Received: 25 May 2021 | Revised: 19 July 2021 | Accepted: 7 October 2021

IET Quantum Communication

DOI: 10.1049/qtc2.12024

The Institution of Engineering and Technology WILEY

ORIGINAL RESEARCH PAPER

QuaSiMo: A composable library to program hybrid workflows for quantum simulation

Thien Nguyen[1,2] | Lindsay Bassman Oftelie[3] | Phillip C. Lotshaw[2,4] | Dmitry Lyakh[2,5] | Alexander McCaskey[1,2] | Vicente Leyton-Ortega[2,4] | Raphael Pooser[2,4] | Wael Elwasif[2,4] | Travis S. Humble[1,2] | Wibe A. de Jong[3]

[1]Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA

[2]Quantum Computing Institute, Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA

[3]Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, California, USA

[4]Computer Science and Engineering Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA

[5]National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA

Correspondence

Thien Nguyen, Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, 37831, USA.
Email: nguyentm@ornl.gov

Funding information

U.S. Department of Energy, Grant/Award Number: ACCELERATED RESEARCH IN QUANTUM COMPUTING (ARQC)

Abstract

A composable design scheme is presented for the development of hybrid quantum/classical algorithms and workflows for applications of quantum simulation. The proposed object-oriented approach is based on constructing an expressive set of common data structures and methods that enables programming of a broad variety of complex hybrid quantum simulation applications. The abstract core of the scheme is distilled from the analysis of the current quantum simulation algorithms. Subsequently, it allows synthesis of new hybrid algorithms and workflows via the extension, specialisation, and dynamic customisation of the abstract core classes defined by the proposed design. The design scheme is implemented using the hardware-agnostic programming language QCOR into the QuaSiMo library. To validate the implementation, the authors test and show its utility on commercial quantum processors from IBM and Rigetti, running some prototypical quantum simulations.

KEYWORDS

quantum computing, quantum information

1 | INTRODUCTION

Quantum simulation is an important use case of quantum computing for scientific computing applications. While numerical calculations of quantum dynamics and structure are staples of modern scientific computing, quantum simulation represents the analogous computation based on the principles of quantum physics. Specific applications are wide-ranging and include calculations of electronic structure [1–4], scattering [5], dissociation [6], thermal rate constants [7], materials dynamics [8], and response functions [9].

Presently, this diversity of quantum simulation applications is being explored with quantum computing despite the limitations on the fidelity and capacity of quantum hardware [10–12].

These applications are tailored to such limitations by designing algorithms that can be tuned and optimised in the presence of noise or model representations that can be reduced in dimensionality. Examples include variational methods such as the variational quantum eigensolver (VQE) [13–16], quantum approximate optimisation algorithm (QAOA) [17], quantum imaginary time evolution (QITE) [18], and quantum machine learning (QML) among others.

The varied use of quantum simulation raises concerns for efficient and effective programming of these applications. The current diversity in quantum computing hardware and low-level, hardware-specific languages imposes a significant burden on the application user. For instance, IBM provides Aqua [19], which is part of the Qiskit framework, targeting high-level quantum applications such as chemistry and finance. The emphasis of Aqua is on providing robust implementations of quantum algorithms, yet the concept of reusable and extensible workflows, especially for quantum simulations, is not formally supported. The user usually has to implement custom workflows from lower-level constructs, such as circuits and operators, available in Qiskit Terra [20]. Similarly, Tequila [21] is another Python library that provides commonly used functionalities for quick prototyping of variational-type quantum algorithms. Orquestra [22], a commercially available solution from Zapata, on the other hand, orchestrates quantum application workflows as black-box phases, thus requires users to provide implementations for each phase.

The lack of a common workflow for applications of quantum simulation hinders broader progress in testing and evaluation of such hardware. A common, reusable and extensible programming workflow for quantum simulation would enable broader adoption of these applications and support more robust testing by the quantum computing community.

In this contribution, we address development of common workflows to unify applications of quantum simulation. Our approach constructs common data structures and methods to program varying quantum simulation applications, and we leverage the hardware-agnostic language QCOR and programming framework XACC to implement these ideas. We demonstrate these methods with example applications from material science and chemistry, and we discuss how to extend these workflows to the experimental validation of quantum computation advantage, in which numerical simulations can benchmark programs for small-sized models [12, 23–26].

Upon release of the QuaSiMo library, we became aware of the additional application modules available to the Qiskit framework, which share a number of features with our proposed composable workflow design. In particular, the Qiskit Nature module [27] introduces new concepts to model and solve quantum simulation problems, and reiterates the need for modular, domain-specific, and workflow-driven quantum application builders, which the QuaSiMo work addresses. The key differentiators for our work lie in the performance and extensibility of the implementation. Since QuaSiMo is developed on top of the C++ QCOR compiler infrastructure, it can take advantage of optimal performance for classical computing components of the workflow, such as circuit construction [28] or post-processing. Moreover, the plugin-based extension model of QuaSiMo is distinct from that of Qiskit Nature. Rather than requiring new extensions being imported, QuaSiMo puts forward a common interface for all of its extension points [29], and therefore enables the development of portable user applications with respect to the underlying library implementation.

2 | SOFTWARE ARCHITECTURE

Cloud-based access to quantum computing naturally differentiates programming into conventional and quantum tasks [30, 31]. The resulting hybrid execution model yields a loosely integrated computing system by which common methods have emerged for programming and data flow. We emphasise this concept of workflow to organise programming applications for quantum simulation.

Figure 1 shows the blueprint of our Quantum Simulation Modelling (QuaSiMo) library. The programming workflow is defined by a QuantumSimulationWorkflow concept, which encapsulates the hybrid quantum-classical procedures pertinent to a quantum simulation, for example, VQE, QAOA, or dynamical quantum simulation. A quantum simulation workflow exposes an execute method taking as input a QuantumSimulationModel object representing the quantum model that needs to be simulated. This model captures quantum mechanical observables, such as energy, spin magnetisation, etc., that we want the workflow to solve or simulate for. In addition, information about the system Hamiltonian, if different from the observable operator of interest, and customised initial quantum state preparation can also be specified in the QuantumSimulationModel.

By separating the quantum simulation model from the simulation workflow, our object-oriented design allows the concrete simulation workflow to simulate rather generic quantum models. This design leverages the ModelFactory utility, implementing the object-oriented factory method pattern. A broad variety of input mechanisms, such as those provided by the QCOR infrastructure or based on custom interoperability wrappers for quantum-chemistry software, can thus be covered by a single customisable polymorphic model. For additional flexibility, the last createModel factory method overload accepts a polymorphic builder interface ModelBuilder, the implementations of which can build arbitrarily composed QuantumSimulationModel objects.

