Electronic Supplementary Material (ESI) for Physical Chemistry Chemical Physics.

Supplementary Material: Molecular Excited State Calculations with Adaptive Wavefunctions on a Quantum Eigensolver Emulation

Hans Hon Sang Chan*
Department of Materials, University of Oxford, Parks Road, Oxford OX1 3PH, United Kingdom

Nathan Fitzpatrick
Cambridge Quantum Computing Ltd., 9a Bridge Street, Cambridge CB2 1UB, United Kingdom

Javier Segarra-Martí
Instituto de Ciencia Molecular, Universitat de Valencia, PO Box 22085 Valencia, Spain

Michael J. Bearpark
Department of Chemistry, Molecular Sciences Research Hub, Imperial College London, White City Campus, 82 Wood Lane, London W12 0BZ , United Kingdom

David P. Tew
Physical and Theoretical Chemical Laboratory, University of Oxford, South Parks Road, Oxford OX1 3QZ, United Kingdom
(Dated: November 11, 2021)

The is a technical document detailing the functionalities of an open-sourced package, the Quantum Eigensolver Building on Achievments of Both quantum computing and quantum chemistry (QEBAB), which we developed for investigating adaptive ansatz generation in excited state calculations in the Variational Quantum Eigensolver (VQE) framework. The code interfaces a number of other open-sourced packages for quantum chemistry and quantum computing; PySCF[1] and Libcint [2] for extracting the required one- and two-electron integrals and initialising the molecule, OpenFermion[3] for generation and transformation of UCC excitation operators, Pytket[4] for construction and compilation of ansatz circuits, and any backend supported by Pytket for circuit simulation. Additionally, in the associated work Scipy[5] was employed for variational minimisation of ansatz expectation energies, as well as the computation of operator gradients for adaptive ansatz growth. This document serves as a how-to guide to its usage (with annotated pseudo-code), and also as a step-by-step introduction to the VQE.

Typical workflow as follows: the user first initialises a molecule. The package is used to generate a set of Unitary Coupled Cluster (UCC)-type excitation operators, and construct an ansatz wavefunction out of the operators according to the users choice. The ansatz wavefunction is then transformed into a parameterised quantum circuit, which can be simulated on a number of quantum circuit simulator backends (in this work Qulacs [6] with GPU acceleration was used). A classical optimiser is used to adjust the parameters in the quantum circuit until the simulated energy expectation value converges.

* hans.chan@materials.ox.ac.uk

I. MOLECULE INITIALISATION

Molecules are initialised as `MolecularData` objects from OpenFermion:

```
from openfermion import MolecularData

# Initialise LiH with bond length 1.2 A
molecule = MolecularData(geometry=[('Li',
(0., 0., 0.)), ('H', (0., 0., 1.2))],
                basis='sto-3g',
                multiplicity=1)
```

With the OpenFermion PySCF wrapper, 1- and 2-electron integrals are calculated, which are stored in the `MolecularData` object. In this work, Restricted Hartree-Fock (RHF) Self-Consistent Field (SCF) calculations were used. It is then possible to generate the second quantised molecular Hamiltonian from `MolecularData`:

$$\hat{H} = h_{\mathrm{Nu}} + \sum_{p,q}^{N} h_{pq} a_p^\dagger a_q + \frac{1}{2} \sum_{p,q,r,s}^{N} h_{pqrs} a_p^\dagger a_q^\dagger a_r a_s \quad (1)$$

This needs to be modified into operations on a quantum computer. Using the Jordan-Wigner transform, the creation and annihilation operators ($a_j^\dagger$ and a_j) in the second quantised Hamiltonian from can be expressed in terms of Pauli matrices $\sigma_i \in \{\sigma^x, \sigma^y, \sigma^z\}$, which conveniently translate directly to 1-qubit Pauli logic gates

$P_i \in \{X, Y, Z\}$:

$$a_j^\dagger = \bigotimes_i^{j-1} \sigma_i^z \bigotimes \frac{1}{2}(\sigma_j^x - \mathrm{i}\sigma_j^y) \tag{2}$$

$$= \frac{1}{2}(X - \mathrm{i}Y) \otimes [Z_{j-1} \otimes \cdots \otimes Z_0] \tag{3}$$

$$a_j = \bigotimes_i^{j-1} \sigma_i^z \bigotimes \frac{1}{2}(\sigma_j^x + \mathrm{i}\sigma_j^y) \tag{4}$$

$$= \frac{1}{2}(X + \mathrm{i}Y) \otimes [Z_{j-1} \otimes \cdots \otimes Z_0] \tag{5}$$

The second quantised qubit Hamiltonian is thus a linear combination of Pauli terms $\bigotimes_i P_i$ (each term is a tensor multiplication). The following is an example of the terms in the series:

$$\hat{H} = h_0 I + h_1 Z_0 + \cdots + h_7 X_0 \otimes Z_1 \otimes X_2 + \ldots + h_{13} Y_0 \otimes Z_1 \otimes Y_2 \otimes Z_3 + \ldots \tag{6}$$

$$= \sum_j h_j \bigotimes_i P_i^j \tag{7}$$

where h_j are the 1 and 2 electron integrals and h_0 the nuclear Hamiltonian contribution from the SCF calculation.

