A backend-agnostic, quantum-classical framework for simulations of chemistry in C++

DANIEL CLAUDINO, Computer Science and Mathematics, Oak Ridge National Laboratory, USA

ALEXANDER J. MCCASKEY, Computer Science and Mathematics, Oak Ridge National Laboratory, USA

DMITRY I. LYAKH, National Center for Computational Sciences, Oak Ridge National Laboratory, USA

As quantum computing hardware systems continue to advance, the research and development of performant, scalable, and extensible software architectures, languages, models, and compilers is equally as important in order to bring this novel coprocessing capability to a diverse group of domain computational scientists. For the field of quantum chemistry, applications and frameworks exists for modeling and simulation tasks that scale on heterogeneous classical architectures, and we envision the need for similar frameworks on heterogeneous quantum-classical platforms. Here we present the XACC system-level quantum computing framework as a platform for prototyping, developing, and deploying quantum-classical software that specifically targets chemistry applications. We review the fundamental design features in XACC, with special attention to its extensibility and modularity for key quantum programming workflow interfaces, and provide an overview of the interfaces most relevant to simulations of chemistry. A series of examples demonstrating some of the state-of-the-art chemistry algorithms currently implemented in XACC are presented, while also illustrating the various APIs that would enable the community to extend, modify, and devise new algorithms and applications in the realm of chemistry.

1 INTRODUCTION

Recent experimental results leveraging noisy intermediate-scale quantum (NISQ) computers have demonstrated use cases whereby the quantum processor surpasses conventional computing technologies [4, 47], and have provided reassurance for the potential of quantum computing, even though the notion of supremacy/advantage is believed to be intimately tied to tailored operation toward a given domain of application. At the heart of such use cases is a heterogeneous compute model heavily reliant on the interplay between classical computers and quantum co-processors,

Authors' addresses: Daniel Claudino, Computer Science and Mathematics, Oak Ridge National Laboratory, 1 Bethel Valley Road, Oak Ridge, TN, USA, claudinodc@ornl.gov; Alexander J. McCaskey, Computer Science and Mathematics, Oak Ridge National Laboratory, 1 Bethel Valley Road, Oak Ridge, TN, USA, mccaskeyaj@ornl.gov; Dmitry I. Lyakh, National Center for Computational Sciences, Oak Ridge National Laboratory, 1 Bethel Valley Road, Oak Ridge, TN, USA, liakhdi@ornl.gov.

ultimately implying the necessity for a robust software infrastructure capable of realizing the demanded level of interoperability. Moreover, compliance with vendor-specific native instruction sets and offering an environment that enables hybrid algorithmic development — all the while remaining highly extensible and easy to maintain — are some of the key components of a software framework meant to be competitive and outlast the NISQ era into the fault-tolerant regime. These are some of the foundational design features of the XACC framework. [23]

Modern classical high performance computing has been steadily moving towards an increased level of heterogeneity in both supercomputing centers and clouds. The GPU-accelerated heterogeneous computer architectures are a *de facto* standard of data processing these days. On top of that, more specialized accelerators based on the FPGA and ASIC designs (e.g., Google's TPU) have been developed and already deployed in data centers, addressing specialized workflow needs, for example machine learning and artificial intelligence workloads. To accommodate a variety of computational workloads on diverse hardware, virtualization and service-oriented architectures have become in high performance computing. In the anticipation of even more heterogeneity in future tightly coupled computing platforms enhanced with quantum hardware accelerators, the hybrid quantum-classical programming framework XACC [23] is designed to serve as a system-level software infrastructure based on a service-oriented architecture. Implemented in modern C++, XACC provides low-latency access to any level of the quantum software stack with a high degree of customization and adaptation to a given hybrid workload via dynamically loaded user-defined plugins. This draws a sharp contrast with most of other quantum programming frameworks which focus on a higher level of language abstraction at the cost of native performance. The XACC framework also provides higher level functionality via both C++ and Python bindings, but it is built on a solid system-level core. Finally, in contrast to many vendor-developed frameworks, XACC is vendor-agnostic, thereby enabling execution of quantum-classical workloads on any physical or virtual backend.

The inherent disposition of quantum bits (qubits) to encode entanglement make natural their usage for simulation of entangled quantum entities, with chemistry being among the most championed applications. There is a wide array of tangible technical areas that are to greatly benefit from an exponential improvement in the speed and scale which computational chemistry can provide a decisive answer, such as drug design,[6, 7, 35] green catalysts,[34, 43] and solar cells.[8, 11] While time evolution of a general Hamiltonian is still reserved for devices with a much lower fault rate, the electronic structure of molecules is believed to be addressable within the NISQ paradigm. Such an advance has sparked a great effort in rethinking established formalisms and introducing novel ideas, resulting in new algorithmic strategies for detailing and predicting the electronic structure of systems with chemical interest while exhibiting resource demands that are within the constraints observed in NISQ devices. Advanced software engineering stands to play a critical role in the realization of such ideas, providing the common ground mediating algorithmic development and hardware execution.

With this in mind, in this work we show the utility of XACC as backend-agnostic, hybrid quantum-classical framework that enables development, execution, and benchmarking of chemistry simulations. In the following, we review the core concept of a service-oriented architecture in XACC (Subsection 2) as the foundational design feature in enabling modularity. This gives the necessary context for a proper presentation of the most relevant interfaces involved in applications of chemical relevance (Subsection 3). Demonstrations of a variety of quantum algorithms for chemistry simulations are presented and discussed as well as a brief investigation highlighting the superior performance of XACC and its intermediate representation. We close with a summary of the main elements considered throughout the manuscript, along with a brief outlook of how XACC fits into the present and future of quantum software for chemistry applications.

2 SERVICE-ORIENTED ARCHITECTURE

Ultimately, XACC puts forward a service-oriented architecture — the underlying software architecture of XACC defines a collection of core C++ classes with pure virtual methods (we henceforth call them *interfaces* or *services* since interface in C++ can be modeled via classes with pure-virtual methods) that are intended to be implemented by concrete sub-types and contributed as plugins. XACC puts forward interfaces for language parsing / compiling, describing a domain core intermediate representation, transforming instances of that IR, backend execution, and high-level algorithm description. These interfaces are implemented, compiled into shared-libraries, and contributed to the underlying XACC framework via installation to a pre-specified *plugin* directory. At startup, XACC scans this directory and loads all available plugins, and a corresponding public API call enables programmers to retrieve specific plugin instances corresponding to their unique name and interface type. XACC follows this pattern in an effort to promote maximal extensibility and modularity — this enables XACC installations to grow over time and adapt or evolve to the rapidly maturing quantum-classical programming, compilation, and execution research field. It also enables developer productivity in that the framework provides core functionality and extensions are provided as simple sub-type definitions.