QuantumSimulationWorkflow is the main extension point of our QuaSiMo library. Built upon the CppMicroServices framework conforming to the Open Services Gateway Initiative (OSGi) standard [29], QuaSiMo allows implementation of a new quantum workflow as a plugin loadable at runtime. At the time of this writing, we have developed the QuantumSimulationWorkflow plugins for the VQE, QAOA, QITE, and time-dependent simulation algorithms, as depicted in Figure 1. All these plugins are implemented in the QCOR language [28, 33] using the externally provided library routines.

At its core, a hybrid quantum-classical workflow is a procedural description of the quantum circuit composition, pre-processing, execution (on hardware or simulators), and

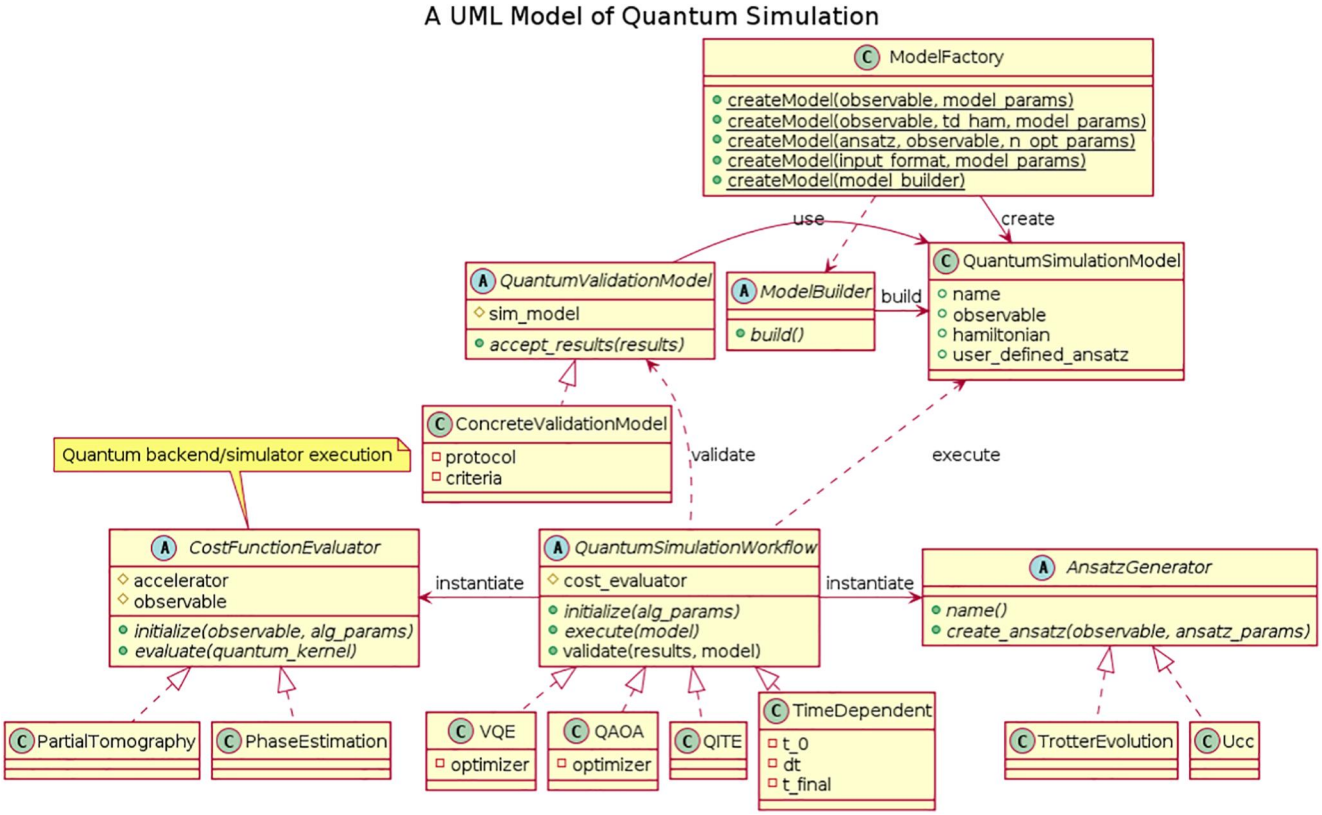

FIGURE 1 The class UML diagram of the quantum simulation application. The fully typed version is provided separately (see [32])

post-processing. To facilitate modularity and reusability in workflow development, we put forward two concepts, AnsatzGenerator and CostFunctionEvaluator. AnsatzGenerator is a helper utility used to generate quantum circuits based on a predefined method such as the Trotter decomposition [34, 35] or the unitary coupled-cluster (UCC) ansatz [36]. CostFunctionEvaluator automates the process of calculating the expectation value of an observable operator. For example, a common approach is to use the partial state tomography method of adding change-of-basis gates to compute the operator expectation value in the Z basis. Given the CostFunctionEvaluator interface, quantum workflow instances can abstract away the quantum backend execution and the corresponding post-processing of the results. This functional decomposition is particularly advantageous in the NISQ regime since one can easily integrate the noise-mitigation techniques, for example, the verified quantum phase estimation protocol [37], into the QuaSiMo library, which can then be used interchangeably by all existing workflows.

Finally, our abstract QuantumSimulationWorkflow class also exposes a public validate method accepting a variety of concrete implementations of the abstract QuantumValidationModel class via a polymorphic interface. Given the quantum simulation results produced by the execute method of QuantumSimulationWorkflow, the concrete implementations of QuantumValidationModel must implement its accept_results method based on different validation protocols and acceptance criteria. For example, the acceptance criteria can consist of distance measures of the results from previously validated values, or from the results of validated simulators. The measure may also be taken relative to experimentally obtained data, which, with sufficient error analysis to bound confidence in its accuracy, can serve as a ground truth for validation. A more concrete example in a NISQ workflow includes the use of the QuantumSimulationWorkflow class to instantiate a variational quantum eigensolver simulator, followed by the use of validate to instantiate a state vector simulator. Results from both simulators can be passed to the QuantumValidationModel accept_results method that evaluates a distance measure method and optionally calls a decision method that returns a binary answer. Other acceptance criteria include evaluation of formulae with input data, application of curve fits, and user-defined criteria provided in the concrete implementation of the abstract QuantumValidationModel class. The validation workflow relies on the modular architecture of our approach, which effectively means that writing custom validation methods and constructing user-defined validation workflows is achieved by extending the abstract QuantumValidationModel class.

