` example, here the ansatz is generated by two excitations which were passed manually

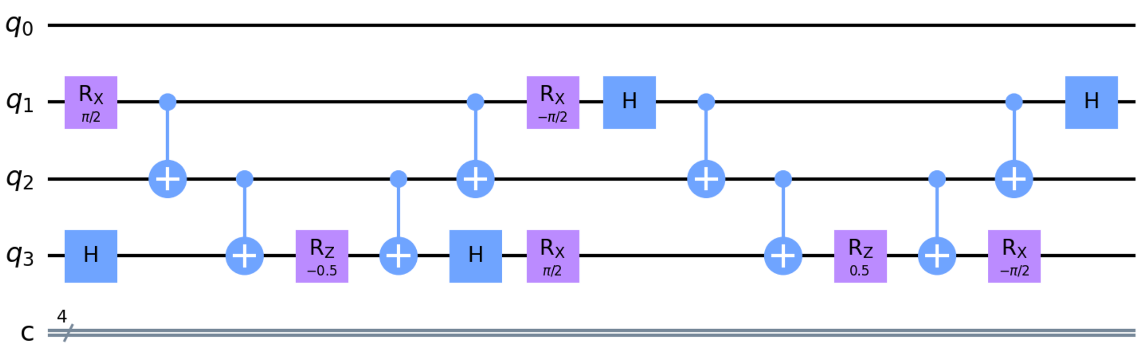

**Figure 2.** The circuit for the first of `CustomAnsatz` excitations used in Listing 12.

- `UCCAnsatz` class. This class represents the disentangled unitary coupled cluster ansatz (it is a descendant of the `CustomAnsatz`). Unlike the `CustomAnsatz`, the set of excitations in the unitary coupled cluster is predefined for the given number of electrons and orbitals, and it depends on the selected truncation level for the excitations (i.e., UCCSD includes single and double excitations). In our code, one can select any types of excitations to include, i.e., (1, 2) build a UCCSD ansatz, (1, 3) build a UCCST ansatz. Moreover, three types of excitations are built-in: *conventional* (excitations from occupied to unoccupied orbitals), *generalized* (excitations between any orbitals), and *paired* (a pair of electrons with opposite spin excite simultaneously) [20, 32]. The latter type can be combined with *conventional* or *generalized* excitations.

```
from src.ansatzes import UCCAnsatz
from src.models import Molecule
from omegaconf import OmegaConf

config_path = "src/configs/molecules/lih_rhf_sto3g_eq.yaml"
config = OmegaConf.load(config_path)
config._set_item_impl("path", config_path)
mol = Molecule(config)
freeze_inds = config.molecule_info.frozen_orbs[-1]
occ_orbs, virt_orbs = mol.occupied_virtual_orbs(freeze_inds=freeze_inds)
ansatz = UCCAnsatz(occ_orbs, virt_orbs, excitation_order_list=(1, 2),
  excitation_type="conventional")
ansatz_g = UCCAnsatz(occ_orbs, virt_orbs, excitation_order_list=(1, 2),
  excitation_type="generalized")
ansatz_cp = UCCAnsatz(occ_orbs, virt_orbs, excitation_order_list=(1, 2),
  excitation_type="conventional", paired=True)

print("CONVENTIONAL EXCITATIONS", ansatz.excitations)
print("GENERALIZED EXCITATIONS", ansatz_g.excitations)
print("CONVENTIONAL PAIRED EXCITATIONS", ansatz_cp.excitations)

OUTPUT:
CONVENTIONAL EXCITATIONS [((0,), (1,)), ((0,), (2,)), ((3,), (4,)), ((3,),
   (5,)), ((0, 3), (1, 4)), ((0, 3), (1, 5)), ((0, 3), (2, 4)), ((0, 3),
  (2, 5))]
GENERALIZED EXCITATIONS [((0,), (1,)), ((0,), (2,)), ((1,), (2,)), ((3,),
  (4,)), ((3,), (5,)), ((4,), (5,)), ((0, 3), (1, 4)), ((0, 3), (1, 5)),
  ((0, 3), (2, 4)), ((0, 3), (2, 5)), ((0, 4), (1, 5)), ((0, 4), (2, 5)),
  ((1, 3), (2, 4)), ((1, 3), (2, 5)), ((1, 4), (2, 5))]
CONVENTIONAL PAIRED EXCITATIONS [((0, 3), (1, 4)), ((0, 3), (2, 5))]
```

**Listing 12.** `UCCAnsatz` example with conventional, generalized, and conventional paired type excitations of 1st and 2nd order

