Lawrence Berkeley National Laboratory
LBL Publications

Title

Composable Programming of Hybrid Workflows for Quantum Simulation

Permalink
https://escholarship.org/uc/item/32r2g30n

Authors
Nguyen, T
Bassman Oftelie, L
Lyakh, D
et al.

Publication Date
2021-03-01

DOI
10.1109/ICSA-C52384.2021.00028

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Composable Programming of Hybrid Workflows for Quantum Simulation

Thien Nguyen[*¶], Lindsay Bassman[†], Dmitry Lyakh[‡¶], Alexander McCaskey[*¶], Vicente Leyton-Ortega[§¶],
Raphael Pooser[§¶], Wael Elwasif[*¶], Travis S. Humble[§¶], and Wibe A. de Jong[†]
[*]Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, 37831, USA
[†]Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, California 94720, USA
[‡]National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, TN, 37831, USA
[§]Computer Science and Engineering Division, Oak Ridge National Laboratory, Oak Ridge, TN, 37831, USA
[¶]Quantum Computing Institute, Oak Ridge National Laboratory, Oak Ridge, TN, 37831, USA

arXiv:2101.08151v1 [quant-ph] 20 Jan 2021

***Abstract*—We present a composable design scheme for the development of hybrid quantum/classical algorithms and workflows for applications of quantum simulation. Our object-oriented approach is based on constructing an expressive set of common data structures and methods that enable programming of a broad variety of complex hybrid quantum simulation applications. The abstract core of our scheme is distilled from the analysis of the current quantum simulation algorithms. Subsequently, it allows a synthesis of new hybrid algorithms and workflows via the extension, specialization, and dynamic customization of the abstract core classes defined by our design. We implement our design scheme using the hardware-agnostic programming language QCOR into the `QuaSiMo` library. To validate our implementation, we test and show its utility on commercial quantum processors from IBM, running some prototypical quantum simulations.**

***Index Terms*—quantum computing, quantum programming, programming languages**

I. Introduction

Quantum simulation is an important use case of quantum computing for scientific computing applications. Whereas 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 optimized 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 optimization algorithm (QAOA), quantum imaginary time evolution (QITE) [17], 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. 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 materials science and chemistry, and we discuss how to extend these workflows to experimental validation of quantum

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 and Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. (http://energy.gov/downloads/doe-public-access-plan).

computation advantage, in which numerical simulations can benchmark programs for small-sized models [12], [18]–[21].

II. Software Architecture

Cloud-based access to quantum computing naturally differentiates programming into conventional and quantum tasks [22], [23]. The resulting hybrid execution model yields a loosely integrated computing system by which common methods have emerged for programming and data flow. We emphasize this concept of workflow to organize programming applications for quantum simulation. Figure 1 shows the blueprint of our Quantum Simulation Modeling (`QuaSiMo`) library. The programming workflow is defined by a `QuantumSimulationWorkflow` concept which encapsulates the hybrid quantum-classical procedures pertinent to a quantum simulation, e.g., 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 magnetization, 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 customized 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 customizable 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 [25], `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 Fig. 1. All these plugins are implemented in the QCOR language [26], [27] 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 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 [28], [29] or the unitary coupled-cluster (UCC) ansatz [30]. `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, e.g., the verified quantum phase estimation protocol [31], 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 which evaluates a distance measure method and optionally calls a decision method which returns a binary