To enable this service-oriented architecture, XACC builds upon the CppMicroServices framework [1] — a C++ native implementation of the Open Services Gateway Initiative (OSGi).[2] The OSGi specification has proved extremely successful in the development service-oriented architectures in Java (e.g. the Eclipse IDE), and promotes the concept of a *bundle* as the core unit of extensibility. Bundles declare the services that they provide, and can be deployed in a stand-alone fashion, with the underlying framework implementation understanding how to search for and load available bundle plugins. In XACC, all bundles are deployed as shared libraries containing specific core XACC interface sub-type implementations and code that declares and registers this sub-type with the underlying framework (an `Activator`). Additionally, all services must inherit from the `xacc::Identifiable` interface, which requires implementation of a
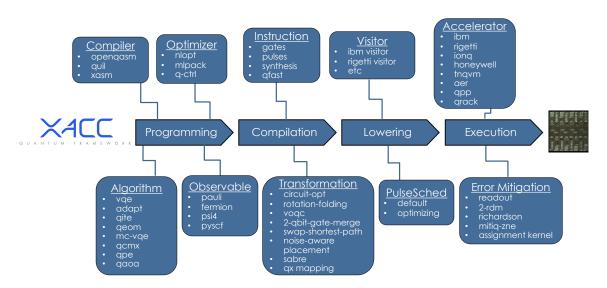

Fig. 1. Some pertinent XACC interfaces and concrete implementations at different stages of the quantum computing workflow.

`name()` method returning the unique name of the plugin sub-type. The core of XACC provides a `ServiceRegistry` data type that is created at startup and initializes and loads all available plugins and registers them with the underlying CppMicroServices framework instance. The `ServiceRegistry` exposes a service retrieval API that enables programmers to request plugin instances by name and XACC service interface type (e.g. `getService<Accelerator>("ibm")`). Through this approach, at the high-level, algorithm programmatic execution consists of the retrieval of pertinent service instances specific to the algorithm, language, and quantum backend desired for the current context.

We now proceed with a brief discussion on the interfaces that are most relevant in chemistry applications.

3 XACC INTERFACES

XACC enables an extensible workflow for the tasks associated with quantum programming, compilation, and execution via the service-oriented architecture described above. Here we detail the interfaces promoted by XACC that are pertinent for the high-level description and execution of common workflows for quantum chemistry utilizing quantum coprocessors. We note that in this work we seek to provide high-level details on these pertinent interface definitions, but leave a more extensive discussion to [24].

3.1 Accelerator and AcceleratorBuffer

To enable extensibility for backend quantum execution XACC puts forward the `Accelerator` interface. This interface enables one to inject custom backend implementations for available quantum hardware (IBM, Rigetti, Honeywell, etc.) as well as numerical simulators that scale from laptops to Summit. Ultimately, `Accelerators` provide an API for one-time backend intitialization and execution of compiled quantum programs. The `initialize()` method takes a map of key-value pairs — where keys are strings and values can be any type (`HeterogenousMap`, models a Python `dict`) — with the desired parameters to control the accelerator execution, such as number of shots and specific backend type (`ibmq_rome` for example), among others. The execution of a circuit instance, or a collection of instances, is carried out via `Accelerator::execute()`, which takes as arguments a qubit register (or an abstraction thereof, see below discussion on `AcceleratorBuffer`) and the compiled quantum program (see section on XACC IR). The goal of `execute()` is to map the incoming quantum program to the native instruction set of the targeted backend in the appropriate data format, affect execution of that representation, and post execution results to the provided qubit register buffer.

XACC currently provides public support for superconducting quantum architectures from IBM and Rigetti and the D-Wave quantum annealers, given the users have the required credentials. Due to difficulties posed by operation of quantum computers in the NISQ era, the employment of virtual simulators is quite common in quantum algorithms research and applications. We offer plugins to several numerical simulators that ship with the XACC installation, among which are the Quantum++ (QPP)[12] and Qrack C++ libraries, the Qsim and Aer statevector simulators, and the D-Wave Neal annealer simulator. The TNQVM[22] plugin exploits the quantum circuit topology and maps quantum circuits onto tensor networks, with a subsequent interface to highly efficient libraries for tensor network manipulations. It also offers the capabilities for density matrix simulations, thereby enabling inclusion of noise channels and backend specific noise models.

The quantum register buffer is modeled through a data type called the `AcceleratorBuffer`. This type abstracts a register of qubits, with each qubit assigned a unique integer index. Programmers create instances of this data type via a public `qalloc` call providing the size of the register (number of qubits). Programmers keep reference to this buffer handle and pass it to invocations of `Accelerator::execute()`, which subsequently persists execution results to the buffer, thereby giving the programmer access these results through the existing buffer handle reference. Another primary role

performed by this data type is the storage of execution-specific metadata such as backend noise information, compute statistical values, etc. `Accelerator` implementations are free to persist any execution-relevant metadata that is then readily available to the programmer with reference to the buffer.

3.2 The XACC Intermediate Representation

XACC provides a polymorphic, extensible intermediate representation for the in-memory expression of parsed and compiled quantum programs. At its core, XACC defines an `Instruction` interface, which is meant to be sub-typed for concrete quantum instructions. Each `Instruction` exposes a unique name, the qubit indices that it operates on, and any instruction parameters. Instruction parameters may be provided as string variables or as concrete numeric values, which is useful for the expression of parameterized quantum circuits and templated parameterized quantum circuits (string variable gate rotations which can be made concrete via the evaluation of the variables at numerical values). Moving up the abstraction hierarchy, XACC puts forward the `CompositeInstruction`, which subtypes `Instruction` but contains further instructions, enabling a n-ary tree pattern on `Instruction` types (here nodes in the tree are `CompositeInstructions` and leaves are concrete `Instruction` subtypes. On top of these core IR interfaces, XACC also exposes a factory pattern (the `IRProvider` for the creation of concrete instructions, which ultimately delegates to the core `ServiceRegistry`.