One option to reduce the depth of the UCCSD ansatz is to filter out all excitations that lead to irreducible representation other than the Hartree-Fock determinant's [33]. One can do this by passing True to the *irrep_symmetry* argument of a `UCCAnsatz` class instance:

```
ansatz = UCCAnsatz(occ_orbs, virt_orbs, excitation_order_list=(1, 3),
  excitation_type="conventional", irrep_symmetry=True)
```

**Listing 13.** `UCCAnsatz` example with enabled filtering of excitations

- `HardwareEfficientAnsatz` class. This class provides basic hardware-efficient ansatz templates constructed from single-qubit rotations and CNOTs with various qubit connectivity options: *full*, *linear*, and *circular*.

```
from src.ansatzes import HardwareEfficientAnsatz

num_layers = 2
parameters = np.random.uniform(-np.pi, np.pi, num_qubits*2*num_layers)
hwe_ansatz = HardwareEfficientAnsatz(num_qubits, num_layers=num_layers,
  connectivity="chess")
hwe_qc = hwe_ansatz.build_circuit(parameters)
```

**Listing 14.** `HardwareEfficientAnsatz` example with 2 layers and *chess* connectivity of CNOT gates

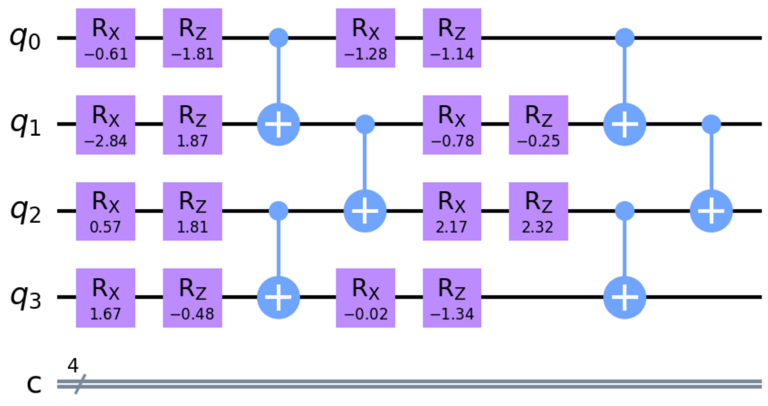

**Figure 3.** The quantum circuit with the *chess* structure of `HardwareEfficientAnsatz`.

- `ManualAnsatz` class
  This class is designed for cases where users want to manually create an ansatz. Its purpose is to transform the circuit into a form compatible with the VQA framework.

```python
from qiskit.circuit import QuantumCircuit

def make_circuit(num_qubits):
  circuit = QuantumCircuit(num_qubits, num_qubits)
  circuit.ry(0.25, 0)
  circuit.ry(0.5, 1)
  circuit.cx(0, 1)
  circuit.ry(0.33, 0)
  circuit.ry(0.27, 1)
  circuit.cx(0, 1)
  return circuit

num_qubits, num_layers = 2
circuit = make_circuit(num_qubits)
ansatz = ManualAnsatz(num_qubits=num_qubits, ansatz_state=circuit,
  num_layers=n_layers)
```