We use the `jordan_wigner` transform function native to OpenFermion to map the fermionic operators in the Hamiltonian into unitary Pauli matrices that can be applied as quantum logic gates onto qubits. We chose to store the qubit Hamiltonian as Pytket `QubitPauliOperator` objects.

```
from openfermionpyscf import run_pyscf
from openfermion.transforms import
jordan_wigner
from pytket import QubitPauliOperator

# Calculate the 1,2 electron integrals
molecule = run_pyscf(molecule, run_scf=1)

# Hamiltonian
ham_qubit = jordan_wigner(mol.
get_molecular_hamiltonian())
ham_qubit.compress() # Now in QubitOperator
form
ham = QubitPauliOperator.from_OpenFermion(
ham_qubit)
```

II. GENERATING EXCITATION OPERATORS

We created custom `OperatorPool` classes which initialises different groups of UCC fermionic excitation operators from a given input number of electrons and number of orbitals. In the reported work, only the singlet-restricted, generalised excitations in the `sUCCGSD_Pool` and `sUpCCGSD_Pool` classes were used. Figure I further elaborates on which excitations are included The excitations are constructed using OpenFermion. Both inherit from the `OperatorPool` class, which has a number of built-in excitation operator functions described in the following sections.

```
from qebab.operators import sUpCCGSD_Pool

pool = sUpCCGSD_Pool()
pool.init(n_orb=molecule.n_orbitals,
          n_occ=molecule.
get_n_alpha_electrons(),
          n_vir=molecule.n_orbitals -
molecule.get_n_alpha_electrons())
```

A. Pauli Gadgets

The `OperatorPool` class transforms the fermionic excitation operators into strings of Pauli operators which maps onto quantum circuit components. Consider the parameterised UCC state preparation operator:

$$U(\vec{\theta}) = \prod_m e^{\theta_m(\tau_m - \tau_m^\dagger)} \tag{8}$$

where m indexes all possible single and double excitations, $\theta_m \in \{\theta_i^a, \theta_{ij}^{ab}\}$ and $\tau_m \in \{a_a^\dagger a_i, a_a^\dagger a_b^\dagger a_i a_j\}$. Using the Jordan-Wigner transform, each of the excitation terms $\theta_m(\tau_m - \tau_m^\dagger)$ are translated to Pauli matrices:

$$\theta_i^a(a_i^\dagger a_a - a_a^\dagger a_i) = \frac{\mathrm{i}\theta_i^a}{2} \bigotimes_{k=i+1}^{a-1} \sigma_k^z (\sigma_i^y \sigma_a^x - \sigma_i^x \sigma_a^y) \tag{9}$$

$$\begin{aligned}\theta_{ij}^{ab}(a_i^\dagger a_j^\dagger a_a a_b - a_a^\dagger a_b^\dagger a_i a_j) = &\frac{\mathrm{i}\theta_{ij}^{ab}}{8} \bigotimes_{k=i+1}^{j-1} \sigma_k^z \bigotimes_{l=a+1}^{b-1} \sigma_l^z \\ &(\sigma_i^x \sigma_j^x \sigma_a^y \sigma_b^x + \sigma_i^y \sigma_j^x \sigma_a^y \sigma_b^y \\ &+ \sigma_i^x \sigma_j^y \sigma_a^y \sigma_b^y + \sigma_i^x \sigma_j^x \sigma_a^x \sigma_b^y \\ &- \sigma_i^y \sigma_j^x \sigma_a^x \sigma_b^x - \sigma_i^x \sigma_j^y \sigma_a^x \sigma_b^x \\ &- \sigma_i^y \sigma_j^y \sigma_a^y \sigma_b^x - \sigma_i^y \sigma_j^y \sigma_a^x \sigma_b^y)\end{aligned} \tag{10}$$

where each product is a tensor multiplication. An exponentiated excitation term $e^{\theta_m(\tau_m - \tau_m^\dagger)}$ forms a string of circuit blocks (*Pauli gadget*), each with a single qubit rotation gate parameterised to θ_m of the excitation term[7].

Alternatively, the `OperatorPool` class can also express excitation operators into unitary matrices that can be used for direct matrix evaluation of the expected circuit behaviour.

B. Analytic Gradients

The `OperatorPool` class also computes the expected energy gradient of an ansatz (with respect to the free parameter θ_i) for each candidate excitation operator $\hat{A}_i$ in