3.3 Optimizer

Many use cases of quantum computing in the NISQ era rely on finding a set of parameters that locates an extremum of an objective function. This is foundational in the algorithms grounded on the variational quantum eigensolver, and is also a crucial ingredient in carrying out computation at the pulse level. The problem at hand is defined in terms of an objective function with the general form $F(x_1, \ldots, x_n)$, $F : \mathbb{R}^n \to \mathbb{R}$, such that the quantum processor is responsible for computing F for a certain set of parameters, and F is passed as the argument to a classical optimizer, which in turn checks for convergence and updates the parameters if necessary. To this end, XACC puts forward an `Optimizer` interface that exposes an `optimize()` method which takes as input a C++ function with signature `std::vector<double> x, std::vector<double>& gradx`. The `gradx` argument is only used in the event the optimization algorithm conducts parameter search guided by the gradients.

3.4 Observables, Algorithms, and Gradient Strategies

In order to describe high-level, domain specific algorithmic expressions we must provide data types enabling common quantum operator or observable creation and manipulation, as well as a unique API for initialization and execution of quantum-classical algorithms. First, XACC exposes the `Observable` interface, which is meant to provide an abstraction for quantum mechanical observable quantities. Ultimately, one usually expresses these observables in some basis and constructs new instances algebraically. XACC sub-types this interface for pauli or spin-based observable quantities (sums of pauli-tensor product terms) as well as fermionic second quantized representations. At the high level, `Observables` enable the observation of un-measured quantum circuits. By this we mean that `Observable` exposes an `observe` method that takes as input a `CompositeInstruction` without any concrete `Measure Instructions` and returns a vector of measured instances of the given `CompositeInstruction` dependent on the structure of the `Observable`. In this way, `Observables` with terms that involve an σ_x on a given qubit will result in a new `CompositeInstruction` with a σ_x basis measurement (a Hadamard followed by a Measure-Z). This functionality is leveraged heavily in variational quantum algorithms whereby we have some parameterized quantum circuit and we wish to execute it given some

concrete parameters for each term in the observable of interest. Because fermionic modes are not naturally implemented by quantum hardware, the necessary transformation is made available by the `ObservableTransform` class, which exposes the `transform()` method, an example of which is the Jordan-Wigner mapping.[16] Of note, `Observables` such as electronic Hamiltonians can be seamlessly generated by interfacing with quantum chemistry packages, where the specification for the quantum chemistry calculation, e.g., basis set, molecular geometry, active spaces, etc., are provided via the `HeterogeneousMap` with the corresponding keys. XACC currently offers interfaces with the Psi4[40] and PySCF[45] suites.

XACC provides an abstraction for algorithm expression, creation, and execution that enables one to contribute and inject new hybrid quantum-classical algorithms as plugin extensions to the underlying framework. The `Algorithm` interface exposes a mechanism for one-time initialization with pertinent algorithmic input parameters as well as a method for execution of the entire algorithm workflow. XACC exposes `Algorithm::initialize()` which takes as input a heterogeneous map of key-value pairs that seed the algorithm implementation with pertinent input parameters. For example, this may be the backend to target, the observable to compute expectation values for, and a parameterized `CompositeInstruction` detailing the ansatz circuit for a variational algorithm. Once initialized, programmers can affect execution of the quantum-classical algorithm via the `Algorithm::execute()` method, which takes as input an `AcceleratorBuffer` upon which execution results are persisted.
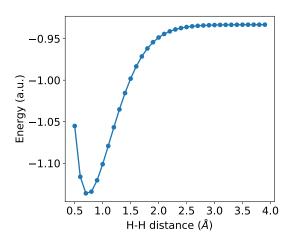

Fig. 2. Potential energy curve for H_2 in the STO-3G basis set from VQE simulations with the first-order trotter UCCSD ansatz obtained from the program in Figure 3.

As mentioned in Subsection 3.3, XACC offers the possibility of gradient-based optimization. In practical terms, this involves the execution of additional circuits since gradients are computed from varying the rotation angles in the parameterized gates. As such, these circuits can be obtained by calling `getGradientExecutions()` declared by the `AlgorithmGradientStrategy` interface whose arguments are the circuit instance (`CompositeInstruction`) and the current set of parameters x. Once these instructions are executed, the derivative with respect to each x is obtained by the `compute()` method. Currently, XACC enables gradient computation using numerical finite differences (backward, forward, and central differences), parameter shift,[26, 36] quantum natural gradient,[44] and automatic differentiation via the `AutoDiff` library.

4 XACC ALGORITHMS FOR QUANTUM CHEMISTRY

4.1 Variational Quantum Eigensolver

The variational quantum eigensolver (VQE)[33] has risen as the quintessential example of hybrid quantum-classical algorithm. The preparation and manipulation of highly entangled states, for instance, those that describe interacting fermions, such as in the electronic structure of molecules, pose a level of complexity that classical machines

have difficulty coping with. In practical terms, VQE casts the variational principle by targeting the ground state energy from a given Hamiltonian $\hat{H}$ upon preparing a trial state $|\Psi(\vec{\theta})\rangle$ via a circuit ansatz with variable parameters $\vec{\theta}$ in the form of single-qubit rotations, and these parameters are varied until optimal rotation angles are found.

This naturally translates into a concerted operation of quantum and classical computers, with the former preparing $|\Psi(\vec{\theta})\rangle$ and making the required measurements to estimate the value of the objective function, usually the expectation value of the Hamiltonian, for a given set of variational parameters $\vec{\theta}$. The classical computer receives and processes the outcomes of the measurements, which serve as input for an optimization routine that checks whether the parameters that minimize the objective function have been found.

A popular ansatz choice derives from the unitary coupled cluster theory, which is not feasible for direct implementation in quantum hardware and needs to be approximated, often to low-order in the Trotter-Suzuki approximation (also known as Trotterization).[14] We illustrate the usage of VQE in the case of the potential energy curve of H_2 with the STO-3G basis set where we prepare the ground state according to the first-order Trotterized UCCSD ansatz in Figure 3.