**Listing 15.** `ManualAnsatz` example the ansatz state built with *make_circuit* function

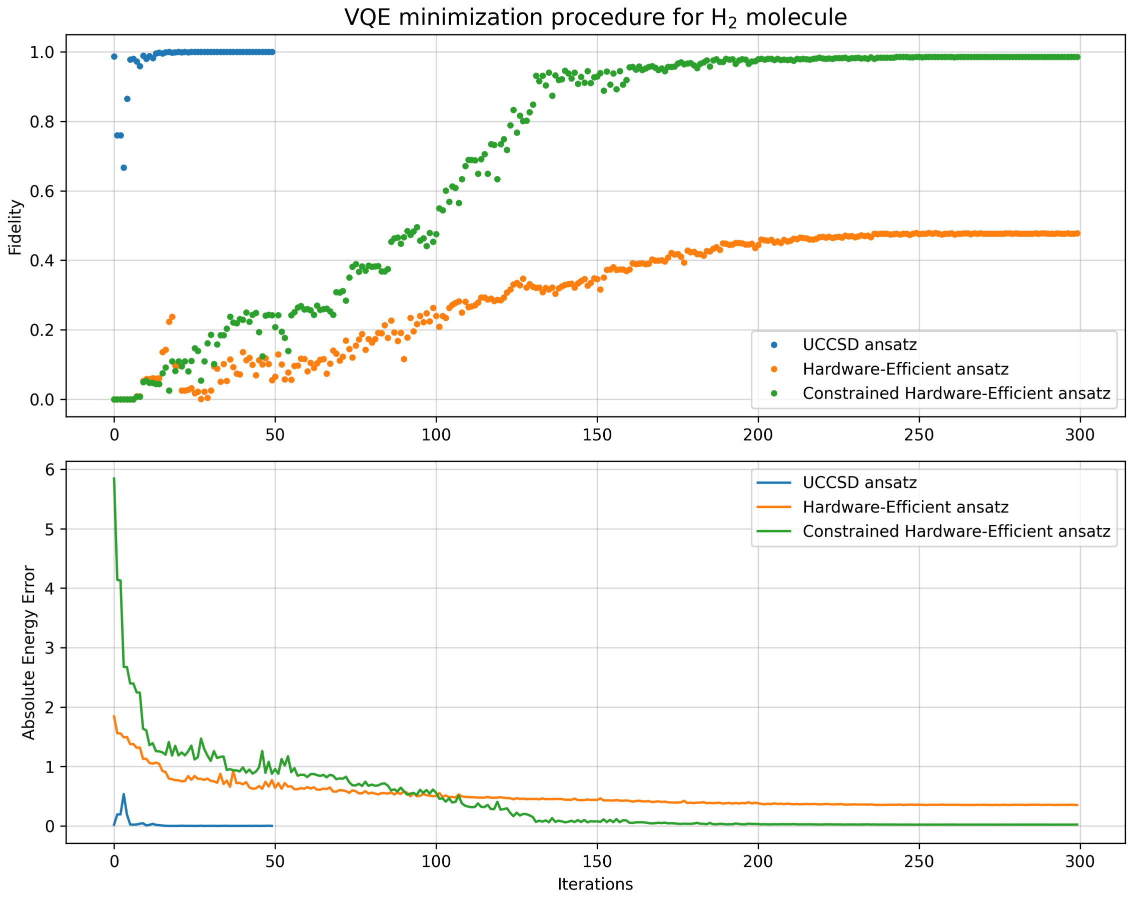

**Figure 4.** Convergence behavior for various ansatz types for the hydrogen molecule.

For each type of ansatz, one can define an initial state (for UCC, this is the Hartree-Fock state) and initial ansatz parameters, as shown in the following listing:

```python
from src.symmetry_utils import hartree_fock_state
import numpy as np
from src.models import Molecule
from omegaconf import OmegaConf

config_path = "src/configs/molecules/h2_rhf_sto3g_eq.yaml"
config = OmegaConf.load(config_path)
config._set_item_impl("path", config_path)
mol = Molecule(config)
freeze_inds = config.molecule_info.frozen_orbs[-1]

init_state = hartree_fock_state(num_qubits, mol.active_electrons(freeze_inds))
ansatz = UCCAnsatz(occ_orbs, virt_orbs, excitation_order_list=(1, 2),
excitation_type="conventional", init_state=init_state)
parameters = np.random.uniform(-np.pi, np.pi, len(ansatz.excitations))

qc = ansatz.build_circuit(parameters)
```

**Listing 16.** `UCCAnsatz` example with initial state and parameters passed

### *3.4. Variational Quantum Algorithms*

Within the developed library, variational algorithms with customizable cost functions can be built using the `VariationalAlgorithm` class. This ability to customize the cost function enables one to quickly optimize and improve existing algorithms. Further examples demonstrate how different algorithms can be realized within our framework. The required cost function typically takes the following form:

$$L = w_0\langle U_0|E_0|U_0\rangle + ... + w_n\langle U_n|E_n|U_n\rangle + o_1\langle U_0|U_1\rangle^2 + ... + o_n\langle U_i|U_j\rangle^2 + \\ c_0\left(\langle U_0|C_0|U_0\rangle - C_0^{val}\right)^2 + ... + c_0\left(\langle U_0|C_0|U_0\rangle - C_0^{val}\right)^2, \quad (6)$$

where the $w_i$, $o_1$ and $c_i$ represent the weights of the expected values, overlaps and constrains, respectively. $U_i$ represents different quantum circuits. In its straightforward VQE implementation (Listing 17) the cost function includes only the expectation value of the Hamiltonian:

$$L = \langle 0|U(\theta)^\dagger HU(\theta)|0\rangle. \quad (7)$$

However, one may want to add some constrains for better performance i.e. to lead to optimization into correct spin symmetry sector (Listing 18). This may be useful with hardware-efficient ansatzes:

$$L = \langle 0|U(\theta)^\dagger HU(\theta)|0\rangle + c_0\left(\langle 0|U(\theta)^\dagger S^2U(\theta)|0\rangle - S^2_{val}\right)^2. \quad (8)$$

