QEOM refactoring

Anthony Gandon

I. PRINCIPLES OF QUANTUM SUBSPACE EXPANSION

The most general definition of quantum subspace expansion methods introduces a reference state $|\psi_{ref}\rangle$ and a linear set of expansion operators $F = \{F_\alpha, \dots\}$ which together give the extended subspace $\{F_\alpha |\psi_{ref}\rangle, \dots\}$.

QSE formalism One can represent any operator in this linear subspace. In particular, one can build the Hamiltonian $H_{el}|_F$ and of the identity $S|_F$ in this subspace from quantum measurements on the reference state

$$(H_{el}|_F)_{\alpha\beta} = \langle\psi_{ref}| F_\alpha^\dagger H_{el} F_\beta |\psi_{ref}\rangle \tag{1}$$

$$(S|_F)_{\alpha\beta} = \langle\psi_{ref}| F_\alpha^\dagger F_\beta |\psi_{ref}\rangle, \tag{2}$$

and then classically diagonalize the pseudo-eigenvalue problem

$$H_{el}|_F X = S|_F X E|_F \tag{3}$$

to obtain the states $|\psi_{k,QSE}\rangle = \sum_\alpha X_\alpha^k F_\alpha |\psi_{ref}\rangle$ with energies

$$(E|_F)_k = \frac{\langle\psi_{k,QSE}| H_{el} |\psi_{k,QSE}\rangle}{\langle\psi_{k,QSE}|\psi_{k,QSE}\rangle} = \frac{X^{k\dagger} H_{el}|_F X^k}{X^{k\dagger} S|_F X^k}, \tag{4}$$

such that the stationary conditions with respect to the coefficient X are satisfied

$$\frac{\partial (E|_F)_k}{\partial X^k} = 0. \tag{5}$$

QEOM formalism An alternative to the QSE formulation was proposed in the form of the quantum Equation of Motion (qEOM). It is based on the observation that when the operators in the linear subspace are expressed with commutators instead of products

$$(H_{el}|_F)_{\alpha\beta} = \langle\psi_{ref}| [F_\alpha^\dagger, H_{el}, F_\beta] |\psi_{ref}\rangle \tag{6}$$

$$(S|_F)_{\alpha\beta} = \langle\psi_{ref}| [F_\alpha^\dagger, F_\beta] |\psi_{ref}\rangle, \tag{7}$$

then the solutions of the pseudo-eigenvalue problem eq. (3) satisfying the 'killer-condition' yield the excitation energy differences $(E|_F)_{0k} = (E|_F)_k - (E|_F)_0$ instead of the absolute excitation energies. The 'killer-condition' is defined as a vacuum condition for the excitation operator $O_k = \sum_\beta X_\beta^k F_\beta$

$$O_k^\dagger |\psi_{ref}\rangle = 0 \tag{8}$$

II. SYMMETRIES IN QEOM

When the system of interest exhibits symmetries, many of the matrix elements in eqs. (1), (2), (6) and (7) can be set to 0 without requiring any numerical calculation.

This comes from the observation that the expectation value on the ground state of an operator A_{aux} that anti-commutes with at least one symmetry of the system will be analytically 0.

Take σ a symmetry of the Hamiltonian such that $\{A_{aux}, \sigma\} = 0$. The ground state of the Hamiltonian lies in one unique symmetry sector corresponding to an eigenvalue that we write as s ($s = \pm 1$).

$$\langle\psi_0| A_{aux}\sigma |\psi_0\rangle = -\langle\psi_0| \sigma A_{aux} |\psi_0\rangle \tag{9}$$

$$\implies s\langle\psi_0| A_{aux} |\psi_0\rangle = -s\langle\psi_0| A_{aux} |\psi_0\rangle \tag{10}$$

$$\implies 2s\langle\psi_0| A_{aux} |\psi_0\rangle = 0 \tag{11}$$

$$\implies \langle\psi_0| A_{aux} |\psi_0\rangle = 0 \tag{12}$$

For a double product of the form $F_\alpha^\dagger A_{aux} F_\beta$, (the same applies for a double commutator of the form $[F_\alpha^\dagger, H_{el}, F_\beta]$