In this first example, it is beneficial to highlight and explain many of the APIs that will be recurrent in the demonstrations to follow. In lines 1-3 we include the necessary headers: xacc_service.hpp provides the getService<typename Interface>(serviceName) API to load service plugins and xacc_observable.hpp allowing creation of a general object of the Observable type. The former originates from the interface with PySCF, a Python native package, which are loaded and unloaded as shown in lines 8 and 50, respectively. Getting a reference to many of the most common interface types is facilitated by wrappers of the sort xacc::getInterface(serviceName, options),

```cpp
#include "xacc.hpp"                                        1
#include "xacc_service.hpp"                                2
#include "xacc_observable.hpp"                             3
using namespace xacc::quantum;                             4
int main(int argc, char **argv) {                          5
  // Initialize and load python plugins                    6
  xacc::Initialize(argc, argv);                            7
  xacc::external::load_external_language_plugins();        8
                                                           9
  // instantiate accelerator and optimizer                10
  auto accelerator = xacc::getAccelerator("qpp");          11
  auto optimizer = xacc::getOptimizer("nlopt",             12
                        {{"algorithm", "l-bfgs"}});        13
                                                           14
  // instantiate UCCSD ansatz                              15
  auto uccsd = xacc::createComposite("uccsd",              16
                        {{"ne", 2}, {"nq", 4}});           17
                                                           18
  // loop over H-H distances                               19
  for (double r = 0.5; r <= 4.0; r += 0.1) {               20
    // create hamiltonian                                  21
    auto geom = std::string("H 0.0 0.0 0.0\n"              22
                "H 0.0 0.0 " + std::to_string(r));         23
    auto hamiltonian = getObservable(                      24
        "pyscf", {{"basis", "sto-3g"},                     25
                  {"geometry", geom}});                    26
                                                           27
    // instantiate and initialize VQE                      28
    auto vqe = xacc::getAlgorithm("vqe",                   29
        {{"ansatz", uccsd},                                30
        {"optimizer", optimizer},                          31
        {"observable", hamiltonian},                       32
        {"accelerator", accelerator}                       33
        {"gradient_strategy", "parameter-shift"}});        34
                                                           35
    // allocate buffer and execute                         36
    auto buffer = xacc::qalloc(4);                         37
    vqe->execute(buffer);                                  38
                                                           39
    // retrieve energy from buffer and print               40
    auto energy = (*buffer)["opt-val"].as<double>()         41
    std::cout << energy << "\n";                            42
  }                                                         43
                                                           44
  xacc::external::unload_external_language_plugins();      45
  xacc::Finalize();                                        46
  return 0;                                                47
}                                                          48
```

Fig. 3. C++ program to compute the potential energy curve of H_2 in the STO-3G basis using the VQE algorithm with the first-order trotterized UCCSD ansatz.

which are illustrated in lines 11 (Accelerator), 12-13 (Optimizer), 24-26 (Observable), and 29-34 (Algorithm), where options is a HeterogenousMap with the parameters

specific to the service in question. In this particular example, we use the virtual backend accelerator building on the Quantum++[12] library (line 11), the L-BFGS optimization algorithm in NLOpt (line 14), and the first-order trotterized UCCSD ansatz, whose instantiation is shown in lines 18-19, with the keys "ne" and "nq" taking as values the number of electrons and number of qubits, respectively. Within the `for` loop enclosed in lines 22-47, the geometry of the H_2 molecule and the corresponding `Observable` is updated in lines 25-29. With all the necessary parameters, we get a reference to the VQE algorithm, initialized with all the previously instantiated parameters as well as a `string` that identifies the desired gradient strategy, which in this case is parameter shift.

The energy is persisted in the buffer as a `double` and can be retrieved with the proper key ("opt-val") and knowledge of the associated type. The resulting potential energy curve is plotted in Figure 2.

4.2　ADAPT-VQE

```
1   // instantiate accelerator, optimizer
2   // and allocate buffer
3
4   // instantiate observable in an active space
5   std::vector<int> frozen = {0,6},
6       active = {1, 2, 3, 4, 5, 7, 8, 9, 10, 11};
7   auto hamiltonian =
8       xacc::quantum::getObservable("pyscf",
9       {{"basis", "sto-6g"},
10      {"geometry", geom},
11      {"frozen-spin-orbitals", frozen},
12      {"active-spin-orbitals", active}});
13
14  // instantiate and initialize ADAPT-VQE
15  auto adapt = xacc::getAlgorithm("adapt",
16                  {{"optimizer", optimizer},
17                  {"observable", hamiltonian},
18                  {"sub-algorithm", "vqe"},
19                  {"n-electrons", 4},
20                  {"pool", "singlet-adapted-uccsd"},
21                  {"accelerator", accelerator}});
22
23  // execute
24  adapt->execute(buffer);
```
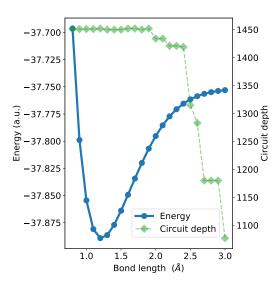

Fig. 4. Code snippet highlighting the ADAPT-VQE code initialization and execution for H_4 with the singlet-adapted single and double excitation operators (left) and the potential energy curve of CH^+ in the STO-6G with the lowest energy spatial orbital frozen. For comparison, the circuit depth of the first-order trotterized UCCSD ansatz for the same molecule is 11665.

Despite being able to execute certain tasks exponentially faster than its classical counterparts, quantum computers are at a stage that there is only a limited time window in which this can be done before the information stored in the qubits is lost due to external effects, collectively referred to as "noise" . This time window is known as the coherence time and its narrow width is a major impediment in the maturing of quantum information technologies as a whole. In practical terms, this caps the so-called circuit depth, which correlates to the amount of successive operations that can be performed by a quantum circuit. In an effort to work around this limitation, the Adaptive Derivative-Assembled Pseudo-Trotter (ADAPT)[13, 46, 48] ansatz replaces the alternatives based on the straightforward application of the UCC by capitalizing on the Trotter-Suzuki formula. An ADAPT-VQE simulation of the potential energy curve of CH^+ ion in the singlet electronic state can be set up with the code snippet in Figure 4, along with the output energy values and circuit depths for different bond lengths.

The core assumption in the ADAPT-VQE algorithm is that an easy-to-prepare state, normally a mean-field approximation such as Hartree-Fock, can be connected to the ground state with degree of accuracy by only a limited number of many-body rotations drawn out of an "operator pool". With the example of direct application of the UCCSD ansatz in mind (Subsection 4.1), it aims at retaining only the operators that provide a sizable ground state energy improvement, emphasizing double excitations over singles. At each ADAPT iteration, the gradient with respect to the operator pool, which equals the commutators of the Hamiltonian with the operators in the pool, have to be measured, and the norm of this gradient controls the number of iterations the ADAPT algorithm goes through. If this norm is not found below a user-defined threshold, the operator displaying the largest commutator (in absolute terms) is introduced in the ansatz, whose parameters are optimized by a usual VQE call. This can greatly reduce circuit depth by only retaining the many-body rotations required to reach the ground state energy (to some precision). On the other hand, the relative importance of each operator as determined by their commutators needs to be re-evaluated at each iteration, shifting the demand for quantum resources to the measurement task.