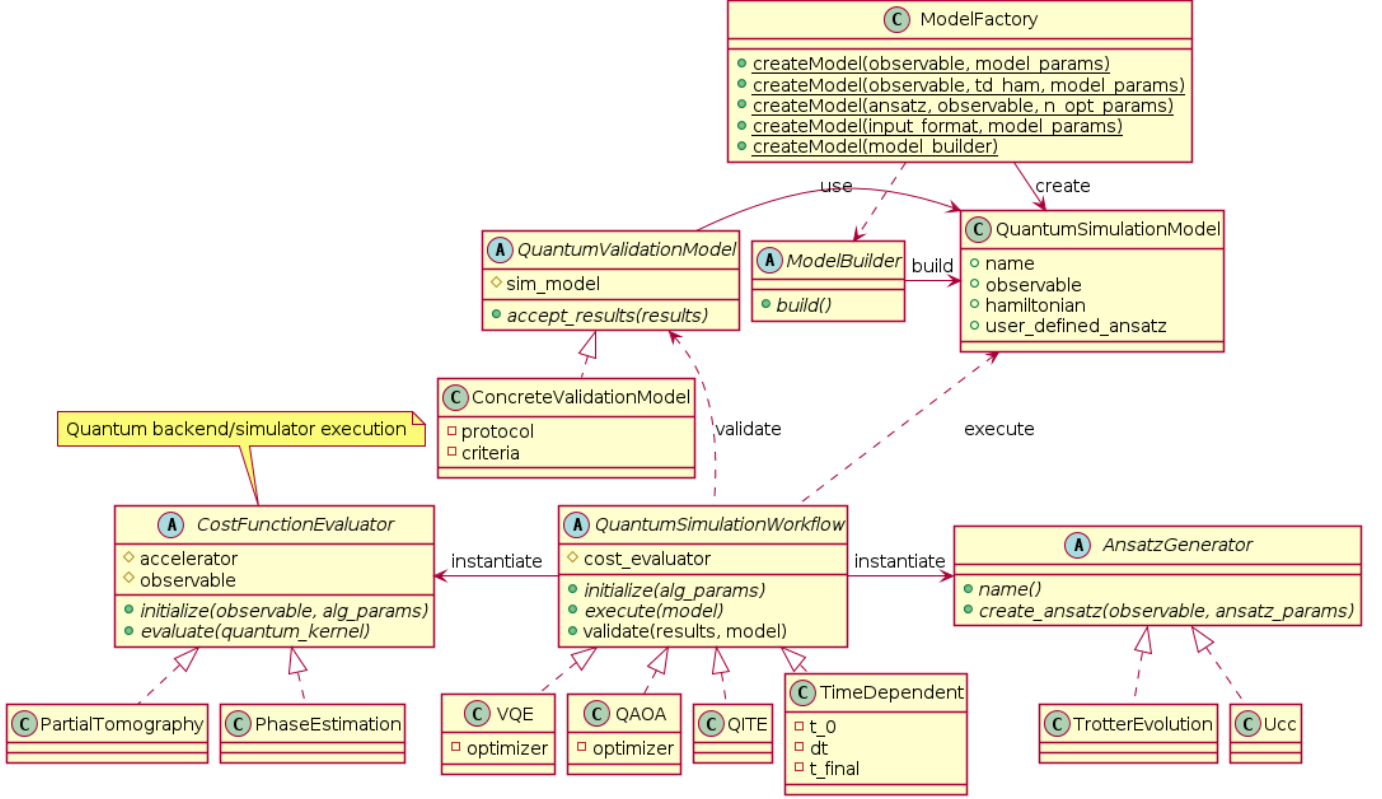


Fig. 1: The class UML diagram of the quantum simulation application. The fully typed version is provided separately (see [24]).

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 standardized 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.

III. Testing and Evaluation

Our implementation of the programming workflow for applications of quantum simulation is available online [32]. We have tested this implementation against several of the original use cases to validate the correctness of the implementation and to evaluate performance considerations.

A. 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 initializing 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 neighbor 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 magnetization [33], 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.

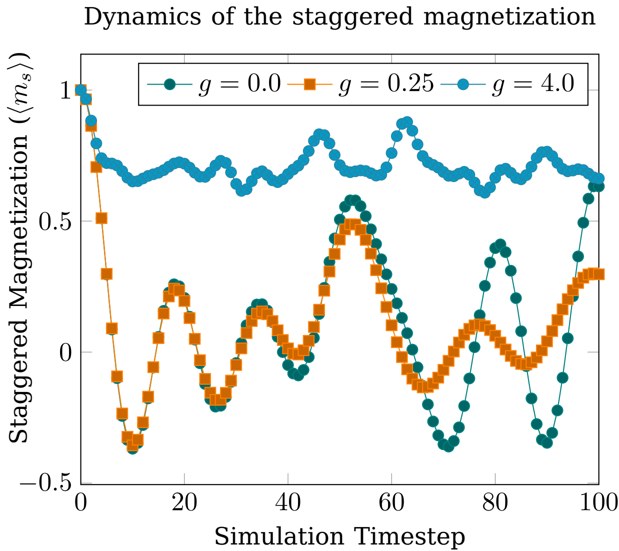


Fig. 2: Simulation results of staggered magnetization for an Heisenberg model with nine spins after a quantum quench. The Trotter step size (dt) is 0.05.

```
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 = getWorkflow("td-evolution",
                 {{"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");
```