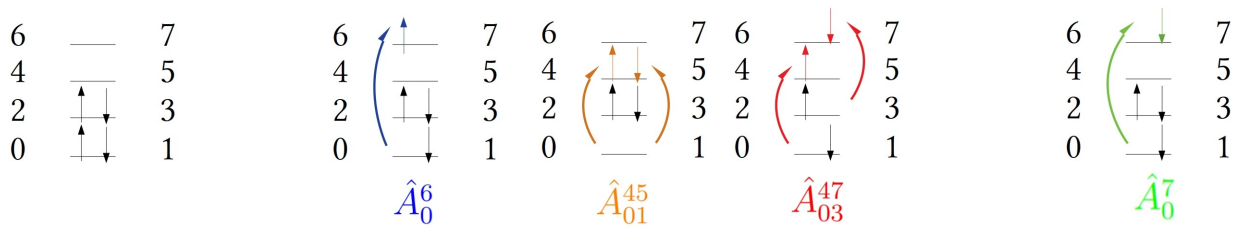
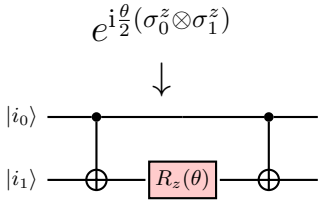
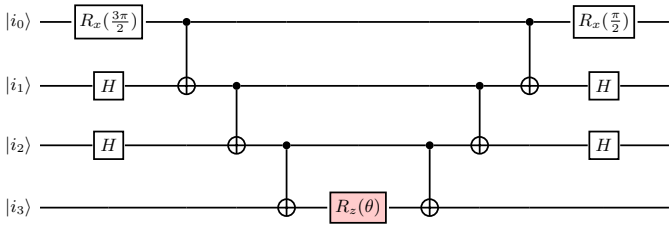


FIG. I: Single and double excitations. Three diagrams in the middle express spin-preserving, physical excitations. Diagram in the far right does not preserve the spin, and is not generated.



(A) Creating the unitary gate $e^{i\frac{\theta}{2}(\sigma_0^z \otimes \sigma_1^z)}$. CNOT gates are used to first entangle two qubits, then the rotation gate R_z is applied, followed by a second CNOT gate.



(B) A Pauli gadget for $e^{i\frac{\theta}{2}(\sigma_0^y \otimes \sigma_1^x \otimes \sigma_2^x \otimes \sigma_3^z)}$ in a 4 qubit circuit. The R_x gates rotate the phase of a qubit; in the 0^{th} qubit it rotates the phase of the qubit to the y-basis. The Hadamard gates H generate a superposition of the $|0\rangle$ and the $|1\rangle$ state; here it is used to access the x-basis of the qubit.

FIG. II: Examples of Pauli gadgets. Note that the each excitation term is formed of multiple Pauli gadgets, all parameterised to the same θ.

the pool, a key aspect in the ADAPT methods. Consider first the derivative of a single qubit rotation gate. We can represent the rotation matrices for the three cartesian axes as exponentials :

$$R(\theta_i)_X = e^{-i\theta_i X/2} \tag{11}$$
$$R(\theta_i)_Y = e^{-i\theta_i Y/2} \tag{12}$$
$$R(\theta_i)_Z = e^{-i\theta_i Z/2} \tag{13}$$

Therefore for each differential with respect to the rotation angles there is only a single term in the sum:

$$\frac{\partial}{\partial\theta_i} R(\theta_i)_X = -\frac{i}{2} \cdot X e^{-i\theta_i X/2} \tag{14}$$

$$\frac{\partial}{\partial\theta_i} R(\theta_i)_Y = -\frac{i}{2} \cdot Y e^{-i\theta_i Y/2} \tag{15}$$

$$\frac{\partial}{\partial\theta_i} R(\theta_i)_Z = -\frac{i}{2} \cdot Z e^{-i\theta_i Z/2} \tag{16}$$

Now consider an arbitrary unitary ansatz of the following form:

$$\left|\Psi(\vec{\theta})\right\rangle = U(\vec{\theta}) \left|\psi_{\text{ref}}\right\rangle \tag{17}$$

where the unitary is composed of exponentiated Pauli terms:

$$U(\vec{\theta}) = \prod_{i=1}^{N} e^{\theta_i \hat{A}_i} \tag{18}$$

The energy expectation is:

$$E(\vec{\theta}) = \left\langle \Psi(\vec{\theta}) \right| \hat{H} \left| \Psi(\vec{\theta}) \right\rangle \tag{19}$$

The energy gradient with respect to the i^{th} parameter θ_i in the ansatz is thus:

$$\frac{\partial E}{\partial\theta_i} = \langle\psi_{\text{ref}}| U^\dagger(\vec{\theta}) \hat{H} \frac{\partial U(\vec{\theta})}{\partial\theta_i} |\psi_{\text{ref}}\rangle + \langle\psi_{\text{ref}}| \frac{\partial U^\dagger(\vec{\theta})}{\partial\theta_i} \hat{H} U(\vec{\theta}) |\psi_{\text{ref}}\rangle \tag{20}$$

where:

$$\frac{\partial U(\vec{\theta})}{\partial\theta_i} = \prod_{j=i+1}^{N} (e^{\theta_j \hat{A}_j}) \hat{A}_i \prod_{k=1}^{i} (e^{\theta_i \hat{A}_i}) \tag{21}$$

Substituting in:

$$\frac{\partial E}{\partial \theta_i} = \left\langle \Psi(\vec{\theta}) \right| \hat{H} \prod_{j=i+1}^{N} (e^{\theta_j \hat{A}_j}) \hat{A}_i \prod_{k=1}^{i} (e^{\theta_i \hat{A}_i}) \left| \psi_{\text{ref}} \right\rangle - \langle \psi_{\text{ref}} | \prod_{k=i}^{1} (e^{-\theta_k \hat{A}_k}) \hat{A}_i \prod_{j=N}^{i+1} (e^{-\theta_j \hat{A}_j}) \hat{H} \left| \Psi(\vec{\theta}) \right\rangle \tag{22}$$