4.3 Quantum imaginary time evolution

The two algorithms above highlight one frequent impasse in the choice of a variational quantum algorithm, which is the trade-off between circuit depth and number of measurements. An alternative to methods

```
// Hamiltonian                                              1
using namespace xacc::quantum;                              2
auto H = PauliOperator(0.2976);                             3
H += PauliOperator({{0, "Z"}}, 0.3593);                     4
H += PauliOperator({{1, "Z"}}, -0.4826);                    5
H += PauliOperator({{0, "Z"}, {1, "Z"}}, 0.5818);          6
H += PauliOperator({{0, "X"}, {1, "X"}}, 0.0896);          7
H += PauliOperator({{0, "Y"}, {1, "Y"}}, 0.0896);          8
Observable* observable = &H;                                9
                                                           10
// Get reference to the TNQVM Accelerator                  11
auto accelerator = xacc::getAccelerator("tnqvm");          12
                                                           13
// Map quantum kernel code into XACC IR                    14
// and compile to accelerator                              15
auto compiler = xacc::getCompiler("xasm");                 16
auto ansatz = compiler->compile(R"(                        17
__qpu__ void kernel(qbit q) {                              18
  X(q[0]);                                                 19
})", accelerator)->getComposite("kernel");                 20
                                                           21
// Time propagation parameters                             22
int nSteps = 100;                                          23
int stepSize = 0.05;                                       24
                                                           25
// Initialize QITE                                         26
auto qite = xacc::getAlgorithm("qite",                     27
                    {{"accelerator", accelerator},         28
                     {"observable", observable},           29
                     {"step-size", stepSize},              30
                     {"steps", nSteps},                    31
                     {"ansatz", ansatz}});                 32
                                                           33
// Allocate buffer and execute                             34
auto buffer = xacc::qalloc(2);                             35
qite->execute(buffer);                                     36
```

Fig. 5. Code snippet for the QITE simulation of H_2 at H-H distance of 0.7Å with the Hamiltonian from Ref.[29].

ground on the variational principle is found in the family of approaches stemming from imaginary time evolution.[21, 27] It has the benefit of targeting eigenstates as well as thermal states on an equal footing in an efficient fashion as long as the time propagation as done slowly or, in other words, the propagated terms remain relatively local.

The code snippet provided in Figure 5 can be used to reproduce the plot in Figure 6. This is a simple use case of the QITE algorithm that displays the convergence of the ground state energy throughout propagation of the H_2 Hamiltonian in the STO-6G in imaginary time, which is shown in Figure 6. We also note that the closely quantum analog of the Lanczos method to target specific eigenvalues is also available in XACC. Here we present yet another way to construct compound observables, showcasing some of the operator algebra underlying the `PauliOperator` type in lines 1-7 in Figure 5, whose overloaded constructor enables several variants of Pauli operators to be accessed. We also show an

alternative way to get the `CompositeInstruction` object that represents the circuit to be employed in the simulation. Upon creating instances for the accelerator and compiler, a string literal representing the quantum kernel to be compiled is passed to the `xacc::Compiler::compile()` method, which maps the string into an instance of XACC IR and further process it if necessary before compilation to the accelerator, with the resulting `CompositeInstruction` being retrieved by calling `getComposite()` such that the argument matches the name of the quantum kernel ("kernel" here). In Figure 6 we can see how the ground state energy of H_2 in the STO-6G evolves in imaginary time by propagating the HF state through 100 imaginary time steps of 0.05 units of time each.

4.4 Quantum moments expansions

Methods grounded on expansions of the moments of the Hamiltonian have a long history that dates back to the "t-expansion" in terms of connected moments.[15] While it provides the exact ground state energy in the limit $t \to \infty$, several approximations have been proposed over the years in order to establish energy estimates for finite t.[9, 17, 31, 41] Several connections between moments expansions, power methods,[37] and imaginary time evolution can be drawn, each also displaying their particularities. At the heart of this family of approaches lies the computation of matrix elements of the type $\langle H^k \rangle$, which can be in turn employed in the computation of connected moments appearing in some of these methodologies. Computation of such quantities can expected to be performed efficiently by quantum computers, thus opening the door to quantum variants of moments expansions.[10, 19, 32]
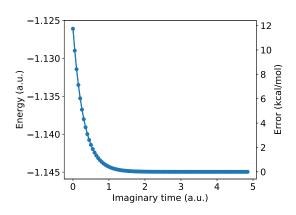

Fig. 6. Ground state energy H_2 in the STO-6G basis set throughout imaginary time evolution.

We show in Figure 7 a sample program to perform such simulations.

Considering that we map each spin-orbital in the Hamiltonian to a qubit, the number of orbitals can be simply retrieved as the number of bits in the observable, as shown in line 2 of the snippet in Figure 7. Taking the square H_4 molecule in the STO-6G basis set where the distance between adjacent hydrogen atoms is 2Å, we end up with 8 spin-orbitals. The circuit to prepare the Hartree–Fock (HF) state is obtained following lines 5-12. In XACC we follow the convention that all alpha spin-orbitals are mapped to the first half of the qubits in the register, followed by the beta spin-orbitals, thus resulting in $|11001100\rangle$ as the computational basis state representation of the HF state. The QCMX algorithm computes the energy of all orders from 2 up to `cmx_order` (line 18) using three different expansion types: Cioslowski's connected moments expansion (CMX), the Peeters–Devreese–Soldatov (PDS) energy functional, and the Knowles' generalized Padé approximants, while many others can be implemented. Going up to order 8 for the H_4 molecule above, we obtain the errors from the exact ground state energy reported in Figure 7.