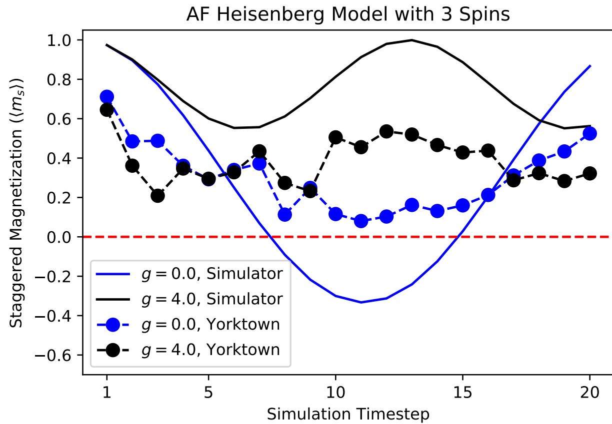
Fig. 3: Defining the AF Heisenberg problem model and simulating its dynamics with `QuaSiMo`. In this example, `g` is the anisotropy parameter, as shown in equation 1, and `n_spins` is the number of spins/qubits. `initial_spins` is an array of 0 or 1 values denoting the initial spin state. `initial_spins` was initialized (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.

Fig. 2 shows sample results for $N = 9$ spins for a three different values for g in H_f. The qualitatively different behaviours of the staggered magnetization after the quench for $g < 1$ and $g > 1$ are apparent, and agree with previous studies [33]. We present a listing of the code expressing this implementation in Fig. 3.

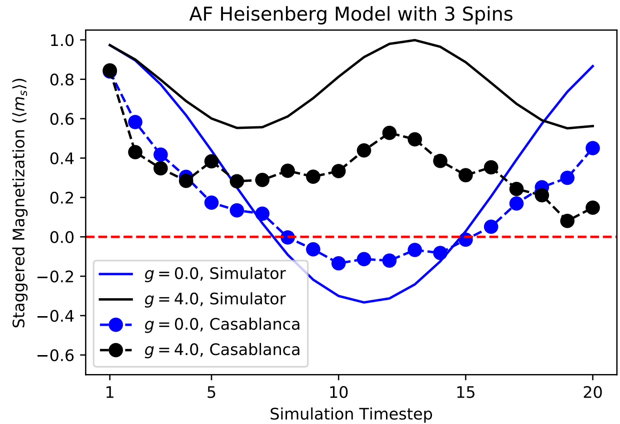
We develop `QuaSiMo` on top of the QCOR infrastructure, as shown in Fig. 1; thus, any quantum simulation workflows constructed in `QuaSiMo` are retargetable to a broad range of quantum backends. The results that we have demonstrated in Fig. 2 are from a simulator backend. The same code as shown in Fig. 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 [34], 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 [35]. To limit the circuit size, we simulated a small AF Heisenberg model, eq. 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 Fig. 4, where we can see the effects of gate-errors and qubit decoherence leading to a significant impairment of the measured staggered magnetization (circles) compared to the theoretical values (solid lines). In particular,



(a) IBMQ Yorktown device



(b) IBMQ Casablanca device

Fig. 4: Results of simulation an AF model (eq. 1) for a system with three spins using the code snippet in Fig. 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 optimized using the QCOR compiler before submitting for execution. The Trotter time-step (dt) is 0.05.

Fig. 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, 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.

The staggered magnetization response to a quench for a simple three-qubit AF Heisenberg model in Fig. 4, albeit noisy, illustrates non-trivial dynamics beyond that of decoherence (decaying to zero). Improvements in circuit construction (Trotter decomposition) and optimization, 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.

B. Variational Quantum Eigensolver

As a second use case demonstration, we apply the Variational Quantum Eigensolver (VQE) to find the ground state energy of H_2. The VQE is a quantum-classic hybrid algorithm used to find a Hamiltonian's eigenvalues, where the quantum process side is represented by a parametrized quantum circuit whose parameters are updated by a classical optimization process [36]. The algorithm updates the quantum circuit parameters θ to minimize the Hamiltonian's expectation value E_θ until it converges.