In our opinion, the proposed object-oriented design is well suited to serve as a pattern for implementing diverse hybrid quantum-classical simulation algorithms and workflows, which can then be aggregated inside a library under a unified object-oriented interface. Importantly, our standardised polymorphic design with a clear separation of concerns and multiple extension points provides a high level of composability to developers interested in implementing rather complex quantum simulation

workflows. The modularity and composability of our workflow design scheme provide an effective mechanism to break complex quantum simulation procedures into reusable building blocks and also serve as a platform for iterative quantum algorithm development. In particular, we envision that a variety of new algorithms could leverage or be derived from existing algorithms using the workflow representation as we proposed here. By leveraging the workflow design formalism and the QuaSiMo library, algorithm developers benefit from pre-built and validated software components while focussing on novel implementations of workflow subroutines and compositions of workflow protocols.

3 | TESTING AND EVALUATION

As shown in Figure 1, QuaSiMo includes several built-in workflow implementations, such as time-dependent dynamical simulation, VQE, QAOA, and QITE. Here, we presented an extended evaluation of these workflows by testing the implementation against several of the original use cases to validate the correctness of the implementation and to evaluate performance considerations. It is worth noting that the correctness validation of quantum simulation results in the following is performed against known 'ground truth' values. Automatic result validation via QuantumValidationModel, proposed as a part of this QuaSiMo library (Figure 1), is for future consideration and is not included in this evaluation. Our implementation of the programming workflow for applications of quantum simulation is available online [38].

We also note that a preliminary evaluation of VQE and dynamical simulation workflows has been presented in Ref. [39] due to their representative characteristics. Specifically, the VQE algorithm demonstrates the variational simulation procedure comparable to that of the QAOA algorithm. Similarly, both QITE and generic time-dependent simulation procedures are classified broadly as dynamical time evolution algorithms.

3.1 | Dynamical simulation

As a first sample use case, we consider a non-equilibrium dynamics simulation of the Heisenberg model in the form of a quantum quench. A quench of a quantum system is generally carried out by initialising the system in the ground state of some initial Hamiltonian, H_i, and then evolving the system through time under a final Hamiltonian, H_f. Here, we demonstrate a simulation of a quantum quench of a one-dimensional (1D) antiferromagnetic (AF) Heisenberg model using the QCOR library to design and execute the quantum circuits.

Our AF Heisenberg Hamiltonian of interest is given by

$$H = J\sum_{\langle i,j\rangle}\{\sigma_i^x\sigma_j^x + \sigma_i^y\sigma_j^y + g\sigma_i^z\sigma_j^z\} \tag{1}$$

where $J > 0$ gives the strength of the exchange couplings between nearest neighbour spins pairs $\langle i, j\rangle$, $g > 0$ defines the anisotropy in the system, and σ_i^α is the α-th Pauli operator acting on qubit i. We choose our initial Hamiltonian to be the Hamiltonian in Equation (1) in the limit of $g \to \infty$. Thus, setting $J = 1$, $H_i = C\sum\sigma_i^z\sigma_{i+1}^z$, where C is an arbitrarily large constant. The ground state of H_i is the Néel state, given by $|\psi_0\rangle = |\uparrow\downarrow\uparrow\ldots\downarrow\rangle$, which is simple to prepare on the quantum computer. We choose our final Hamiltonian to have a finite, positive value of g, so $H_f = \sum_i\{\sigma_i^x\sigma_{i+1}^x + \sigma_i^y\sigma_{i+1}^y + g\sigma_i^z\sigma_{i+1}^z\}$. Our observable of interest is the staggered magnetisation [40], which is related to the AF order parameter and is defined as

$$m_s(t) = \frac{1}{N}\sum_i (-1)^i\langle\sigma_i^z(t)\rangle \tag{2}$$

where N is the number of spins in the system.

Figure 2 shows the sample results for $N = 9$ spins for three different values for g in H_f. The qualitatively different behaviours of the staggered magnetisation after the quench for $g < 1$ and $g > 1$ are apparent, and agree with previous studies [40]. We present a listing of the code expressing this implementation in Figure 3.

We develop QuaSiMo on top of the QCOR infrastructure, as shown in Figure 1; thus, any quantum simulation workflows constructed in QuaSiMo are retargetable to a broad range of quantum backends. The results that we have demonstrated in Figure 2 are from a simulator backend. The same code as shown in Figure 3 can also be recompiled with a −qpu flag to target a cloud-based quantum processor, such as those available in the IBMQ network.

Currently available quantum processors, known as noisy intermediate-scale quantum (NISQ) computers [41], have relatively high gate-error rates and small qubit decoherence times, which limit the depth of quantum circuits that can be executed with high fidelity. As a result, long-time dynamic simulations are challenging for NISQ devices as current algorithms produce quantum circuits that increase in depth with increasing numbers of time-steps [42]. To limit the circuit size, we simulated a small AF Heisenberg model, Equation (1), with only three spins on the IBM's Yorktown (ibmqx2) and Casablanca (ibmq_casablanca) devices.

The simulation results from real quantum hardware for g values of 0.0 and 4.0 are shown in Figure 4, where we can see the effects of gate-errors and qubit decoherence leading to a significant impairment of the measured staggered magnetisation (circles) compared to the theoretical values (solid lines). In particular, Figure 4 demonstrates how the quality of the quantum hardware can affect simulation performance. The Yorktown backend has considerably worse performance metrics than the Casablanca backend[1]. Specifically, compared to the Casablanca backend, Yorktown has a slightly higher two-qubit gate-error rate, nearly double the read-out error rate,

[1]Calibration data: IBMQ Casablanca: Avg. CNOT Error: 1.165×10^{-2}, Avg. Readout Error: 2.069×10^{-2}, Avg. T1: 85.68 μs, Avg. T2: 78.5 μs. IBMQ Yorktown: Avg. CNOT Error: 1.644×10^{-2}, Avg. Readout Error: 4.440×10^{-2}, Avg. T1: 50.95 μs, Avg. T2: 34.3 μs.