4.5 Quantum equation of motion

A great deal of effort in quantum chemistry revolves around finding suitable estimates for the ground state and its energy. For this reason, VQE has been quite popular for chemistry simulations since its inception due to its foundation on the variational principle. It has been shown it can be used in the search for excited states, yet for this application

```
1   // instantiate accelerator, optimizer, and observable
2   auto nOrbitals = observable->nBits();
3
4   // instantiate ansatz that prepares the HF state
5   auto irgen =
6       xacc::getService<xacc::IRProvider>("quantum");
7   auto ansatz = irgen->createComposite("initial-state");
8   for (auto i = 0; i < nElectrons / 2; i++) {
9     auto alphaX = irgen->createInstruction("X", {i});
10    ansatz->addInstruction(alphaX);
11    auto betaX = irgen->createInstruction("X",
12                          {i + nOrbitals / 2});
13    ansatz->addInstruction(betaX);
14  }
15
16  // instantiate and initialize QCMX for some cmx_order
17  auto qcmx = xacc::getAlgorithm("qcmx", {
18                          {"ansatz", ansatz},
19                          {"accelerator", accelerator},
20                          {"observable", observable},
21                          {"cmx-order", cmx_order}
22                          });
23
24  // allocate buffer and execute
```
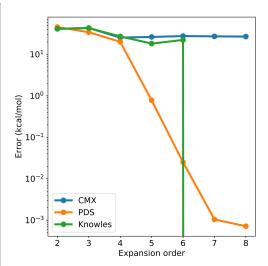

Fig. 7. Code snippet highlighting the ADAPT-VQE code initialization and execution for H_4 with the singlet-adapted single and double excitation operators (left) and the potential energy curve of CH^+ in the STO-6G with the lowest energy spatial orbital frozen. For comparison, the circuit depth of the first-order trotterized UCCSD ansatz for the same molecule is 11665.

it requires certain deviations from its original formulation, such as the introduction of penalty terms to prevent the solver to enter the symmetry subspace where the ground state solution can be found.[20] Borrowing from the equation-of-motion (EOM) formalism and its classical implementations,[38, 42] the quantum version of the EOM method (QEOM)[28] yields the spectrum of electronic excitations and de-excitations solving a secular system equations in the basis of the excitations operators that span the targeted excitation subspace where the desired state manifold can be found. Once the required matrix elements are computed from a provided approximation to the ground state, the energy spectrum follows classically from the solution of a generalized eigenvalue problem. The preparation of the input ground state precedes the QEOM algorithm and is largely detached from it, so there is great freedom in choosing how to approximate the ground state, e.g., VQE.

An example showing the parameters that are required by the QEOM algorithm is displayed in Figure 8. Preparing the approximate ground state of the H_2 molecule in the 6-31G basis set with 4 ADAPT-VQE iterations and simulating the electronic transitions using the QEOM algorithm results in the potential energy curves plotted in Figure 9.

4.6 MC-VQE

Some properties exhibited by electrons make them particularly demanding in their quantum-mechanical treatment, such as the antisymmetry of the wavefunction under exchange of indistinguishable fermionic

```
1   // instantiate accelerator, ansatz, and observable
2   // get number of electrons (nElectrons)
3   auto qeom = xacc::getAlgorithm("qeom",
4                          {{"ansatz", ansatz}
5                          {"accelerator", accelerator},
6                          {"observable", observable},
7                          {"n-electrons", nElectrons}});
8   auto q = xacc::qalloc(observable->nBits());
9   qeom->execute(q);
```

Fig. 8. Code snippet showing the required parameters for the QEOM algorithm.
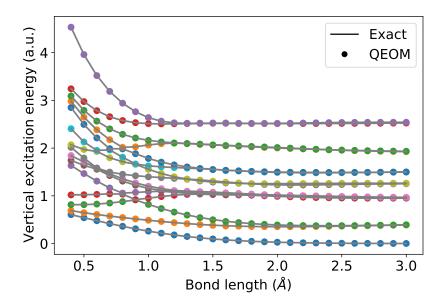
Fig. 9. Potential energy curves for the electronic excited states of the H_2 molecule in the 6-31G basis set (particle-number conserving subspace).

modes and the long-range Coulomb interaction between any two electrons. However, in the limit where these electrons are fairly constrained to disjoint spatial regions, these requirements can be relaxed to a good approximation. This is at the heart of the ab initio exciton model (AIEM) [39] which offers a possibility of modeling excited states in large molecular systems formed by weakly coupled units, or monomers, for example in biological chromophore complexes. The multi-contracted VQE (MC-VQE)[30] quantum algorithm casts the AIEM Hamiltonian into the basis of Pauli operators and, by restricting the number of states in each monomer to their ground and first excited states, enables the collective absorption spectrum to be obtained via two-qubit gates entangling qubits corresponding to isolated monomers. It deviates from the usual VQE due to 1) minimizing the state-averaged energy while all states share the same entanglers, and 2) the final spectrum comes from a classical diagonalization in the so-called "interference" state basis.

In the MC-VQE algorithm, the ground and first excited state of each chromophore map onto the states of a dedicated qubit, so the number of chromophores (line 3) determines the size of the qubit register. Besides some of the recurrent `HeterogeneousMap` keys, the MC-VQE takes the `cyclic` key to inform if the chromophore connectivity is cyclic or linear, which in turn translates into whether or not the qubits indexed by `0` and `nQubits - 1` should be entangled. The value associated to the `data-path` is the directory where the classical data (electronic energies, dipole moments, etc.) is located, which are processed with the help of an `Importable` helper class. For example, we can simulate a 4-chromophore segment of the LH2 B850 ring complex, with a sample multi-contracted state preparation circuit being shown in Figure 10a). Figure **??**b) highlights the code pertinent to the MC-VQE algorithm (classical quantities needed to compute the AIEM Hamiltonian are obtained from the Supplemental Material in Ref. [?]).The resulting absorption

a)

b)

```
1   // get reference to Accelerator and Optimizer
2   // define data_path
3   auto nQubits = 4;
4   auto isCyclic = false;
5   auto mc_vqe = xacc::getAlgorithm("mc-vqe");
6   mc_vqe->initialize({{"cyclic", isCyclic},
7                       {"accelerator", accelerator},
8                       {"optimizer", optimizer},
9                       {"data-path", data_path},
10                      {"nChromophores", nQubits}});
11
12  // allocate buffer and execute
13  auto buffer = xacc::qalloc(nQubits);
14  mc_vqe->execute(buffer);
```
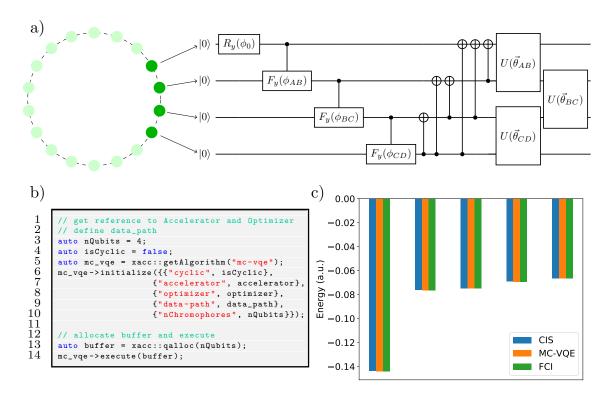
c)

Fig. 10. a) Pictorial representation of the LH2 B850 ring complex and the selected segment with the associated MC-VQE state preparation circuit; b) Code snippet illustrating MC-VQE; c) Comparison of the resulting spectrum among configuration interaction singles, MC-VQE, and full configuration interaction.

spectrum, compared with the configuration interaction singles (CIS) and exact diagonalization (full configuration interaction, FCI) results are displayed in Figure 10c).