The performance of the VQE algorithm, as any other quantum-classical variational algorithms, [37] depends on the selection of the classical optimizer 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 Fig. 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. In Fig. 6, we present simulations considering different classical optimizers for ansatz updating. From those simulations, we can infer the quality of the chosen ansatz and the classical optimizer, when after a few quantum circuits, the prepared state's energy approaches the exact value E^*.

An important feature included in the QCOR compiler is the fermion-to-qubit mapping that facilitates the quantum state searching in VQE. In Fig. 7, we present an example of how to use OpenFermion operators [39] 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 Fig. 5 for

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

```
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 NLOpt derivative free optimizer
optimizer = createOptimizer('nlopt')
# Create the VQE workflow
workflow = QuaSiMo.getWorkflow('vqe',
                   {'optimizer': optimizer})
# Execute
result = workflow.execute(problemModel)
# Get the result
energy = result['energy']
```

Fig. 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.

the VQE workflow.

IV. 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

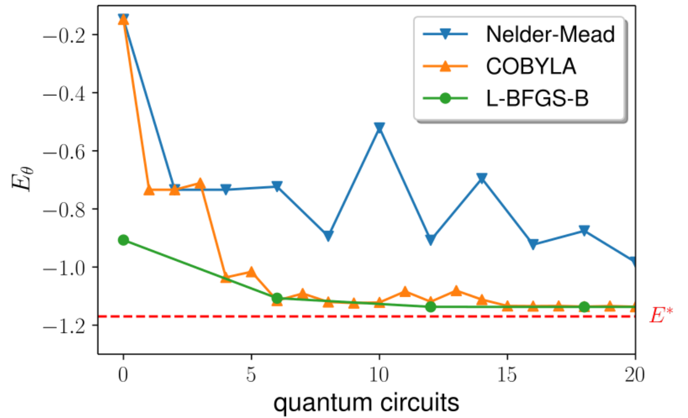


Fig. 6: Energy ground state estimation for H2 using VQE. We consider three different classical optimizers, Nelder-Mead, COBYLA, and L-BFGS-B, to update the quantum circuit parameters. This plot shows how the energy E_θ approaches to the exact value E^* as the optimizer defines new quantum circuits, following the variational principle.

- the workflow is also extendable to universal algorithms. Further, simulators which rely on fault tolerant (FT) methods can be built by abstracting stabilizer code or other quantum error correction code classes and by providing polymorphs of the simulator `execute` method, which implements the necessary rules and machinery of FT algorithms during execution. In this manner, an FT simulation workflow can be implemented with an abstracted FT backend, such that simulators written with the original `QuantumSimulationWorkflow` class can be called with a FT execute method. Therefore, we expect the framework to be extendable and to find use in workflows involving universal or FT applications in the future.

Acknowledgments

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.

References

[1] A. Aspuru-Guzik, A. D. Dutoi, P. J. Love, and M. Head-Gordon, “Simulated quantum computation of molecular energies,” *Science*, vol. 309, no. 5741, pp. 1704–1707, 2005.

[2] J. D. Whitfield, J. Biamonte, and A. Aspuru-Guzik, “Simulation of electronic structure hamiltonians using quantum computers,” *Molecular Physics*, vol. 109, no. 5, pp. 735–750, 2011.

```python
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']
```

Fig. 7: Here we depict how to use OpenFermion operators to construct the state-preparation kernel (ansatz) for the VQE workflow. We use SciPy [38] and OpenFermion [39] 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.

[3] Y. Cao, J. Romero, J. P. Olson, M. Degroote, P. D. Johnson, M. Kieferová, I. D. Kivlichan, T. Menke, B. Peropadre, N. P. Sawaya *et al.*, "Quantum chemistry in the age of quantum computing," *Chemical reviews*, vol. 119, no. 19, pp. 10 856–10 915, 2019.

[4] S. McArdle, S. Endo, A. Aspuru-Guzik, S. C. Benjamin, and X. Yuan, "Quantum computational chemistry," *Reviews of Modern Physics*, vol. 92, no. 1, p. 015003, 2020.

[5] K. Yeter-Aydeniz, G. Siopsis, and R. C. Pooser, "Scattering in the ising model using quantum lanczos algorithm," *arXiv preprint arXiv:2008.08763*, 2020.

[6] P. J. O'Malley, R. Babbush, I. D. Kivlichan, J. Romero, J. R. McClean, R. Barends, J. Kelly, P. Roushan, A. Tranter, N. Ding *et al.*, "Scalable quantum simulation of molecular energies," *Physical Review X*, vol. 6, no. 3, p. 031007, 2016.