More complex cases may include different circuits, but with the same ansatz for different parameter values. Variational Quantum Deflation (VQD) [23] algorithm for sequential computation of $k$-th excited states can be an example of the such case with the following cost function (Listing 19):

$$L = \langle 0|U(\theta_k^{exc})HU(\theta_k^{exc})|0\rangle + \ o_1\langle 0|U(\theta^{gs})U(\theta_k^{exc})|0\rangle + ...o_{k-1}\langle 0|U(\theta_{k-1}^{exc})U(\theta_k^{exc})|0\rangle. \quad (9)$$

Another case of different circuirs may be Subspace Search Variational Quantum Eigensolver (SS-VQE) algorithm [22] for simultaneous search of multiple excited states:

$$L = \langle \phi_0|U(\theta)HU(\theta)|\phi_0\rangle + ... + \langle \phi_k|U(\theta)HU(\theta)|\phi_k\rangle. \quad (10)$$

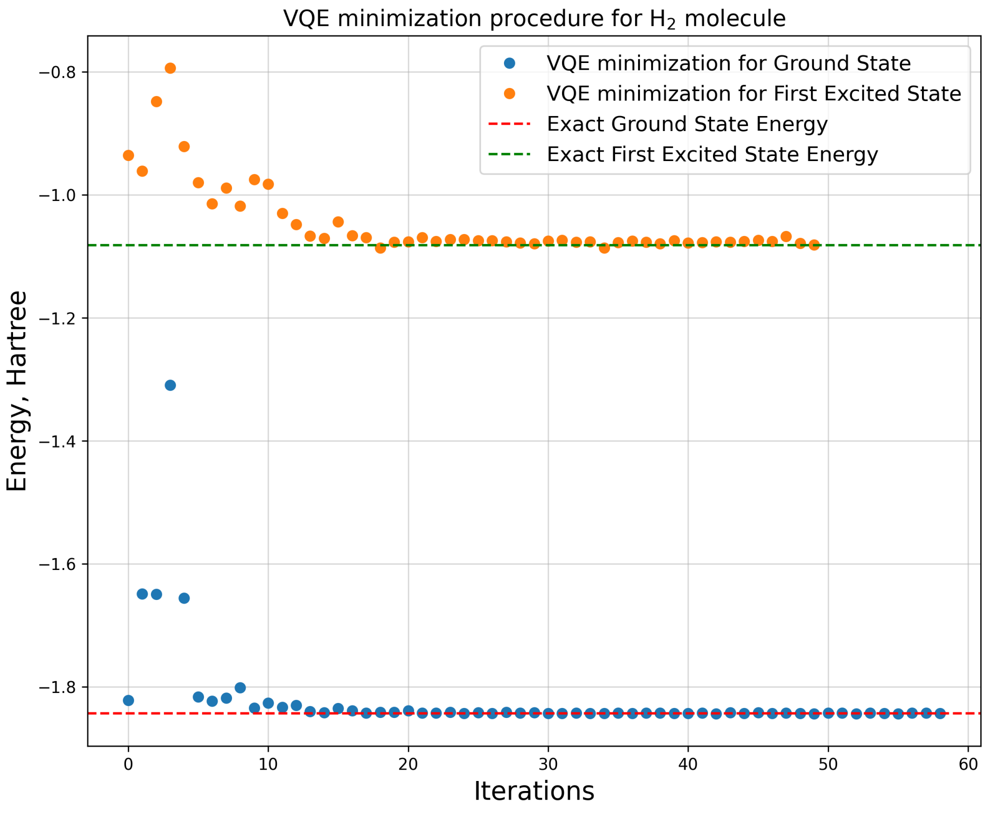

**Figure 5.** Using the cost function (9) to compute the ground and first excited state of $H_2$ molecule.

Here we provide the examples, which runs VQA with the cost function examples given above. It the first two we use a `HardwareEfficientAnsatz` class for this purpose:

```python
from src.models import Molecule
from omegaconf import OmegaConf
from src.ansatzes import HardwareEfficientAnsatz

config_path = "src/configs/molecules/h2_rhf_sto3g_eq.yaml"
config = OmegaConf.load(config_path)
config._set_item_impl("path", config_path)
mol = Molecule(config)
freeze_inds = config.molecule_info.frozen_orbs[-1]

fermionic_operator = FermionicOperator.from_integrals(integrals).
  operator_to_normal_order()
hamiltonian = QubitOperator.from_fermionic_operator(fermionic_operator,
  mapping = "jordan-wigner")
num_qubits = hamiltonian.num_qubits
n_layers = 3

hwe_ansatz = HardwareEfficientAnsatz(num_qubits=num_qubits,
num_layers=n_layers,
connectivity="chess")
```

```

objective_hwe = [(hwe_ansatz, ((hamiltonian, ), (1.0, )))]