If we are only concerned with the last m^{th} operator in the ansatz, this simplifies to:

$$\frac{\partial E}{\partial \theta_m} = \left\langle \Psi(\vec{\theta}) \right| [\hat{H}, \hat{A_m}] \left| \Psi(\vec{\theta}) \right\rangle \tag{23}$$

which is equivalent to:

$$\frac{\partial E}{\partial \theta_m} = 2\mathcal{R} \left\langle \Psi(\vec{\theta}) \right| \hat{H} \hat{A_m} \left| \Psi(\vec{\theta}) \right\rangle \tag{24}$$

This is measured with the Hadamard test in Figure III. However our circuit is composed of Pauli gadgets rather than single qubit parameterized rotations such as in the UCC case. Following the same methodology using ancilla qubits and the Hadamard test, the circuit primitive corresponding to the derivative of a Pauli gadget with respect to its rotational parameter is given instead by Figure IV.

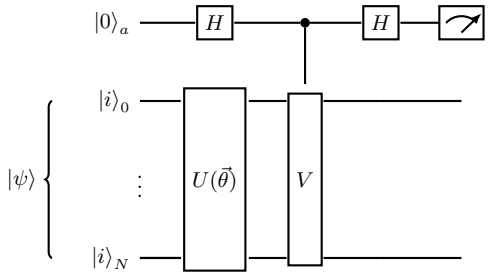


FIG. III: The Hadamard Test circuit for measuring the real part of the expected value when the unitary V is applied to $\left|\Psi(\vec{\theta})\right\rangle$ i.e. $\text{Re}\left\langle \Psi(\vec{\theta}) \right| V \left| \Psi(\vec{\theta}) \right\rangle$.

For adaptively growing the ansatz for excited states, further gradients of the overlap with other eigenstates need to be measured:

$$\begin{aligned}
\frac{\partial E}{\partial \theta_m} =& 2\mathcal{R} \langle \Psi(\vec{\theta}) | \hat{A}_m \hat{H}_k | \Psi(\vec{\theta}) \rangle & (25)\\
=& 2\mathcal{R} \langle \Psi(\vec{\theta}) | \hat{A}_m \left(\hat{H} + \sum_{i=0}^{j-1} \beta_i |\Phi_i\rangle \langle \Phi_i| \right) |\Psi(\vec{\theta})\rangle \\
=& 2\mathcal{R} \langle \Psi(\vec{\theta}) | \hat{A}_m \hat{H} + \hat{A}_m \sum_{i=0}^{j-1} \beta_i |\Phi_i\rangle \langle \Phi_i | \Psi(\vec{\theta}) \rangle \\
=& 2\mathcal{R} \langle \Psi(\vec{\theta}) | \hat{A}_m \hat{H} |\Psi(\vec{\theta})\rangle + \underbrace{2\mathcal{R} \sum_{i=0}^{j-1} \beta_i \langle \Psi(\vec{\theta}) | \hat{A}_m |\Phi_i\rangle \langle \Phi_i | \Psi(\vec{\theta}) \rangle}_{\text{overlap gradient}} & (26)
\end{aligned}$$

Details of circuits for overlap measurement to follow.

Inspired by the original creators of the ADAPT method[8], these gradient computations are computed by evaluating the corresponding unitary matricies instead of simulating circuits which would provide analytic gradients of operators. The code does not implement calculation of the gradient in circuit form.

III. REFERENCE CIRCUITS AND ANSATZ GENERATION

We created custom `Ansatz` constructor classes which take a pool of excitation operators in `OperatorPool` and a reference circuit as input, then creates a symbolically parameterised state preparation ansatz circuit of choice. A number of different references have been defined, and in this work the closed-shell singlet HF reference $|\psi_{\text{HF}}\rangle$ and open-shell lowest energy triplet reference $|\psi_{T_1}\rangle$ were used (see main text). We use the Pytket circuit generator to build the ansatz circuits from sequences of Pauli

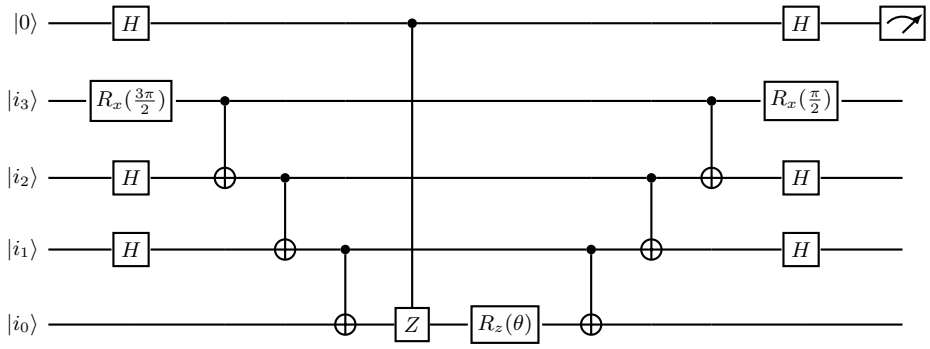


FIG. IV: Circuit Primitive for $\mathcal{R}(\frac{\delta}{\delta\theta_i}e^{i\frac{\theta}{2}(\sigma_0^z\otimes\sigma_1^x\otimes\sigma_2^x\otimes\sigma_3^y)})$

operators, and also Pytket compilation passes to reduce the quantum gate count in the circuit. Figure V is an example of a constructed circuit. In this work, the `k_UCC_Ansatz` and the `ADAPT_VQD_Ansatz` constructors were used.