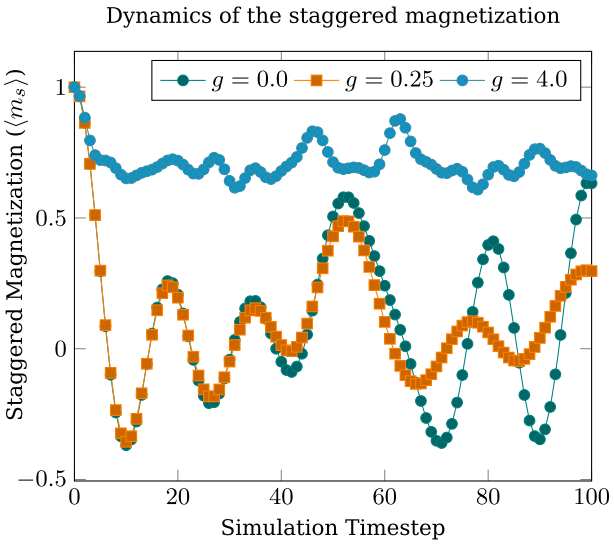

FIGURE 2 Simulation results of staggered magnetisation for a Heisenberg model with nine spins after a quantum quench. The Trotter step size (dt) is 0.05

```
using namespace QuaSiMo;
// AF Heisenberg model
auto problemModel = ModelFactory::createModel(
    "Heisenberg", {{"Jx", 1.0},
                   {"Jy", 1.0},
                   // Jz == g parameter
                   {"Jz", g},
                   // No external field
                   {"h_ext", 0.0},
                   {"num_spins", n_spins},
                   {"initial_spins",
                   initial_spins},
                   {"observable",
                   "staggered_magnetization"}});
// Time-dependent simulation workflow
auto workflow =
    std::make_shared<TimeDependentWorkflow>();
workflow->initialize({{"dt", dt},
                      {"steps", n_steps}});
// Execute the workflow
auto result = workflow->execute(problemModel);
// Get the observable values
// (staggered magnetization)
auto obsVals =
     result.get<std::vector<double>>("exp-vals");
```

FIGURE 3 Defining the AF Heisenberg problem model and simulating its dynamics with QuaSiMo. In this example, g is the anisotropy parameter, as shown in Equation (1), and n_spins is the number of spins/qubits. initial_spins is an array of 0 or 1 values denoting the initial spin state. initial_spins was initialised (not shown here) to a vector of alternating 0 and 1 values (Néel state). dt and n_steps are Trotter step size and number of steps, respectively

and substantially lower qubit decoherence times. While identical quantum circuits were run on the two machines, we see much better distinguishability between the results for the two values of g in the results from Casablanca than those from Yorktown.

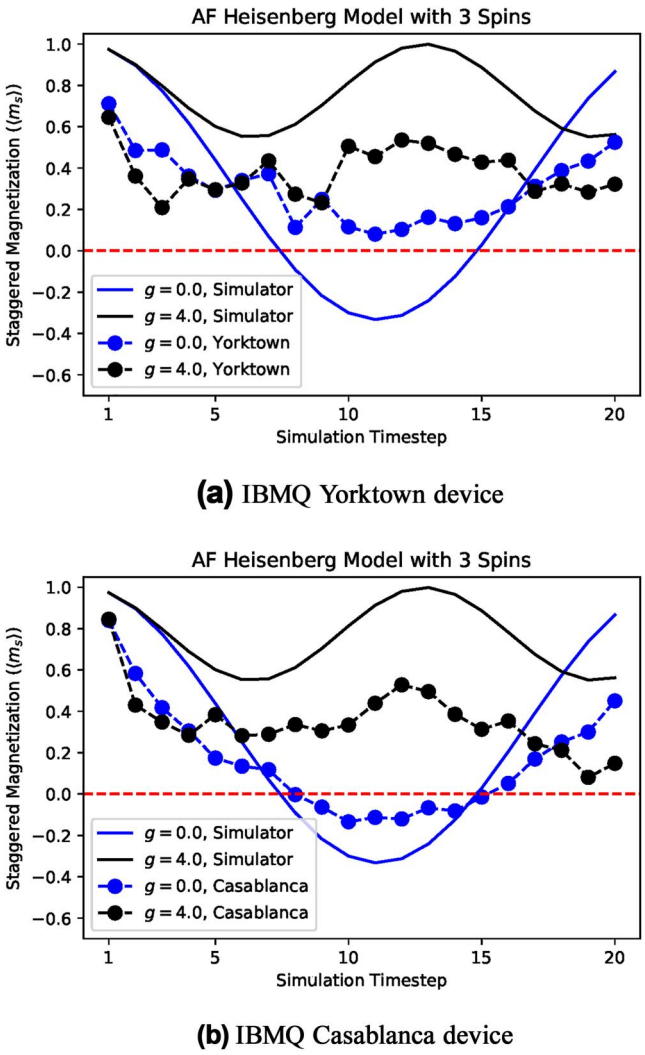

FIGURE 4 Results of simulation an AF model (Equation 1) for a system with three spins using the code snippet in Figure 3 targeting the IBMQ's Yorktown (a) and Casablanca (b) devices. Each data point is an average of five runs of 8192 measurement shots each. The circuits are compiled and optimised using the QCOR compiler before submitting for execution. The Trotter time-step (dt) is 0.05

The staggered magnetisation response to a quench for a simple three-qubit AF Heisenberg model in Figure 4, albeit noisy, illustrates non-trivial dynamics beyond that of decoherence (decaying to zero). Improvements in circuit construction (Trotter decomposition) and optimisation, noise mitigation, and, most importantly, hardware performance (gate fidelity and qubit coherence) are required to scale up this time-domain simulation workflow for large quantum systems.

3.2 | Variational Quantum Eigensolver

As a second use case demonstration, we apply the Variational Quantum Eigensolver (VQE) algorithm to find the ground state energy of H_2. The VQE is a quantum-classic hybrid algorithm used to find Hamiltonian's eigenvalues, where the

quantum process side is represented by a parametrised quantum circuit whose parameters are updated by a classical optimisation process [43]. The algorithm updates the quantum circuit parameters θ to minimise Hamiltonian's expectation value E_θ until it converges.

The performance of the VQE algorithm, as any other quantum-classical variational algorithms [44], depends in part on the selection of the classical optimiser and the circuit ansatz. The design scheme implemented in this work allows us to tune the VQE components to pursue better performance. We present a listing of the code expressing this implementation in Figure 5, in which we define the different parameters of the VQE algorithm in a custom-tailored way. In the code snippet, @qjit is a directive to activate the QCOR just-in-time compiler, which compiles the kernel body into the intermediate representation, and QuaSiMo.getWorkflow is a utility function of the library to construct and initialise workflow objects of various types. In Figure 6a, we present simulations considering the Simultaneous Perturbation Stochastic Approximation (SPSA) for ansatz updating. There, we show the energy as a function of the quantum circuits used in the learning process with a budget of 200 function evaluations with 5000 shots (experimental repetitions) per evaluation by using the QCOR's VQE module, QuaSiMo.getWorkflow('vqe'), and by using the Qiskit class aqua.algorithms.VQE [19]. In both cases, we consider (0.,0.,0.) as the initial set of parameters, therefore the energy values in the first iterations are similar, due to the SPSA hyperparameter's calibration. As it is expected, since we are using the same optimiser in both quantum simulations, there is not a relevant difference between the learning paths. However, the execution time in QCOR is shorter. This feature is presented in Figure 6b, where we show the execution time rate between QCOR and Qiskit as a function of the classical optimisation algorithm. To evaluate the typical execution time of each optimiser, we consider 51 runs of VQE towards the generation of H_2 quantum ground state.

```
from qcor import *
# Hamiltonian for H2
H = -0.22278593024287607*Z(3) + ... + \
    0.04532220205777769*X(0)*X(1)*Y(2)*Y(3) - \
    0.09886396978427353
# Defining the ansatz
@qjit
def ansatz(q : qreg, params : List[float]):
    X(q[0])
    ...
    Rz(q[1],params[0])
    Rz(q[3],params[1])
    ...
    Rz(q[3],params[2])
    ...
    H(q[3])
    Rx(q[0], 1.57079)
# variational parameters
n_params = 3
# Create the problem model
problemModel =
    QuaSiMo.ModelFactory.createModel(ansatz,
                                     H,
                                     n_params)
# Create the optimizer: spsa
optimizer = createOptimizer('nlopt',
                            {'algorithm':'spsa'})
# Create the VQE workflow
workflow = QuaSiMo.getWorkflow('vqe',
                               {'optimizer': optimizer})
# Execute
result = workflow.execute(problemModel)
# Get the result
energy = result['energy']
```