[7] D. A. Lidar and H. Wang, "Calculating the thermal rate constant with exponential speedup on a quantum computer," *Physical Review E*, vol. 59, no. 2, p. 2429, 1999.

[8] L. Bassman, K. Liu, A. Krishnamoorthy, T. Linker, Y. Geng, D. Shebib, S. Fukushima, F. Shimojo, R. K. Kalia, A. Nakano *et al.*, "Towards simulation of the dynamics of materials on quantum computers," *Physical Review B*, vol. 101, no. 18, p. 184305, 2020.

[9] T. Kosugi and Y.-i. Matsushita, "Linear-response functions of molecules on a quantum computer: Charge and spin responses and optical absorption," *Physical Review Research*, vol. 2, no. 3, p. 033043, 2020.

[10] N. K. Lysne, K. W. Kuper, P. M. Poggi, I. H. Deutsch, and P. S. Jessen, "Small, highly accurate quantum processor for intermediate-depth quantum simulations," *Phys. Rev. Lett.*, vol. 124, p. 230501, Jun 2020. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.124.230501

[11] C. Cirstoiu, Z. Holmes, J. Iosue, L. Cincio, P. J. Coles, and A. Sornborger, "Variational fast forwarding for quantum simulation beyond the coherence time," *npj Quantum Information*, vol. 6, no. 1, pp. 1–10, 2020.

[12] G. A. Quantum *et al.*, "Hartree-fock on a superconducting qubit quantum computer," *Science*, vol. 369, no. 6507, pp. 1084–1089, 2020.

[13] J. R. McClean, J. Romero, R. Babbush, and A. Aspuru-Guzik, "The theory of variational hybrid quantum-classical algorithms," *New Journal of Physics*, vol. 18, no. 2, p. 023023, 2016.

[14] J. Romero, R. Babbush, J. R. McClean, C. Hempel, P. J. Love, and A. Aspuru-Guzik, "Strategies for quantum computing molecular energies using the unitary coupled cluster ansatz," *Quantum Science and Technology*, vol. 4, no. 1, p. 014008, 2018.

[15] H. R. Grimsley, S. E. Economou, E. Barnes, and N. J. Mayhall, "An adaptive variational algorithm for exact molecular simulations on a quantum computer," *Nature Communications*, vol. 10, no. 1, pp. 1–9, 2019.

[16] H. L. Tang, E. Barnes, H. R. Grimsley, N. J. Mayhall, and S. E. Economou, "qubit-adapt-vqe: An adaptive algorithm for constructing hardware-efficient ansatze on a quantum processor," *arXiv preprint arXiv:1911.10205*, 2019.

[17] M. Motta, C. Sun, A. T. Tan, M. J. O'Rourke, E. Ye, A. J. Minnich, F. G. Brandão, and G. K.-L. Chan, "Determining eigenstates and thermal states on a quantum computer using quantum imaginary time evolution," *Nature Physics*, pp. 1–6, 2019.

[18] A. Kandala, A. Mezzacapo, K. Temme, M. Takita, M. Brink, J. M. Chow, and J. M. Gambetta, "Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets," *Nature*, vol. 549, no. 7671, p. 242, 2017.

[19] C. Hempel, C. Maier, J. Romero, J. McClean, T. Monz, H. Shen, P. Jurcevic, B. P. Lanyon, P. Love, R. Babbush *et al.*, "Quantum chemistry calculations on a trapped-ion quantum simulator," *Physical Review X*, vol. 8, no. 3, p. 031022, 2018.

[20] A. J. McCaskey, Z. P. Parks, J. Jakowski, S. V. Moore, T. D. Morris, T. S. Humble, and R. C. Pooser, "Quantum chemistry as a benchmark for near-term quantum computers," *npj Quantum Information*, vol. 5, no. 1, pp. 1–8, 2019.

[21] K. Yeter-Aydeniz, R. C. Pooser, and G. Siopsis, "Practical quantum computation of chemical and nuclear energy levels using quantum imaginary time evolution and lanczos algorithms," *npj Quantum Information*, vol. 6, no. 1, pp. 1–8, 2020.

[22] K. A. Britt and T. S. Humble, "High-performance computing with quantum processing units," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, pp. 1–13, 2017.