4.7 Comparison with Python-based framework

The fact that the vast majority of virtual backends (numerical simulators) are written in low-level languages attests to the necessity of a high degree of resource control that they enable. Yet, most software stacks for chemistry simulations in quantum processors, or quantum computation more generally, are devised in high-level languages, with special focus on Python. Despite the notion that the backend is assinged the most arduous tasks, there are still benefits from adopting more perfomant languages throughout the many stages along the workflow. To corroborate this claim, we take IBM's Qiskit [3] to be representative of a Python-based framework and compared it against XACC in that time they take to construct the circuit that prepares the state associated with the first-order trotterized UCCSD ansatz for several combinations of qubit registers and number of electrons. The best timings out of 10 runs are reported in Figure 11.

The number of operators/gates in the UCCSD ansatz is dominated by double excitations. For N_q the number of qubits and N_e the number of electrons, the total number of double excitations is

$$\binom{N_q/2}{N_e}\binom{N_q/2}{N_e} = \binom{N_q/2}{N_q/2 - N_e}\binom{N_q/2}{N_q/2 - N_e}$$
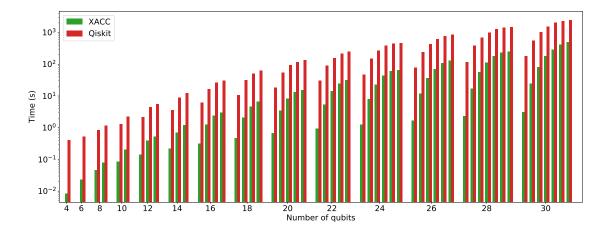
Fig. 11. Timings for the construction of the first-order trotterized UCCSD circuit for XACC and Qiskit.

thus we only report the timings for $N_e < N_q/2$ for a given N_q in Figure 11. Here again we can see additional evidence for harnessing the power of the more robust C++ and the XACC IR it enables, since it leads to circuit constructions that are at least an order of magnitude faster than the analogous process with Python as performed by Qiskit.

5   SUMMARY AND OUTLOOK

We have presented a collection of state-of-the-art algorithms for quantum chemistry leveraging quantum co-processors, as well as their implementation and usage within the XACC framework, thereby allowing users to immediately have access to algorithms covering a wide variety of chemical phenomena. Furthermore, it is important to emphasize that the core functionality of XACC is largely exposed through an extensive list of Python bindings, facilitating utilization by users more familiar with such a language.

Even more importantly, with the preceding discussion of the main XACC interfaces and its comprehensive list of APIs, we hope to have conveyed that XACC stands out for its ability and flexibility in enabling development with the goal of high-performance and potential deployment across multiple quantum hardware platforms and that these features can be exploited in devising, implementing, and benchmarking quantum algorithms for chemistry, all the while fostering an open-source environment. In passing, it is also noteworthy that many other domain sciences with a common ground in quantum mechanics can equally benefit from the infrastructure made available by XACC, such as quantum field theories, solid state physics, quantum dynamics, etc.

The ecosystem of software tools for quantum computation targeting chemical applications has been largely influenced by Pythonic frameworks, with some of the notable examples being Qiskit[3], OpenFermion[25], PennyLane[5], and TEQUILA[18]. With the expected increase of orders of magnitude in the number of qubits envisioned for the next few years and the applications that they will likely enable comes an ever growing need for a robust and lasting software infrastructure, which strengthens the case for system-level, resource-efficient capabilities of C++. Moreover, in order to delineate the regime of the quantum supremacy/advantage, analogous investigation exhausting classical resources are called for and can be performed in a seamless fashion within a C++ framework, while posing many technical obstacles to a high-level language such as Python. With these considerations as a key component governing its design and by

delivering a framework that spans the entire quantum computation workflow in a highly extensible fashion, XACC posits itself in the forefront of quantum software for chemical applications, for both the current hybrid quantum-classical regime as well as future fault tolerant machines.

6 ACKNOWLEDGEMENTS

REFERENCES

[1] [n.d.]. C++ Micro Services. http://cppmicroservices.org/. Accessed: 2021-04-29.

[2] [n.d.]. Open Services Gateway Initiative. https://www.osgi.org/. Accessed: 2021-04-29.

[3] Héctor Abraham et al. 2019. Qiskit: An Open-source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2562110

[4] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Gidney, Marissa Giustina, Rob Graff, Keith Guerin, Steve Habegger, Matthew P. Harrigan, Michael J. Hartmann, Alan Ho, Markus Hoffmann, Trent Huang, Travis S. Humble, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Paul V. Klimov, Sergey Knysh, Alexander Korotkov, Fedor Kostritsa, David Landhuis, Mike Lindmark, Erik Lucero, Dmitry Lyakh, Salvatore Mandrà, Jarrod R. McClean, Matthew McEwen, Anthony Megrant, Xiao Mi, Kristel Michielsen, Masoud Mohseni, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles Neill, Murphy Yuezhen Niu, Eric Ostby, Andre Petukhov, John C. Platt, Chris Quintana, Eleanor G. Rieffel, Pedram Roushan, Nicholas C. Rubin, Daniel Sank, Kevin J. Satzinger, Vadim Smelyanskiy, Kevin J. Sung, Matthew D. Trevithick, Amit Vainsencher, Benjamin Villalonga, Theodore White, Z. Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven, and John M. Martinis. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (Oct. 2019), 505–510. https://doi.org/10.1038/s41586-019-1666-5

[5] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, M. Sohaib Alam, Shahnawaz Ahmed, Juan Miguel Arrazola, Carsten Blank, Alain Delgado, Soran Jahangiri, Keri McKiernan, Johannes Jakob Meyer, Zeyue Niu, Antal Száva, and Nathan Killoran. 2020. PennyLane: Automatic differentiation of hybrid quantum-classical computations. arXiv:1811.04968 [quant-ph]

[6] Claudio N. Cavasotto, Natalia S. Adler, and Maria G. Aucar. 2018. Quantum Chemical Approaches in Structure-Based Virtual Screening and Lead Optimization. *Frontiers in Chemistry* 6 (2018), 188. https://doi.org/10.3389/fchem.2018.00188