```

**Listing 17.** Using `HardwareEfficientAnsatz` class to form a straightforward objective function (7)

```
particle_number_qop = QubitOperator.from_fermionic_operator(
  particle_number_fop(num_qubits))
spin_square_qop = QubitOperator.from_fermionic_operator(spin_square_fop(
  num_qubits))
opt_circuit_hwe = hwe_ansatz.build_circuit(vqe_result_hwe.x)
n_particles = 2
s2 = 0

objective_constr = [(hwe_ansatz, ((hamiltonian, particle_number_qop,
  spin_square_qop), lambda ham_val, pn_val, s2_val: ham_val + (pn_val -
  n_particles) ** 2 + (s2_val - s2) ** 2))]
```

**Listing 18.** Using `HardwareEfficientAnsatz` class to form a constrained objective function (8), incorporating particle number and $S^2$ value for the $H_2$ molecule.

```

init_state = hartree_fock_state(num_qubits, active_electrons=active_electrons)
uccsd_ansatz = UCCAnsatz(occ_orbs, virt_orbs, init_state=init_state)
objective_gs = [(uccsd_ansatz, ((hamiltonian,), (1,)))]
vqe_uccsd = VariationalAlgorithm(objective=objective_gs)
vqe_result_uccsd = vqe_uccsd.run()

gs_parameters = vqe_result_uccsd.x
gs_ansatz = ucc_ansatz.build_circuit(gs_parameters) # building a fixed ground
  state circuit with the optimized parameters

beta = [0.9, ]
objective_vqd = [(ucc_ansatz, ((hamiltonian,), lambda ham_val, current_ansatz:
   ham_val + beta[0] * PauliExpectation(state_ket=current_ansatz, state_bra=
  gs_ansatz).get_overlap()))]
vqe_uccsd_excited = VariationalAlgorithm(objective=objective_vqd, costfn="
  custom", overlap=True).run() # one have to pass a boolean flag to the class
  for computing the overlaps
```

**Listing 19.** Using `UCCAnsatz` class to construct a VQD objective function (9) for the first excited state of a molecule.

In our VQA implementation, terms from different operators for each quantum circuit are first collected and grouped, if necessary. Subsequently, unequal groups (terms) are measured. This optimizes the VQA procedure by eliminating the need to prepare identical circuits from different operators (i.e., most of the terms in $S^2$ are already in Hamiltonian operators). After completing all the previous steps (including selecting the active space, constructing the problem Hamiltonian, choosing the ansatz, selecting the appropriate cost function, and picking a classical optimizer), one can execute the variational quantum algorithm for a molecule:

```
from src.symmetry_utils import hartree_fock_state
import numpy as np
from src.models import Molecule
from omegaconf import OmegaConf
from src.algorithms import VariationalAlgorithm
from qiskit.providers.aer import AerSimulator

```

```
shots = 10 ** 4
backend = ("qiskit", AerSimulator(method="statevector"))
optimizer_config = {"optimizer": "COBYLA", "options": {"maxiter": 1000, "catol
  ": 1e-7, "tol": 1e-7}}

config_path = "src/configs/molecules/h2_rhf_sto3g_eq.yaml"
config = OmegaConf.load(config_path)
config._set_item_impl("path", config_path)
mol = Molecule(config)
frozen_inds = config.molecule_info.frozen_orbs[-1]
print("frozen_inds", frozen_inds)
integrals = mol.transform_mo2so(frozen_inds)
fermionic_operator = FermionicOperator.from_integrals(integrals).
  operator_to_normal_order()
hamiltonian = QubitOperator.from_fermionic_operator(fermionic_operator,
  mapping = "jordan-wigner")
n_qubits = hamiltonian.num_qubits
occ_orbs, virt_orbs = mol.occupied_virtual_orbs(frozen_inds)
active_electrons = mol.active_electrons(frozen_inds)

initial_state = hartree_fock_state(n_qubits, active_electrons)
uccsd = UCCAnsatz(excitation_order_list=(1, 2),
occ_orbs=occ_orbs, virt_orbs=virt_orbs,
init_state=initial_state)
parameters = np.random.uniform(-1, 1, uccsd.num_parameters)
uccsd_circuit = uccsd.build_circuit(parameters)

objective = [(uccsd, ((hamiltonian,), (1.0,)))]

vqe_uccsd = VariationalAlgorithm(objective=objective,
shots=shots,
backend=backend,
config=optimizer_config)

vqe_result_uccsd = vqe_uccsd.run()

nuc_repulsion_energy = mol.hf.energy_nuc()
exact_energy = mol.fci(freeze_inds).e_tot - mol.hf.energy_nuc()

print("VQE RESULT", vqe_result_uccsd)
print("FCI RESULT", exact_energy)
print("ENERGY DELTA", vqe_result_uccsd.fun - exact_energy)

OUTPUT:
VQE RESULT
fun: -1.841287970788897
maxcv: 0.0
message: 'Optimization terminated successfully.'
nfev: 67
status: 1
success: True
x: array([ 0.06824215, -0.02625553, -0.23748941])
FCI RESULT -1.8426866819057306
ENERGY DELTA 0.0013987111168336508
```