[23] A. McCaskey, E. Dumitrescu, D. Liakh, M. Chen, W. Feng, and T. Humble, "A language and hardware independent approach to quantum–classical computing," *SoftwareX*, vol. 7, pp. 245

– 254, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352711018300700

[24] "Source code repository," https://code.ornl.gov/elwasif/qsa-workflow, 2020.

[25] D. Marples and P. Kriens, "The open services gateway initiative: An introductory overview," *IEEE Communications magazine*, vol. 39, no. 12, pp. 110–114, 2001.

[26] T. M. Mintz, A. J. McCaskey, E. F. Dumitrescu, S. V. Moore, S. Powers, and P. Lougovski, "Qcor: A language extension specification for the heterogeneous quantum-classical model of computation," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 16, no. 2, pp. 1–17, 2020.

[27] T. Nguyen, A. Santana, T. Kharazi, D. Claudino, H. Finkel, and A. McCaskey, "Extending c++ for heterogeneous quantum-classical computing," *arXiv preprint arXiv:2010.03935*, 2020.

[28] H. F. Trotter, "On the product of semi-groups of operators," *Proceedings of the American Mathematical Society*, vol. 10, no. 4, pp. 545–551, 1959.

[29] M. Suzuki, "Generalized trotter's formula and systematic approximants of exponential operators and inner derivations with applications to many-body problems," *Communications in Mathematical Physics*, vol. 51, no. 2, pp. 183–190, 1976.

[30] P. K. Barkoutsos, J. F. Gonthier, I. Sokolov, N. Moll, G. Salis, A. Fuhrer, M. Ganzhorn, D. J. Egger, M. Troyer, A. Mezzacapo *et al.*, "Quantum algorithms for electronic structure calculations: Particle-hole hamiltonian and optimized wave-function expansions," *Physical Review A*, vol. 98, no. 2, p. 022322, 2018.

[31] T. E. O'Brien, S. Polla, N. C. Rubin, W. J. Huggins, S. McArdle, S. Boixo, J. R. McClean, and R. Babbush, "Error mitigation via verified phase estimation," *arXiv preprint arXiv:2010.02538*, 2020.

[32] "Qcor - c++ compiler for heterogeneous quantum-classical computing built on clang and xacc," https://github.com/ORNL-QCI/qcor, 2020.

[33] P. Barmettler, M. Punk, V. Gritsev, E. Demler, and E. Altman, "Quantum quenches in the anisotropic spin-heisenberg chain: different approaches to many-body dynamics far from equilibrium," *New Journal of Physics*, vol. 12, no. 5, p. 055017, 2010.

[34] J. Preskill, "Quantum computing in the nisq era and beyond," *Quantum*, vol. 2, p. 79, 2018.

[35] N. Wiebe, D. W. Berry, P. Høyer, and B. C. Sanders, "Simulating quantum dynamics on a quantum computer," *Journal of Physics A: Mathematical and Theoretical*, vol. 44, no. 44, p. 445308, 2011.

[36] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, and J. L. O'Brien, "A variational eigenvalue solver on a photonic quantum processor," *Nature Communications*, vol. 5, no. 1, p. 4213, Jul. 2014. [Online]. Available: https://doi.org/10.1038/ncomms5213

[37] M. Benedetti, D. Garcia-Pintos, O. Perdomo, V. Leyton-Ortega, Y. Nam, and A. Perdomo-Ortiz, "A generative modeling approach for benchmarking and training shallow quantum circuits," *npj Quantum Information*, vol. 5, no. 1, p. 45, May 2019. [Online]. Available: https://doi.org/10.1038/s41534-019-0157-8

[38] 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. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. 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, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[39] J. R. McClean, K. J. Sung, I. D. Kivlichan, Y. Cao, C. Dai, E. S. Fried, C. Gidney, B. Gimby, P. Gokhale, T. Häner, T. Hardikar, V. Havlíček, O. Higgott, C. Huang, J. Izaac, Z. Jiang, X. Liu, S. McArdle, M. Neeley, T. O'Brien, B. O'Gorman, I. Ozfidan, M. D. Radin, J. Romero, N. Rubin, N. P. D. Sawaya, K. Setia, S. Sim, D. S. Steiger, M. Steudtner, Q. Sun, W. Sun, D. Wang, F. Zhang, and R. Babbush, "Openfermion: The electronic structure package for quantum computers," 2019.