FIGURE 5 Code snippet to learn the ground state energy of H_2 by the VQE. For the sake of simplicity, we have omitted most of the terms in the Hamiltonian and ansatz

3.2.1 | Symmetry reduction

The presence of symmetries, such as rotations, reflections, number of particles etc., in the Hamiltonian model allows us to map the model to a model with fewer qubits [45]. In H_2, we apply the QCOR's function operatorTransform('qubit-tapering', H) to reduce the four-qubit Hamiltonian, see Figure 5, to a one-qubit Hamiltonian model. We present this implementation in Figure 7, in which we transform the Hamiltonian introduced in Figure 5 into a one-qubit model, and we redefine the anzats. In the output section of Figure 7, we present the one-qubit model and the VQE's output that converge to a similar value of the four-qubit model.

3.2.2 | Fermion-qubit map

An important feature included in the QCOR compiler is the fermion-to-qubit mapping that facilitates the quantum state searching in VQE. In Figure 8, we present an example of how to use OpenFermion operators [47] in the VQE workflow. In that implementation, we define the ansatz by using Scipy and OpenFermion, the QCOR compiler decomposes the ansatz into quantum gates; we follow the same structure presented in Figure 5 for the VQE workflow.

3.3 | Quantum approximate optimisation algorithm

To further demonstrate the utility of QCOR, we present an implementation of the quantum approximate optimisation algorithm (QAOA) [17]. QAOA translates a classical cost function into a quantum operator H_C and then uses a variational quantum-classical optimisation loop to find quantum states that minimise the expectation value $\langle H_C \rangle$. The optimised

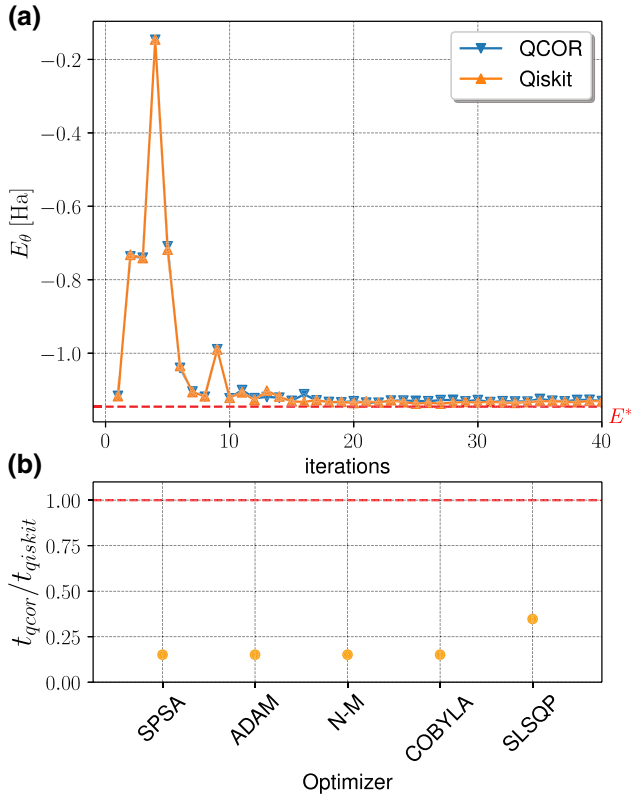

FIGURE 6 Energy ground state estimation for H_2 using VQE by using QCOR languange and Qiskit software. In panel (a), we present the VQE's learning path using the stochastic algorithm SPSA to find a quantum state with minimal energy. This plot shows how the energy E_θ approaches the exact value $E^* = -1.1456295$ Ha as the optimiser defines new quantum circuits, following the variational principle. In addition, panel (b) presents an execution time ratio between QCOR's VQE module and Qiskit's VQE module. The execution times ratio is below 1 (red dotted line) for different classical optimisers. The dots represent the 50th percentile of the execution time ratio from 51 independent runs of VQE. Optimiser labels correspond to Simultaneous Perturbation Stochastic Approximation (SPSA), ADAptive Moment estimation (ADAM), Nelder-Mead, also known as downhill simplex algorithm (N-M), Constraint Optimisation By Linear Approximation (COBYLA), and the Sequential Least SQuares Programming (SLSQP)

```
# Now we taper Hamiltonian H2
H_tapered = operatorTransform('qubit-tapering', H)
# For the new Hamiltonian
# we must define an one qubit ansatz
@qjit
def ansatz(q : qreg, phi : float, theta : float):
    Rx(q[0], phi)
    Ry(q[0], theta)

 $ Output:
 $ Reduced Hamiltonian:
 $ (-0.328717,0) + (0.181289,0) X0 + (-0.787967,0) Z0
 $ with energy = -1.13727017466
```

FIGURE 7 Symmetry reduction of the Hamiltonian model. In this snippet, we transform the Hamiltonian model used in 6 to a one-qubit model following symmetry arguments. This feature is implemented in the function operatorTransform('qubit-tapering', H)

```
from qcor import *
@qjit
def ansatz(q : qreg, x : List[float]):
    X(q[0])
    with decompose(q, kak) as u:
        from scipy.sparse.linalg import expm
        from openfermion.ops import QubitOperator
        from openfermion.transforms import \
            get_sparse_operator
        qop = QubitOperator('X0 Y1') \
            - QubitOperator('Y0 X1')
        qubit_sparse = get_sparse_operator(qop)
        u = expm(0.5j * x[0] * qubit_sparse).todense()
# Define the Hamiltonain
H = -2.1433 * X(0) * X(1) - 2.1433 * \
    Y(0) * Y(1) + .21829 * Z(0) - \
    6.125 * Z(1) + 5.907
num_params = 1
# Create the VQE problem model
problemModel =
    QuaSiMo.ModelFactory.createModel(ansatz,
                                     H,
                                     num_params)
# Create the NLOpt derivative free optimizer
optimizer = createOptimizer('nlopt')
# Create the VQE workflow
workflow = QuaSiMo.getWorkflow('vqe',
                    {'optimizer': optimizer})
# Execute the workflow
# to determine the ground-state energy
result = workflow.execute(problemModel)
energy = result['energy']
```

FIGURE 8 Here, we depict how to use OpenFermion operators to construct the state-preparation kernel (ansatz) for the VQE workflow. We use SciPy [46] and OpenFermion [47] to construct the exponential of $(X_0Y_1 - Y_0X_1)$ operator as a matrix. The matrix will be decomposed into quantum gates by the QCOR compiler

quantum states are then prepared and measured to obtain bitstrings that correspond to classical solutions to the optimisation problem.