**Listing 20.** `VariationalAlgorithm` workflow demonstration.

### *3.5. Adaptive Algorithm*

The VQE algorithm requires a good initial guess of the ansatz to give accurate results, which is not always straightforward to obtain. To overcome this issue, the Adaptive Derivative-Assembled Pseudo-Trotter ansatz Variational Quantum Eigensolver (ADAPT-VQE) [34] extends the VQE algorithm by using an adaptive procedure. It iteratively constructs the ansatz by selecting and adding operators from a predefined pool, prioritizing those that most significantly contribute to energy reduction at each step.

The main steps of the Adaptive VQE algorithm are as follows:

(i) Initialization: Start with an initial state $|\psi(\theta)\rangle$ and a predefined pool of excitation operators $O_i$.

(ii) Outer Loop (Operator Selection): For each operator in the pool, compute the gradient of the expectation value $\langle\psi(\theta)|O_i H|\psi(\theta)\rangle$ with respect to the parameters $\theta$. Select the operator $O_{\text{max}}$ with the largest gradient.

(iii) Ansatz Update: Append the selected operator $O_{\text{max}}$ to the current ansatz to create a new ansatz $|\psi'(\theta')\rangle$.

(iv) Inner Loop (Parameter Optimization): For the new ansatz, calculate the expectation value $\langle\psi'(\theta')|H|\psi'(\theta')\rangle$. Use a classical optimizer to find the parameters $\theta'$ that minimize this expectation value.

(v) Convergence Check: Check if the energy has converged (i.e., the energy difference between iterations is below a certain threshold). If not, return to step 2 using the updated ansatz $|\psi'(\theta')\rangle$ and optimized parameters $\theta'$.

The following is an example of using the `AdaptiveAlgorithm` class for calculating the ground state energy of a LiH molecule:

```python
from src.algorithms import AdaptiveAlgorithm
from omegaconf import OmegaConf
from src.models.Molecule import Molecule

config_lih_path = "src/configs/molecules/lih_rhf_sto3g_eq.yaml"
config_lih_rhf = OmegaConf.load(config_lih_path)
config_lih_rhf._set_item_impl("path", config_lih_path)

mol = Molecule(config_lih_rhf)
conf = config_lih_rhf

freeze_inds = conf.molecule_info.frozen_orbs[0]


qubit_op = mol.build_hamiltonian(freeze_inds)
occ_orbs, virt_orbs = mol.occupied_virtual_orbs(freeze_inds=freeze_inds)

active_electrons = mol.active_electrons(freeze_inds)
tap_op = qubit_op.taper(active_electrons)

res = AdaptiveAlgorithm((mol, freeze_inds), tap_op, iterations=10,
  add_multiple_ops=True, grad_ratio=2, max_batch=2, gradient_method="num_diff"
  , tapering_info=qubit_op.tapering_info).run()

print("Energy", res)

nuc_repulsion_energy = mol.hf.energy_nuc()
exact_energy = mol.fci(freeze_inds).e_tot - mol.hf.energy_nuc()
print("FCI RESULT WITH NUCLEAR ENERGY", mol.fci(freeze_inds).e_tot)
print("FCI RESULT", exact_energy)
print("ENERGY DELTA", res - exact_energy)

OUTPUT:
Ansatz operators length:  15
Energy -8.83794699957919
FCI RESULT WITH NUCLEAR ENERGY -7.88217599080109
FCI RESULT -8.877556035167508
ENERGY DELTA 0.03960903558831852
```

**Listing 21.** Example of using `AdaptiveAlgorithm` class.

The `AdaptiveAlgorithm` class includes a parameter named *operator_pool*, which can accept a `QubitOperator` instance. This feature enables users to provide their own pool of qubit operators, from which the ansatz is constructed. By default, the operator pool is generated internally by the `AdaptiveAlgorithm` class.