- If $[A_{aux}, \sigma] = 0$, then $\langle\psi_0| F_\alpha^\dagger A_{aux} F_\beta |\psi_0\rangle \neq 0$ if and only if
  - $[F_\alpha, \sigma] = 0$ and $[F_\alpha, \sigma] = 0$
  - $\{F_\alpha, \sigma\} = 0$ and $\{F_\alpha, \sigma\} = 0$
- If $\{A_{aux}, \sigma\} = 0$, then $\langle\psi_0| F_\alpha^\dagger A_{aux} F_\beta |\psi_0\rangle \neq 0$ if and only if
  - $[F_\alpha, \sigma] = 0$ and $\{F_\alpha, \sigma\} = 0$
  - $\{F_\alpha, \sigma\} = 0$ and $[F_\alpha, \sigma] = 0$

In practice, this means that an efficient computation of the QEOM matrix elements should run through the expansion operators in an order dictated by the symmetry sectors of the problem. One starts by enumerating the 2^{n_s} symmetry sectors for the n_s symmetries and then gathers all expansion operators in this sector.

III. PARALLELISM IN QUANTUM SUBSPACE EXPANSION

Question: At the moment, the parallelism is called in each sector

The classical overhead of the QSE codes comes from the computation of all operators of the form

$$[F_\alpha^\dagger, H_{el}, F_\beta] \text{ or } F_\alpha^\dagger H_{el} F_\beta. \tag{13}$$

Current implementation The parallelism is currently implemented at the level of the sector.

```
# Each expansion operator can either commute or anti-commute with each of the ns symmetries. This splits them into 2^ns sectors.
def _build_one_sector(
        available_hopping_ops,
        untapered_op,
        z2_symmetries
    ):
    to_be_computed_list = []
    # Define the list of inputs
    for idx, m_u in enumerate(mus):
        n_u = nus[idx]
        left_op_1 = available_hopping_ops.get(f"E_{m_u}")
        right_op_1 = available_hopping_ops.get(f"E_{n_u}")
        right_op_2 = available_hopping_ops.get(f"Edag_{n_u}")
        to_be_computed_list.append(
            (m_u, n_u, left_op_1, right_op_1, right_op_2)
        )

    # Builds the commutator for each of these entries
    results = parallel_map(
        self._build_commutator_routine,
        to_be_computed_list,
        task_args=(untapered_op, z2_symmetries),
        num_processes=algorithm_globals.num_processes,
    )

    # Post-process results
    for result in results:
        m_u, n_u, eom_operators = result
        for index_op, op in eom_operators.items():
            if op is not None:
                all_matrix_operators[f"{index_op}_{m_u}_{n_u}"]
                    = op

for targeted_tapering_values in combinations:
    # Find the expansion operators in the sector
    available_hopping_ops = {}
    targeted_sector = np.asarray(targeted_tapering_values) == 1
    for key, value in type_of_commutativities.items():
        value = np.asarray(value)
        if np.all(value == targeted_sector):
            available_hopping_ops[key] = pre_tap_hopping_ops[key]
    # untapered_qubit_op is a PauliSumOp and should not be exposed.
    _build_one_sector(available_hopping_ops, pre_tap_operator, z2_symmetries)
```

Possible change The alternative is to apply the parallelism at the global level. (First create a global list of operators to be computed and then use parallelism on the global list.)

```

to_be_computed_list = []

# Each expansion operator can either commute or anti-commute with each of the ns symmetries. This
    splits them into 2^ns sectors.
def _build_one_sector(
        available_hopping_ops,
        untapered_op,
        z2_symmetries
    ):
    # Fill the list of inputs
    for idx, m_u in enumerate(mus):
        n_u = nus[idx]
        left_op_1 = available_hopping_ops.get(f"E_{m_u}")
        right_op_1 = available_hopping_ops.get(f"E_{n_u}")
        right_op_2 = available_hopping_ops.get(f"Edag_{n_u}")
        to_be_computed_list.append(
            (m_u, n_u, left_op_1, right_op_1, right_op_2)
        )

for targeted_tapering_values in combinations:
    # Find the expansion operators in the sector
    available_hopping_ops = {}
    targeted_sector = np.asarray(targeted_tapering_values) == 1
    for key, value in type_of_commutativities.items():
        value = np.asarray(value)
        if np.all(value == targeted_sector):
            available_hopping_ops[key] = pre_tap_hopping_ops[key]
    # untapered_qubit_op is a PauliSumOp and should not be exposed.
    _build_one_sector(available_hopping_ops, pre_tap_operator, z2_symmetries)

# Builds the commutator for each of these entries
results = parallel_map(
    self._build_commutator_routine,
    to_be_computed_list,
    task_args=(untapered_op, z2_symmetries),
    num_processes=algorithm_globals.num_processes,
)

# Post-process results
for result in results:
    m_u, n_u, eom_operators = result
    for index_op, op in eom_operators.items():
        if op is not None:
            all_matrix_operators[f"{index_op}_{m_u}_{n_u}"]
                = op
```