Figure 9 shows an example python code that uses QCOR simulations to find optimised quantum states with QAOA. Opening lines define the number of qubits n and construct the problem Hamiltonian H_C for MaxCut on a star graph S_n. The Hamiltonian is then used to create the QuaSiMo model and workflow to simulate p-step QAOA and return the expectation $\langle H_C \rangle$, similar to the previous VQE example of Figure 5.

Figure 10 shows optimised energies $\langle H_C \rangle$ computed in QCOR for star graphs with numbers of qubits $n = 2, \ldots, 9$ at various QAOA depth parameters p. Each time the program is run it begins with random initial parameters to be , so we used several random starts for some of the graphs, keeping the smallest result as the $\langle H_C \rangle$ shown in the figure. The QCOR

```
from qcor import *
# QAOA for unweighted MaxCut
# Define a star graph Hamiltonian
n_qubits = 8
H = 0
for n in range(1,n_qubits):
    H += -0.5*(1-Z(0)*Z(n))
# create QuaSiMo model
problemModel = QuaSiMo.ModelFactory.createModel(H)
# Create the NLOpt derivative-free optimizer
optimizer = createOptimizer('nlopt')
# QAOA steps p
p = 3
# Quasimo workflow
workflow = QuaSiMo.getWorkflow('qaoa',
                {'optimizer': optimizer,
                'steps': p})
# Execute the workflow
# to determine the energy expectation value
result = workflow.execute(problemModel)
energy = result['energy']
```

FIGURE 9 Code to find the expectation value of the cost Hamiltonian with QAOA, for a star-graph instance of the unweighted MaxCut problem

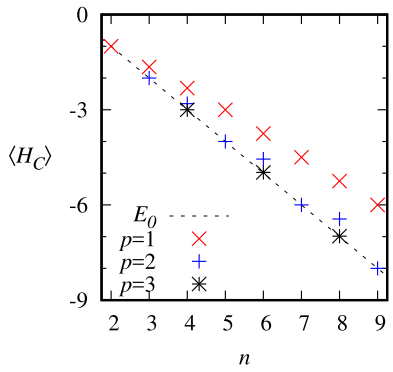

FIGURE 10 Optimised expectation values of the cost Hamiltonian H_C for star-graph instances of unweighted MaxCut with QAOA

results are in perfect agreement with the optimised standards [48] of Lotshaw et al. [49]. The results have interesting features, for example, when n is odd, perfect ground-state energies $\langle H_C \rangle = E_0$ are obtained at $p = 2$, whereas when n is even, the results need $p = 3$ to approach close to E_0. The simplicity of the QCOR code and simulations make this an attractive route to studying interesting features like these in future research on QAOA.

3.4 | Quantum imaginary time evolution algorithm

To compute the ground-state energy of an arbitrary Hamiltonian, in addition to VQE as we have demonstrated in Section 3.2, there is another algorithm, so-called quantum imaginary time evolution (QITE) [18, 50], which neither requires the use of an ansatz nor an optimiser. In QITE, we evolve the state through imaginary time $it \equiv \beta$ by applying the time-evolution operator $U = e^{-\beta H}$, which minimises the system energy exponentially. To do so, for each imaginary time-step $\Delta\beta$, we approximate the imaginary time evolution based on the results of the previous steps.

As shown in Figure 1, the QITE workflow is integrated into the QuaSiMo library. In this example, we demonstrate the use of QITE to find the ground-state energy of a three-qubit transverse field Ising model (TFIM),

$$H = J_z \sum_{i=1}^{N-1} \sigma_i^z \sigma_{i+1}^z + h_x \sum_{i=1}^{N} \sigma_i^x \tag{3}$$

where $J_z = h_x = -1.0$ and $N = 3$ is the number of spins (qubits).

Figure 11 is the code snippet to set up the QuaSiMo problem description and workflow for this problem.

The problem model is captured by a single Hamiltonian operator constructed by direct Pauli operator algebra. We also want to note that the Pauli operator algebra of QuaSiMo allows for hierarchical construction of the Hamiltonian, for example, via for loops, suitable for generic Hamiltonians similar to the one described in Equation (3).

For QITE workflow configurations, we set the step-size to 0.45 and steps to 20, for a total imaginary time of $\beta = 9.0$ ($\hbar = 1$). The system begins in different initial states by using a state-preparation circuit, as shown in Figure 11. Similar to the Python API, users can also use the utility function getWorkflow to retrieve an instance of the QITE workflow from the QCOR service registry with the name key 'qite' as shown in Figure 11.

The main drawback of QITE algorithm is that the propagating circuit size increases during the imaginary time-stepping procedure. To alleviate this constraint, especially for execution on NISQ hardware, QuaSiMo's QITE workflow implementation supports custom, externally provided circuit optimisers that will be invoked during the algorithm execution to minimise the circuit depth. In this demonstration, we take advantage of the QSearch [51] optimiser from BQSKit library, which is capable of synthesising constant-depth circuits for a variety of common use cases including the TFIM model in (3). For instance, in this example, the final QITE circuit (step = 20) has approximately 8000 gates (3000 CNOT gates), which is clearly beyond the capability of current NISQ devices. Thanks to QSearch, we can always re-synthesise a constant depth circuit with only 14 CNOT gates (87 gates in total) for any of the time steps, which is the theoretical lower bound [52], $\frac{1}{4}(4^n - 3n - 1) = 13.5$, for CNOT gate count in three-qubit circuits.