The `k_UCC_Ansatz` can be used to repeat a set of excitation operators k times, as prescribed in the k-UpCCGSD ansatz. As it is a fixed ansatz for every eigenstate and geometry, it only needs to be called once in the beginning of a calculation.

```
class k_UCC_Ansatz(Ansatz):
    def generate_Circuit(self, ref: str, k:
int):
        ...
        # Reference Circuit
        ref_circ = reference_circuit_lib[ref
]
        ...
        # Ansatz building, k-depth
        self.symbols = {}
        for rep in range(1, k+1):
            qubit_pauliop = {}
            for i in range(n_params):
                # Generate fresh symbol
                theta = fresh_symbol('t{}'.
format(i))
                self.symbols[theta] = None
                # Isolate operator
                op = self.pool.
qubit_paulistrs[i]
                for qpstr, coeff in op.items
():
                    if coeff.imag > 0:
                        qubit_pauliop[qpstr]
 = theta
                    else:
                        qubit_pauliop[qpstr]
 = -1.0 * theta

            Pauli_U = QubitPauliOperator(
qubit_pauliop)

            if rep==1:
                sym_circ =
gen_term_sequence_circuit(Pauli_U,

         ref_circ,

         partition_strat=PauliPartitionStrat
.CommutingSets,

         colour_method=GraphColourMethod.
Lazy)
            else:
                k_circ = Circuit(n_qubits)
                k_circ =
gen_term_sequence_circuit(Pauli_U,

       k_circ,

       partition_strat=PauliPartitionStrat.
CommutingSets,

       colour_method=GraphColourMethod.Lazy)

                sym_circ.append(k_circ)
        ...
        # Compilation pass
        self.smart_circ = sym_circ.copy()
        Transform.UCCSynthesis(
PauliSynthStrat.Sets, CXConfigType.Tree).
apply(self.smart_circ)

        return self.smart_circ, self.symbols
```

The `ADAPT_VQD_Ansatz` is of course adaptive and needs to be called at each new geometry. The constructor will iteratively grow an ansatz until the convergence criterion is met, and so requires the convergence threshold ϵ as input. It also needs to compute the energy and overlap gradient. For the latter, it calls the analytic gradient functions described above from the `Operator_Pool` classes.

```
class ADAPT_VQD_Ansatz(Ansatz):
    def generate_Circuit(self,
                         ref: str,
                         params: list, #
currently sought after state
                         eigen_ansatze: list,
 # list of circuits
                         beta: float,
                         ham_sparse,
                         threshold):
```

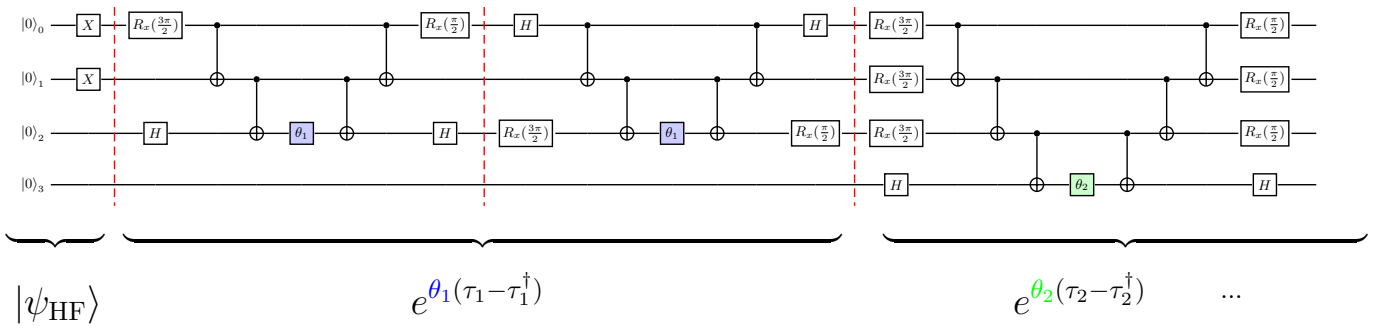
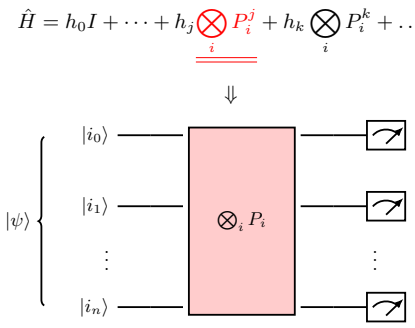


FIG. V: Section of a UCC-type state preparation circuit $\left|\Psi(\vec{\theta})\right\rangle = \ldots e^{\theta_2(\tau_2-\tau_2^\dagger)} e^{\theta_1(\tau_1-\tau_1^\dagger)} |\psi_{\text{HF}}\rangle$.