IV. PREPARING OPERATORS FOR QEOM

The qubit converter has multiple methods for operator preparation:

- `convert()` should be applied to the Hamiltonian only. It maps the Second quantized hamiltonian into a PauliSumOp, applies two qubit reduction. It finds the symmetry sector of the groundstate and tapers the Hamiltonian accordingly, Lastly, it stores the symmetries and the number of particles
- `convert_only()` It maps the Second quantized operator into a PauliSumOp, and applies two qubit reduction.
- `force_match()` Sets user defined symmetries and number of particles
- `convert_match()` Uses the symmetries and number of particles set in (force match or convert) to further convert a list of operators.
- `_symmetry_reduce()` Tapers a list of PauliSumOp.

How do we want to prepare QEOM operators?

- All operators (expansion operators, auxiliary operators and hamiltonian) should at least be mapped to PauliSumOp and two qubit reduced.
- The first step of the tapering can be applied to all operators.
- The second part of the tapering is NOT multiplicative, so it should be applied on the result of the product (instead of taking the product of the tapered operators).
- The operators we are going to measure are of the form $H, A_{aux}, E_\alpha H E_\beta, E_\alpha A_{aux} E_\beta$. They should all be fully tapered.

The tapering of operators is a two step process:

- `convert_clifford` converts operators $H \to H' = UHU^\dagger$
- `taper_clifford` drops n_s (#symmetries) qubits from H'

In the GroundStateEigensolver class, the preparation of operators is defined in the `get_qubit_operator()` method.

```
def get_qubit_operators(
    self,
    problem: BaseProblem,
    aux_operators: dict[str, SecondQuantizedOp | FermionicOp | QubitOperator] | None = None,
) -> tuple[QubitOperator, dict[str, QubitOperator] | None]:
    """Gets the operator and auxiliary operators, and transforms the provided auxiliary operators.
    Note that contrary to the method :meth:‘get_qubit_oprators‘ from the
    :class:‘GroundStateEigensolver‘, this returns three outputs: the hamiltonian, second
    quantization
    default auxilaries, and user defined auxiliaries. Also note that this methods performs a
    specific
    treatment of the symmetries required by the qEOM calculation."""

    main_second_q_op, aux_second_q_ops = problem.second_q_ops()

    # 1. Convert to PauliSumOp and apply two qubit reduction
    # We apply the meth:‘convert()‘ with the symmetries deliberatly set to None
    main_operator = self.qubit_converter.convert_only(
        main_second_q_op,
        num_particles=problem.num_particles,
    )
    self.qubit_converter.force_match(num_particles=problem.num_particles)

    aux_default_ops = self.qubit_converter.convert_match(aux_second_q_ops)
    aux_custom_ops = {}
    if aux_operators is not None:
        for name_aux, aux_op in aux_operators.items():
            if isinstance(aux_op, (SecondQuantizedOp, FermionicOp)):
                converted_aux_op = self.qubit_converter.convert_match(aux_op, True)
            else:
                converted_aux_op = aux_op

            aux_custom_ops[name_aux] = converted_aux_op

    # 2. Find the z2symmetries, set them in the qubit_converter, and apply the first step of the
    # tapering.
    _, z2symmetries = self.qubit_converter._find_taper_op(
        main_operator, problem.symmetry_sector_locator
    )
    self.qubit_converter.force_match(z2symmetries=z2symmetries)

    pre_tap_main_operator = self.qubit_converter._convert_clifford(main_operator)
    pre_tap_default_aux_ops = self.qubit_converter._convert_clifford(aux_default_ops)
    pre_tap_custom_aux_ops = self.qubit_converter._convert_clifford(aux_custom_ops)

    # 3. If the eigensolver does not support auxiliary operators, reset them
    if not self.solver.supports_aux_operators():
        pre_tap_default_aux_ops = None
        pre_tap_custom_aux_ops = None
```

```
48
49      # 4. If a MinimumEigensolverFactory was provided, then an additional call to get_solver() is
50      # required.
51      if isinstance(self.solver, MinimumEigensolverFactory):
52          self._gsc._solver = self.solver.get_solver(problem, self.qubit_converter)
53
54      return pre_tap_main_operator, pre_tap_default_aux_ops, pre_tap_custom_aux_ops
```