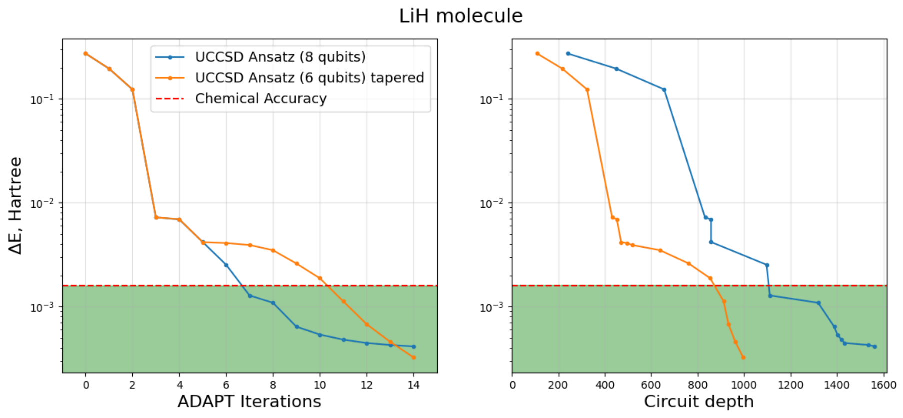

**Figure 6.** An example of applying our library's ADAPT-VQE algorithm for ground state computation of LiH molecule. The shaded green region indicates the area within the chemical accuracy of 0.0016 Hartree.

The algorithm adaptively constructs an efficient ansatz that can often yield more accurate results than standard VQE. Importantly, it does not require a good initial guess for the ansatz. Furthermore, this ADAPT-VQE implementation allows for the addition of multiple operators simultaneously. Studies have shown that algorithm performance improves significantly under these conditions [35]. However, this process can be computationally intensive due to the additional outer loop for operator selection and gradient calculation. The operator pool's selection can significantly impact the Adaptive VQE's performance and accuracy.

## 4. Summary

We have presented the comprehensive description of the abilities of the developed library for quantum chemistry simulations and have provided detailed instructions for its usage. As we have discussed, the developed library enables users to perform quantum computing-based chemistry calculations for small molecules. The library incorporates sophisticated algorithms such as the VQE and its adaptive version, ADAPT VQE, facilitating efficient simulations of quantum chemistry problems using various backends. This manual covers basic usage, advanced features, and offers practical examples, assisting users in leveraging the library to its full potential.

The article overviews the essential classes needed to construct a comprehensive variational quantum algorithm workflow. However, this discussion does not cover all methods and functionalities available within these classes. Certain research directions in this field have been identified and discussed.

## Acknowledgments
We thank Prof. A. Oganov for fruitful discussions and useful comments. This research has been supported by the Russian Quantum Center (QBoard). MDS and AKF also thank the support of the RSF grant (19-71-10092).