Outcome	Parity	Eigenvalue
...000	even	+1
...001	odd	-1
...101	even	+1
...111	odd	-1
⋮	⋮	⋮

$$\langle\psi| \bigotimes_i P_i^j |\psi\rangle = \frac{\sum \text{Measured Eigenvalues}}{\text{Number of Measurements}}$$

FIG. VI: (LEFT) Circuit for measuring one Pauli term. Upon measuring, each qubit collapses to either 0 or 1. The is repeated multiple times and the results (shots) are recorded. (RIGHT) A shot table mapping each measurement outcome to an eigenvalue (parity is whether the number of 1's in the measurement is odd or even, which corresponds to an eigenvalue of -1 and +1 respectively). The energy expectation of the Pauli term P_i is thus the average eigenvalue over multiple shots multiplied by h_i.

```
        if len(params)==0: # new eigenstate
            # reset for new eigenstate
            self.smart_circ = None
            self.symbols = {}
            self.f_op = []

            # reset Reference Circuit
            ref_circ = reference_circuit_lib
[ref]

            qubit_pauliop = {}
            Pauli_U = QubitPauliOperator(
qubit_pauliop)
            self.grad_circ =
gen_term_sequence_circuit(Pauli_U,

            self.ref_circ,

            partition_strat=
PauliPartitionStrat.CommutingSets,

            colour_method=GraphColourMethod.
Lazy)
            self.smart_circ = self.grad_circ
.copy()

        else: # repopulate current
eigenstate
            self.grad_circ = self.smart_circ
.copy()
            self.symbols = dict(zip(self.
symbols, params))
            self.grad_circ.
symbol_substitution(self.symbols)


        # Calculating gradients for
operators in pool
        curr_norm = 0
        next_deriv = 0
        for op_index in range(self.pool.
n_ops):
            ...
            # Energy Gradient
            gi = self.pool.
compute_gradient_i(op_index,

ham_sparse,

self.grad_circ,

backend)
            # Overlap Gradient
            overlap_list = []
            for eigen_circ in eigen_ansatze:
                # 2 Re beta * <ansatz|A(k)|
```

```
eigen><eigen|ansatz>
                overlap = sqrt(gen_overlap(
self.grad_circ, eigen_circ, backend))
                ov_g = abs(self.pool.
compute_ov_grad_i(op_index, self.grad_circ,
eigen_circ, backend))
                overlap_list.append(abs(np.
real(2 * beta * ov_g * overlap)))
            overlap_sum = sum(overlap_list)

            ...
            # Add up total gradient of
operator
            gi = abs(gi) + overlap_sum

            curr_norm += gi**2
            if abs(gi) > next_deriv:
                next_deriv = abs(gi)
                next_index = op_index

        curr_norm = np.sqrt(curr_norm)
        max_of_com = next_deriv

        # Convergence or growth
        if curr_norm < threshold:
            self.converged = True
        else:
            qubit_pauliop = {}
            op_circ = Circuit(self.pool.
n_spin_orb)

            # Generate fresh symbol
            theta = fresh_symbol('t')
            self.symbols[theta] = None

            # Append fermion operator
            self.f_op.append(self.pool.
fermi_ops[next_index])
            self.op_indices.append(
next_index)

            # Isolate operator
            op = self.pool.qubit_paulistrs[
next_index]
            for qpstr, coeff in op.items():
                if coeff.imag > 0:
                    qubit_pauliop[qpstr] =
theta
                else:
                    qubit_pauliop[qpstr] =
-1.0 * theta

            Pauli_U = QubitPauliOperator(
qubit_pauliop)
            op_circ =
gen_term_sequence_circuit(Pauli_U,

    op_circ,

    partition_strat=PauliPartitionStrat.
CommutingSets,

    colour_method=GraphColourMethod.Lazy)
            self.smart_circ.append(op_circ)
            ...
            Transform.UCCSynthesis(
PauliSynthStrat.Sets, CXConfigType.Tree).
apply(self.smart_circ)

        return self.converged, self.
    smart_circ, self.symbols, self.final_map
```

IV. ANSATZ OPTIMISATION AND ENERGY CALCULATION

The total energy expectation is the sum of energy expectation of each Pauli terms in the qubit Hamiltonian:

$$\langle E \rangle = \left\langle \Psi(\vec{\theta}) \right| \hat{H} \left| \Psi(\vec{\theta}) \right\rangle \tag{27}$$

$$= \sum_j h_j \left\langle \Psi(\vec{\theta}) \right| \bigotimes_i P_i^j \left| \Psi(\vec{\theta}) \right\rangle \tag{28}$$