The results of the QITE workflow execution are shown in Figure 12, where we can see the energy value exponentially decays to the analytically computed ground-state energy of −3.49396 for all initial states.

```
// Initial state preparation
__qpu__ void state_prep(qreg q) {
  // e.g., |100>
  X(q[0]);
}

// Create the Hamiltonian: 3-qubit TFIM
auto observable = -(Z(0) * Z(1) + Z(1) * Z(2) +
                    X(0) + X(1) + X(2));
// Construct the problem model
auto problemModel =
       ModelFactory::createModel(state_prep,
                                 &observable);

// Create the qsearch IRTransformation
auto qsearch_optimizer =
    createTransformation("qsearch");
// QITE workflow: 20 steps with dbeta = 0.45
// Also, use qsearch to optimize
// the propagating circuit.
auto workflow =
    getWorkflow("qite",
                {{"steps", 20},
                {"step-size", 0.45},
                {"circuit-optimizer",
                        qsearch_optimizer}});
// Execute
auto result = workflow->execute(problemModel);
```

FIGURE 11 Code snippet (in C++) to find the ground-state energy of a three-qubit TFIM Hamiltonian using the QITE workflow. In this example, we run the QITE algorithm for a total imaginary time β of 9.0 ($d\beta = 0.45$, 20 steps). Additionally, we use the QSearch algorithm from BQSKit to optimise QITE circuits during workflow execution. The state_prep kernel is used to initialise the qubits into the desired initial state, for example, $|100\rangle$ in this particular case

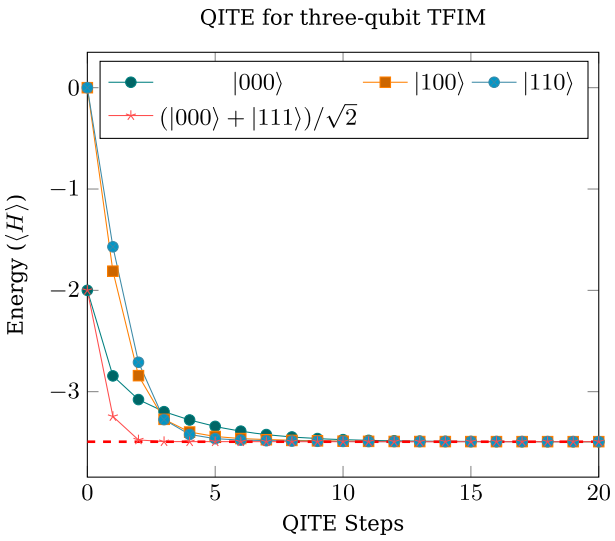

FIGURE 12 QITE results for 3-qubit TFIM with different initial states, step size = 0.45. The red dashed line represents the true ground-state energy of −3.49396

4 | CONCLUSIONS

We have presented and demonstrated a programming workflow for applications of quantum simulation that promotes common, reusable methods and data structures for scientific applications.

We note that while the framework presented here is readily applicable to use cases in the NISQ era – with the QuantumSimulationWorkflow in particular being extendable to NISQ simulation algorithms such as VQE – the workflow is also extendable to universal algorithms. Code portability is ensured through intermediate representation that is then implemented for each backend according to specific APIs, which are accessible in the QuaSiMo library. This enables rapid porting of a simulation algorithm to multiple machines (write once, run everywhere paradigm). This is an ideal environment to construct both benchmarks and validation protocols, along with short depth quantum simulations that can quickly be run on multiple processors and directly compared.

Although we anticipate that our composable design approach can serve the majority of applications and domains, we acknowledge that the problem model-simulation workflow-validation partitioning might not be suitable for certain specialised use cases. For instance, quantum simulation algorithms or applications having a monolithic structure, for example, interrelated solving and validation procedures, may not be readily represented by QuaSiMo. Nevertheless, by open-sourcing the library, we encourage new contributions, either at the design or implementation level, to best capture emerging use cases in this modular and composable design scheme.

While we maintain focus on rapid prototyping of quantum simulation on today's NISQ devices, we note that the nature of the intermediate representation and the modular backend structure enables targeting fault-tolerant devices as well. A fault-tolerant (FT) architecture may be represented as an additional backend with encoding transpiler pre-processing. While this configuration is ideal for QuaSiMo's unencoded qubit targets, we also note that the parent language QCOR is capable of expressing fully quantum-error-correction-encoded algorithms as well. Therefore, we expect the framework to be extendable and to find use in workflows involving universal or FT applications in the future.

ACKNOWLEDGEMENTS

This work was supported by the U.S. Department of Energy (DOE) Office of Science Advanced Scientific Computing Research program office Accelerated Research for Quantum Computing program. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

CONFLICT OF INTEREST

None.

PERMISSION TO REPRODUCE MATERIALS FROM OTHER SOURCES

Part of the content has been published in the IEEE 18th International Conference on Software Architecture Companion (ICSA-C) proceeding. We have requested reuse permission

from the IEEE and submitted it to the IET Quantum Communication editorial office (Sophie Robinson) for the record.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available in the QCOR repository at https://github.com/ORNL-QCI/qcor.