## References
[1] Elfving V E, Broer B W, Webber M, Gavartin J, Halls M D, Lorton K P and Bochevarov A 2020 How will quantum computers provide an industrially relevant computational advantage in quantum chemistry? (*Preprint* 2009.12472)
[2] Bauer B, Bravyi S, Motta M and Chan G K L 2020 *Chemical Reviews* **120** 12685–12717 pMID: 33090772
[3] Whitfield J D, Love P J and Aspuru-Guzik A 2013 *Phys. Chem. Chem. Phys.* **15**(2) 397–411 URL http://dx.doi.org/10.1039/C2CP42695A
[4] van Mourik T, Bhl M and Gaigeot M P 2014 *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences* **372** 20120488
[5] Jones L O, Mosquera M A, Schatz G C and Ratner M A 2020 *Journal of the American Chemical Society* **142** 3281–3295
[6] Bryenton K R, Adeleke A A, Dale S G and Johnson E R 2023 *WIREs Computational Molecular Science* **13** e1631
[7] Lee S, Lee J, Zhai H, Tong Y, Dalzell A, Kumar A, Helms P, Gray J, Cui Z H, Liu W, Kastoryano M, Babbush R, Preskill J, Reichman D, Campbell E, Valeev E, Lin L and Chan G 2023 *Nature Communications* **14**
[8] Preskill J 2018 *Royal Society* **2**
[9] Cerezo M, Arrasmith A, Babbush R, Benjamin S, Endo S, Fujii K, Mcclean J, Mitarai K, Yuan X, Cincio L and Coles P 2021 *Nature Reviews Physics* **3** 1–20
[10] Peruzzo A, Mcclean J, Shadbolt P, Yung M H, Zhou X, Love P, Aspuru-Guzik A and O'Brien J 2013 *Nature communications* **5**
[11] Born M and Oppenheimer R 1927 *Annalen der Physik* **389** 457–484
[12] Baer M 2006 *Beyond Born-Oppenheimer: Conical intersections and electronic nonadiabatic coupling terms* (Wiley-Interscience) ISBN 978-0-471-78007-6
[13] McArdle S, Endo S, Aspuru-Guzik A, Benjamin S C and Yuan X 2020 *Rev. Mod. Phys.* **92**(1) 015003 URL https://link.aps.org/doi/10.1103/RevModPhys.92.015003
[14] Cao Y, Romero J, Olson J, Degroote M, Johnson P, Kieferova M, Kivlichan I, Menke T, Peropadre B, Sawaya N, Sim S, Veis L and Aspuru-Guzik A 2019 *Chemical Reviews* **119**
[15] Yarkoni D R 2012 *Chemical Reviews* **112** 481–498
[16] Qiskit contributors 2023 Qiskit: An open-source framework for quantum computing
[17] Bergholm V Izaac J 2018 *arXiv*
[18] Tilly J, Chen H, Cao S, Picozzi D, Setia K, Li Y, Grant E, Wossnig L, Rungger I, Booth G H and Tennyson J 2022 *Physics Reports* **986** 1–128 ISSN 0370-1573 the Variational Quantum Eigensolver: a review of methods and best practices URL https://www.sciencedirect.com/science/article/pii/S0370157322003118
[19] Seeley J T, Richard M J and Love P J 2012 *The Journal of Chemical Physics* **137** 224109 ISSN 0021-9606 (*Preprint* https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/1.4768229/13999577/224109_1_online.pdf) URL https://doi.org/10.1063/1.4768229
[20] Anand A, Schleich P, Alperin-Lea S, Jensen P W K, Sim S, Díaz-Tinoco M, Kottmann J S, Degroote M, Izmaylov A F and Aspuru-Guzik A 2022 *Chem. Soc. Rev.* **51**(5) 1659–1684 URL http://dx.doi.org/10.1039/D1CS00932J
[21] Ryabinkin I, Genin S and Izmaylov A 2018 *Journal of Chemical Theory and Computation* **15**
[22] Nakanishi K M, Mitarai K and Fujii K 2019 *Phys. Rev. Res.* **1**(3) 033062 URL https://link.aps.org/doi/10.1103/PhysRevResearch.1.033062
[23] Higgott O, Wang D and Brierley S 2019 *Quantum* **3** 156
[24] Sun Q, Berkelbach T, Blunt N, Booth G, Guo S, Li Z, Liu J, McClain J, Sayfutyarova E, Sharma S, Wouters S and Chan G 2017 *Wiley Interdisciplinary Reviews: Computational Molecular Science* **8**
[25] Sun Q 2014 *Journal of computational chemistry* **36**
[26] Attila Szabo N S O 1996 *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory* (Dover Publications) ISBN 978-0486691862

[27] Jordan P and Wigner E 1928 *Z. Phys* **47** 631
[28] S B Bravyi A K 2002 *Annals of Physics* **298** 210–226
[29] Nielsen M A and Chuang I L 2011 *Quantum Computation and Quantum Information: 10th Anniversary Edition* 10th ed (USA: Cambridge University Press) ISBN 1107002176
[30] Gokhale P, Angiuli O, Ding Y, Gui K, Tomesh T, Suchara M, Martonosi M and Chong F T 2020 *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)* pp 379–390
[31] Bravyi S, Gambetta J, Mezzacapo A and Temme K 2017 Tapering off qubits to simulate fermionic hamiltonians preprint on webpage at https://arxiv.org/abs/1701.08213
[32] Lee J, Huggins W, Head-Gordon M and Whaley K 2018 *Journal of Chemical Theory and Computation* **15**
[33] Cao C, Hu J, Zhang W, Xu X, Chen D, Yu F, Li J, Hu H, Lv D and Yung M H 2021 Towards a larger molecular simulation on the quantum computer: Up to 28 qubits systems accelerated by point group symmetry preprint on webpage at https://arxiv.org/abs/2109.02110
[34] Grimsley H, Economou S, Barnes E and Mayhall N 2019 *Nature Communications* **10** 3007
[35] Sapova M and Fedorov A 2022 *Communications Physics* **5** 199