Figure VI elaborates on the statistical nature of these measurements. In excited state VQD calculations, the overlap is calculated using the vacuum test:

$$\text{since} \quad |\Phi_i\rangle = V_i \, |0 \ldots 00\rangle \quad \text{and} \quad \left|\Psi(\vec{\theta})\right\rangle = U \, |0 \ldots 00\rangle$$

where V_i and U are unitary reference and state preparation operators, then the overlap between the two states is the expectation of measuring $|0 \ldots 00\rangle$ when the $V_i^\dagger U$ circuit is applied.

$$\left\langle \Phi_i \middle| \Psi(\vec{\theta}) \right\rangle = \langle 0 \ldots 00| \, V_i^\dagger U \, |0 \ldots 00\rangle$$

This technique doubles the depth of the circuit (refer to Figure VII for the circuit diagram).

```
def get_zero_state_probability(circ):
    """Measures qubits in 0 basis
    """
    statevector = ProjectQBackend().
get_state(circ)

    return abs(statevector[0])**2


def overlap_gen(psi_circ, phi_circ):
    """Overlap measurement: vacuum test
    """
    circ = psi_circ.copy()
    phi_circ = phi_circ.dagger()
    circ.append(phi_circ)

    Transform.OptimisePhaseGadgets().apply(
circ)
    prob_X = get_zero_state_probability(circ
=circ)

    return prob_X
```

Combining the above components, an objective function which calculates expectation values (with orthogonal penalisation included for excited states) given an input of free parameters `thetas` is needed. Any backend supported by Pytket can be used to simulate this measurement; in this investigation expectation calculations were performed on the noiseless quantum simulator ProjectQ.

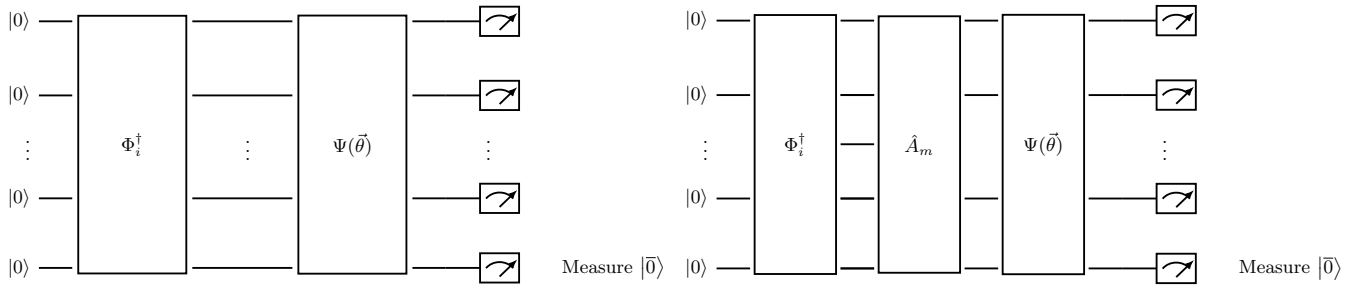


FIG. VII: (LEFT) the vacuum test, which doubles the circuit depth. (RIGHT) Measurement of the overlap gradient.

```
def gen_objective(operator_pool, qubit_ham,
optimised_ansatze, beta):
    def energy_objective(thetas):
        """(pseudo-code) example of energy
objective function
            which also accounts for the
overlap
        """
        circ = gen_ansatz_circ(thetas,
operator_pool)
        # E = <psi|H|psi>
        energy = ProjectQBackend().
get_expectation_value(circ,qubit_ham)

        overlap_sums = []
        if len(optimised_ansatze)!=0:
            for phi_i in optimised_ansatze:
                # b * <phi_i|psi>
                overlap_i = beta *
overlap_gen(circ, phi_i)
                overlap_sums.append(
overlap_i)

        overlap_sums = sum(overlap_sums)
        energy = energy + overlap_sums

        return energy
    return energy_objective
```

This objective function was used for the nonlinear classical optimisation of the wavefunction ansatz and energy eigenvalues with respect to the free parameters. This was achieved with the iterative `optimize.minimize` function from Scipy using the Limited Broyden-Fletcher-Goldfarb-Shannon "Bound" (L-BFGS-B) method. The L-BFGS-B used norm of the projected energy gradient smaller than 10^{-5} as convergence criterion, with a maximum number of iteration set at 30. The initial input values for the free parameters were chosen to be random numbers distributed between 0 and 0.1 throughout.

V. SPIN EXPECTATION CALCULATION

In this investigation the $\hat{S}^2$ expectation values of optimised ansätze were computed to verify spin-restrictions were observed. The $\hat{S}^2$ operator is:

$$\hat{S}^2 = \hat{S}_+\hat{S}_- + \hat{S}_z(\hat{S}_z - 1) \tag{29}$$

where:

$$\hat{S}_+ = \sum_p a^\dagger_{p_\alpha} a_{p_\beta} \tag{30}$$

$$\hat{S}_- = \sum_q a^\dagger_{q_\beta} a_{q_\alpha} \tag{31}$$

$$\hat{S}_z = \frac{1}{2}\sum_p (a^\dagger_{p_\alpha} a_{p_\alpha} - a^\dagger_{p_\beta} a_{p_\beta}) \tag{32}$$