ORCID

Thien Nguyen https://orcid.org/0000-0002-5319-1213

REFERENCES

1. Aspuru-Guzik, A., et al.: Simulated quantum computation of molecular energies. Science. 309(5741), 1704–1707 (2005)
2. Whitfield, J.D., Biamonte, J., Aspuru-Guzik, A.: Simulation of electronic structure Hamiltonians using quantum computers. Mol. Phys. 109(5), 735–750 (2011)
3. Cao, Y., et al.: Quantum chemistry in the age of quantum computing. Chem. Rev. 119(19), 10856–10915 (2019)
4. McArdle, S., et al.: Quantum computational chemistry. Rev. Mod. Phys. 92(1), 015003 (2020)
5. Yeter-Aydeniz, K., Siopsis, G., Pooser, R.C.: Scattering in the ising model using quantum lanczos algorithm. arXiv preprint arXiv:200808763. (2020)
6. O'Malley, P.J., et al.: Scalable quantum simulation of molecular energies. Phys. Rev. X. 6(3), 031007 (2016)
7. Lidar, D.A., Wang, H.: Calculating the thermal rate constant with exponential speedup on a quantum computer. Phys. Rev. 59(2), 2429–2438 (1999)
8. Bassman Oftelie, L., et al.: Towards simulation of the dynamics of materials on quantum computers. Phys. Rev. B. 101(18), 184305 (2020)
9. Kosugi, T., Matsushita, Y.i.: Linear-response functions of molecules on a quantum computer: charge and spin responses and optical absorption. Phy. Rev. Research. 2(3), 033043 (2020)
10. Lysne, N.K., et al.: Small, highly accurate quantum processor for intermediate-depth quantum simulations. Phys. Rev. Lett. 124, 230501 (2020). https://link.aps.org/doi/10.1103/PhysRevLett.124.230501
11. Cirstoiu, C., et al.: Variational fast forwarding for quantum simulation beyond the coherence time. NPJ Quantum Inf. 6(1), 1–10 (2020)
12. Quantum, G.A., et al.: Hartree-Fock on a superconducting qubit quantum computer. Science. 369(6507), 1084–1089 (2020)
13. McClean, J.R., et al.: The theory of variational hybrid quantum-classical algorithms. New J. Phys. 18(2), 023023 (2016)
14. Romero, J., et al.: Strategies for quantum computing molecular energies using the unitary coupled cluster ansatz. Quantum Sci. Technol. 4(1), 014008 (2018)
15. Grimsley, H.R., et al.: An adaptive variational algorithm for exact molecular simulations on a quantum computer. Nat. Commun. 10(1), 1–6 (2019)
16. Tang, H.L., et al.: qubit-adapt-vqe: an adaptive algorithm for constructing hardware-efficient ansatze on a quantum processor. arXiv preprint arXiv:191110205. (2019)
17. Farhi, E., Goldstone, J., Gutmann, S.: A quantum approximate optimization algorithm. arXiv:14114028. (2014)
18. Motta, M., et al.: Determining eigenstates and thermal states on a quantum computer using quantum imaginary time evolution. Nat. Phys. 16(2), 1–6 (2019)
19. Qiskit: Qiskit aqua. Qiskit (2021). https://github.com/Qiskit/qiskit-aqua
20. Qiskit: Qiskit terra. Qiskit (2021). https://github.com/Qiskit/qiskit-terra
21. Kottmann, J.S., et al.: Tequila: a platform for rapid development of quantum algorithms. arXiv preprint arXiv:201103057. (2020)
22. Zapata. Orquestra. Zapata. (2021). https://www.zapatacomputing.com/orquestra
23. Kandala, A., et al.: Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. Nature. 549(7671), 242–246 (2017)
24. Hempel, C., et al.: Quantum chemistry calculations on a trapped-ion quantum simulator. Phys. Rev. X. 8(3), 031022 (2018)
25. McCaskey, A.J., et al.: Quantum chemistry as a benchmark for near-term quantum computers. NPJ Quantum Inf. 5(1), 1–8 (2019)
26. Yeter-Aydeniz, K., Pooser, R.C., Siopsis, G.: Practical quantum computation of chemical and nuclear energy levels using quantum imaginary time evolution and lanczos algorithms. NPJ Quantum Inf. 6(1), 1–8 (2020)
27. Qiskit: Introducing qiskit nature. Qiskit (2021). https://medium.com/qiskit/introducing-qiskit-nature-cb9e588bb004
28. Nguyen, T., et al.: Extending c++ for heterogeneous quantum-classical computing. arXiv preprint arXiv:201003935. (2020)
29. Marples, D., Kriens, P.: The open services gateway initiative: an introductory overview. IEEE Commun. Mag. 39(12), 110–114 (2002)
30. Britt, K.A., Humble, T.S.: High-performance computing with quantum processing units. ACM J. Emerg. Technol. Comput. Syst. 13(3), 1–13 (2017)
31. McCaskey, A.J., et al.: A language and hardware independent approach to quantum–classical computing. Software. 7, 245–254 (2018). http://www.sciencedirect.com/science/article/pii/S2352711018300700
32. Source code repository. GitLab. (2020). https://code.ornl.gov/elwasif/qsa-workflow
33. Mintz, T.M., et al.: Qcor: a language extension specification for the heterogeneous quantum-classical model of computation. ACM J. Emerg. Technol. Comput. Syst. 16(2), 1–17 (2020)
34. Trotter, H.F.: On the product of semi-groups of operators. Proc. Am. Math. Soc. 10(4), 545–545 (1959)
35. Suzuki, M.: Generalized trotter's formula and systematic approximants of exponential operators and inner derivations with applications to many-body problems. Commun. Math. Phys. 51(2), 183–190 (1976)
36. Barkoutsos, P.K., et al.: Quantum algorithms for electronic structure calculations: particle-hole Hamiltonian and optimized wave-function expansions. Phys. Rev. 98(2), 022322 (2018)
37. O'Brien, T.E., et al.: Error mitigation via verified phase estimation. arXiv preprint arXiv:201002538. (2020)
38. Qcor – c++ compiler for heterogeneous quantum-classical computing built on clang and xacc. GitHub. (2020). https://github.com/ORNL-QCI/qcor
39. Nguyen, T., et al.: Composable programming of hybrid workflows for quantum simulation. In: 2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C), pp. 110–116. (2021)
40. Barmettler, P., et al.: Quantum quenches in the anisotropic spin-heisenberg chain: different approaches to many-body dynamics far from equilibrium. New J. Phys. 12(5), 055017 (2010)
41. Preskill, J.: Quantum computing in the NISQ era and beyond. Quantum. 2, 79 (2018)
42. Wiebe, N., et al.: Simulating quantum dynamics on a quantum computer. J. Phys. Math. Theor. 44(44), 445308 (2011)
43. Peruzzo, A., et al.: A variational eigenvalue solver on a photonic quantum processor. Nat. Commun. 5(1), 4213 (2014). https://doi.org/10.1038/ncomms5213
44. Benedetti, M., et al.: A generative modeling approach for benchmarking and training shallow quantum circuits. NPJ Quantum Inf. 5(1), 45 (2019). https://doi.org/10.1038/s41534-019-0157-8
45. Bravyi, S., et al.: Tappering off qubits to simulate fermionic Hamiltonian. arXiv preprint arXiv:170108213. (2017)
46. Virtanen, P., et al.: SciPy 1.0: fundamental algorithms for scientific computing in Python. Nat. Methods. 17, 261–272 (2020)

47. McClean, J.R., et al.: Openfermion: the electronic structure package for quantum computers (2019)
48. Lotshaw, P.C., Humble, T.: QAOA dataset. (2021). https://code.ornl.gov/qci/qaoa-dataset-version1
49. Lotshaw, P.C., et al.: Empirical performance bounds for quantum approximate optimization. arXiv:2102:06813. (2021)
50. Yeter-Aydeniz, K., Siopsis, G., Pooser, R.C.: Scattering in the ising model with the quantum lanczos algorithm. New J. Phys. 23(4), 043033 (2021). https://doi.org/10.1088/1367-2630/abe63d
51. Davis, M.G., et al.: Towards optimal topology aware quantum circuit synthesis. In: 2020 IEEE International Conference on Quantum Computing and Engineering (QCE), pp. 223–234. (2020)
52. Shende, V.V., Markov, I.L., Bullock, S.S.: Minimal universal two-qubit controlled-not-based circuits. Phys. Rev. A. 69, 062321 (2004). https://link.aps.org/doi/10.1103/PhysRevA.69.062321

How to cite this article: Nguyen, T., et al.: QuaSiMo: a composable library to program hybrid workflows for quantum simulation. IET Quant. Comm. 2(4), 160–170 (2021). https://doi.org/10.1049/qtc2.12024