V. EVALUATING AUX OPERATORS ON THE EXCITED STATES

NB: Parallelism can be applied to the prepare excited states observables
NB: Dealing with transition amplitudes and expectation values
NB: Shortcut keywords "diag", "full"
NB: QSE style evaluation VS single shot

```
def _evaluate_observables_excited_states(
    self,
    pre_tap_aux_observables: dict[str, PauliSumOp],
    expansion_basis_data: Tuple[dict[str, PauliSumOp], dict[str, List[bool]], int],
    reference_state: Tuple[QuantumCircuit, Sequence[float]],
    expansion_coefs_rescaled: np.ndarray,
) -> Tuple[dict[str, Tuple], dict[str, Tuple]]:
    """Evaluate the expectation values and transition amplitudes of the auxiliary operators on the
    excited states. Custom rules can be used to define which expectation values and transition
    amplitudes to compute. A typical rule is specified in the form of a dictionnary
    {'hamiltonian':[(1,1)]}

    Args:
        pre_tap_aux_observables: Dict of auxiliary operators for which properties will be computed.
        expansion_basis_data: Dict of transformed hopping operators, dict of commutativity types,
        size of the qEOM problem.
        reference_state: Reference ground state.
        expansion_coefs_rescaled: Expansion coefficient matrix X such that HX=SXE and X^\dag SX is
        the identity.

    Returns:
        List of excitation operators [Identity, O_1, O_2, ...]
    """

    aux_operators_eigenvalues = {}
    transition_amplitudes = {}

    _, _, size = expansion_basis_data

    if pre_tap_aux_observables is not None:

        # 1. Build excitation operators O_l such that O_l |0> = |l>
        excitations_ops_reduced = self._build_excitation_operators(
            expansion_basis_data, reference_state, expansion_coefs_rescaled
        )

        # 2. Prepare observables O_k^\dag @ Aux @ O_l
        op_aux_op_dict = self.prepare_excited_states_observables(
            pre_tap_aux_observables, excitations_ops_reduced, size
        )

        # 3. Measure observables
        tap_op_aux_op_dict = self.qubit_converter._symmetry_reduce_clifford(
            op_aux_op_dict, True
        )
        aux_measurements = estimate_observables(
            self._estimator, reference_state[0], tap_op_aux_op_dict, reference_state[1]
        )

        # 4. Format aux_operators_eigenvalues
        indices_diag = np.diag_indices(size + 1)
        indices_diag_as_list = [(i, j) for i, j in zip(indices_diag[0], indices_diag[1])]
```

```
        for indice in indices_diag_as_list:
            aux_operators_eigenvalues[indice] = {}
            for aux_name in pre_tap_aux_observables.keys():
                aux_operators_eigenvalues[indice][aux_name] = aux_measurements.get(
                    (aux_name, indice[0], indice[1]), (0.0, {}))

        # 5. Format transition_amplitudes
        indices_offdiag = np.triu_indices(size + 1, k=1)
        indices_offdiag_as_list = [
            (i, j) for i, j in zip(indices_offdiag[0], indices_offdiag[1])
        ]
        for indice in indices_offdiag_as_list:
            transition_amplitudes[indice] = {}
            for aux_name in pre_tap_aux_observables.keys():
                transition_amplitudes[indice][aux_name] = aux_measurements.get(
                    (aux_name, indice[0], indice[1]), (np.nan, {})
                )

    return aux_operators_eigenvalues, transition_amplitudes
```