The spin operator in second quantisation is thus:

$$\begin{aligned}\hat{S}^2 &= \sum_{p,q} a^\dagger_{p_\alpha} a_{p_\beta} a^\dagger_{p_\beta} a_{p_\alpha} + \frac{1}{2}\sum_p (a^\dagger_{p_\alpha} a_{p_\alpha} - a^\dagger_{p_\beta} a_{p_\beta}) \left[\frac{1}{2}\sum_q (a^\dagger_{p_\alpha} a_{p_\alpha} - a^\dagger_{p_\beta} a_{p_\beta}) - 1\right] \\ &= \sum_{p,q}\left[a^\dagger_{p_\alpha} a_{p_\beta} a^\dagger_{p_\beta} a_{p_\alpha} + \frac{1}{4}\left(a^\dagger_{p_\alpha} a_{p_\alpha} a^\dagger_{q_\alpha} a_{q_\alpha} - a^\dagger_{p_\alpha} a_{p_\alpha} a^\dagger_{q_\beta} a_{q_\beta} - a^\dagger_{p_\beta} a_{p_\beta} a^\dagger_{q_\alpha} a_{q_\alpha} + a^\dagger_{p_\beta} a_{p_\beta} a^\dagger_{q_\beta} a_{q_\beta}\right)\right] \\ &\quad -\frac{1}{2}\sum_p (a^\dagger_{p_\alpha} a_{p_\alpha} - a^\dagger_{p_\beta} a_{p_\beta})\end{aligned} \tag{33}$$

The above operator is defined as a `FermionOperator` object for a molecule with a given number of orbitals.

Once a converged ansatz is obtained, its $\hat{S}^2$ expectation value is obtained using the same procedure as that of energy expectation calculation, substituting the Hamiltonian `FermionOperator` with the $\hat{S}^2$ `FermionOperator`.

[1] Q. Sun, T. C. Berkelbach, N. S. Blunt, G. H. Booth, S. Guo, Z. Li, J. Liu, J. D. McClain, E. R. Sayfutyarova, S. Sharma, S. Wouters and G. K. L. Chan, *WIREs Computational Molecular Science*, 2018, **8**,.

[2] Q. Sun, *Journal of Computational Chemistry*, 2015, **36**, 1664–1671.

[3] J. R. McClean, F. M. Faulstich, Q. Zhu, B. O'Gorman, Y. Qiu, S. R. White, R. Babbush and L. Lin, *New Journal of Physics*, 2020, **22**, 093015.

[4] I. O. Sokolov, P. K. Barkoutsos, P. J. Ollitrault, D. Greenberg, J. Rice, M. Pistoia and I. Tavernelli, *The Journal of Chemical Physics*, 2020, **152**, 124107.

[5] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. Ä. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, Ä. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, A. Vijaykumar, A. P. Bardelli, A. Rothberg, A. Hilboll, A. Kloeckner, A. Scopatz, A. Lee, A. Rokem, C. N. Woods, C. Fulton, C. Masson, C. Häggström, C. Fitzgerald, D. A. Nicholson, D. R. Hagen, D. V. Pasechnik, E. Olivetti, E. Martin, E. Wieser, F. Silva, F. Lenders, F. Wilhelm, G. Young, G. A. Price, G. L. Ingold, G. E. Allen, G. R. Lee, H. Audren, I. Probst, J. P. Dietrich, J. Silterra, J. T. Webber, J. Slavič, J. Nothman, J. Buchner, J. Kulick, J. L. Schönberger, J. V. de Miranda Cardoso, J. Reimer, J. Harrington, J. L. C. Rodríguez, J. Nunez-Iglesias, J. Kuczynski, K. Tritz, M. Thoma, M. Newville, M. Kümmerer, M. Bolingbroke, M. Tartre, M. Pak, N. J. Smith, N. Nowaczyk, N. Shebanov, O. Pavlyk, P. A. Brodtkorb, P. Lee, R. T. McGibbon, R. Feldbauer, S. Lewis, S. Tygier, S. Sievert, S. Vigna, S. Peterson, S. More, T. Pudlik, T. Oshima, T. J. Pingel, T. P. Robitaille, T. Spura, T. R. Jones, T. Cera, T. Leslie, T. Zito, T. Krauss, U. Upadhyay, Y. O. Halchenko and Y. Vázquez-Baeza, *Nature Methods*, 2020, **17**, 261–272.

[6] Y. Suzuki, Y. Kawase, Y. Masumura, Y. Hiraga, M. Nakadai, J. Chen, K. M. Nakanishi, K. Mitarai, R. Imai, S. Tamiya, T. Yamamoto, T. Yan, T. Kawakubo, Y. O. Nakagawa, Y. Ibe, Y. Zhang, H. Yamashita, H. Yoshimura, A. Hayashi and K. Fujii, *Qulacs: a fast and versatile quantum circuit simulator for research purpose*, 2020.

[7] A. Cowtan, S. Dilkes, R. Duncan, W. Simmons and S. Sivarajah, *Electronic Proceedings in Theoretical Computer Science*, 2020, **318**, 214–229.

[8] H. R. Grimsley, S. E. Economou, E. Barnes and N. J. Mayhall, *Nature Communications*, 2019, **10**, 1–11.