[7] Claudio N. Cavasotto, María Gabriela Aucar, and Natalia S. Adler. 2019. Computational chemistry in drug lead discovery and design. *International Journal of Quantum Chemistry* 119, 2 (2019), e25678. https://doi.org/10.1002/qua.25678 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/qua.25678

[8] Kamal Choudhary, Marnik Bercx, Jie Jiang, Ruth Pachter, Dirk Lamoen, and Francesca Tavazza. 2019. Accelerated Discovery of Efficient Solar Cell Materials Using Quantum and Machine-Learning Methods. *Chemistry of Materials* 31, 15 (2019), 5900–5908. https://doi.org/10.1021/acs.chemmater.9b02166 arXiv:https://doi.org/10.1021/acs.chemmater.9b02166

[9] J Cioslowski. 1987. Connected moments expansion: a new tool for quantum many-body theory. *Phys. Rev. Lett.* 58, 2 (1987), 83.

[10] Daniel Claudino, Bo Peng, Nicholas P. Bauman, Karol Kowalski, and Travis S. Humble. 2021. Improving the accuracy and efficiency of quantum connected moments expansions. arXiv:2103.09124 [quant-ph]

[11] Yongjie Cui, Peipei Zhu, Xunfan Liao, and Yiwang Chen. 2020. Recent advances of computational chemistry in organic solar cell research. *J. Mater. Chem. C* 8 (2020), 15920–15939. Issue 45. https://doi.org/10.1039/D0TC03709E

[12] Vlad Gheorghiu. 2018. Quantum++: A modern C++ quantum computing library. *PLOS ONE* 13, 12 (12 2018), 1–27. https://doi.org/10.1371/journal.pone.0208073

[13] Harper R. Grimsley, Sophia E. Economou, Edwin Barnes, and Nicholas J. Mayhall. 2019. An adaptive variational algorithm for exact molecular simulations on a quantum computer. *Nature Communications* 10, 1 (08 Jul 2019), 3007. https://doi.org/10.1038/s41467-019-10988-2

[14] Naomichi Hatano and Masuo Suzuki. 2005. *Finding Exponential Product Formulas of Higher Orders*. Springer Berlin Heidelberg, Berlin, Heidelberg, 37–68. https://doi.org/10.1007/11526216_2

[15] David Horn and Marvin Weinstein. 1984. The t expansion: a nonperturbative analytic tool for Hamiltonian systems. *Phys. Rev. D* 30, 6 (1984), 1256.

[16] P. Jordan and E. Wigner. 1928. Über das Paulische Äquivalenzverbot. *Zeitschrift für Physik* 47, 9-10 (Sept. 1928), 631–651. https://doi.org/10.1007/bf01331938

[17] Peter J Knowles. 1987. On the validity and applicability of the connected moments expansion. *Chem. Phys. Lett.* 134, 6 (1987), 512–518.

[18] Jakob S Kottmann, Sumner Alperin-Lea, Teresa Tamayo-Mendoza, Alba Cervera-Lierta, Cyrille Lavigne, Tzu-Ching Yen, Vladyslav Verteletskyi, Philipp Schleich, Abhinav Anand, Matthias Degroote, Skylar Chaney, Maha Kesibi, Naomi Grace Curnow, Brandon Solo, Georgios Tsilimigkounakis, Claudia Zendejas-Morales, Artur F Izmaylov, and Alán Aspuru-Guzik. 2021. TEQUILA: a platform for rapid development of quantum algorithms. *Quantum Science and Technology* 6, 2 (March 2021), 024009. https://doi.org/10.1088/2058-9565/abe567

[19] Karol Kowalski and Bo Peng. 2020. Quantum simulations employing connected moments expansions. *The Journal of Chemical Physics* 153, 20 (Nov. 2020), 201102. https://doi.org/10.1063/5.0030688

[20] Kohdai Kuroiwa and Yuya O. Nakagawa. 2021. Penalty methods for a variational quantum eigensolver. *Phys. Rev. Research* 3 (Feb 2021), 013197. Issue 1. https://doi.org/10.1103/PhysRevResearch.3.013197

[21] Sam McArdle, Tyson Jones, Suguru Endo, Ying Li, Simon C. Benjamin, and Xiao Yuan. 2019. Variational ansatz-based quantum simulation of imaginary time evolution. *npj Quantum Information* 5, 1 (Sept. 2019). https://doi.org/10.1038/s41534-019-0187-2

[22] A.J. McCaskey, E. Dumitrescu, M. Chen, D. Lyakh, and T. Humble. 2018. Validating quantum-classical programming models with tensor network simulations. *PLOS ONE* 13, 12 (12 2018), 1–19. https://doi.org/10.1371/journal.pone.0206704

[23] A.J. McCaskey, E.F. Dumitrescu, D. Liakh, M. Chen, W. Feng, and T.S. Humble. 2018. A language and hardware independent approach to quantum–classical computing. *SoftwareX* 7 (2018), 245 – 254. https://doi.org/10.1016/j.softx.2018.07.007

[24] A.J. McCaskey, D.I. Lyakh, E.F. Dumitrescu, S.S. Powers, and T.S. Humble. 2020. XACC: a system-level software infrastructure for heterogeneous quantum–classical computing. *Quantum Science and Technology* 5, 2 (feb 2020), 024002. https://doi.org/10.1088/2058-9565/ab6bf6

[25] Jarrod R McClean, Nicholas C Rubin, Kevin J Sung, Ian D Kivlichan, Xavier Bonet-Monroig, Yudong Cao, Chengyu Dai, E Schuyler Fried, Craig Gidney, Brendan Gimby, Pranav Gokhale, Thomas Häner, Tarini Hardikar, Vojtěch Havlíček, Oscar Higgott, Cupjin Huang, Josh Izaac, Zhang Jiang, Xinle Liu, Sam McArdle, Matthew Neeley, Thomas O'Brien, Bryan O'Gorman, Isil Ozfidan, Maxwell D Radin, Jhonathan Romero, Nicolas P D Sawaya, Bruno Senjean, Kanav Setia, Sukin Sim, Damian S Steiger, Mark Steudtner, Qiming Sun, Wei Sun, Daochen Wang, Fang Zhang, and Ryan Babbush. 2020. OpenFermion: the electronic structure package for quantum computers. *Quantum Science and Technology* 5, 3 (June 2020), 034014. https://doi.org/10.1088/2058-9565/ab8ebc

[26] K. Mitarai, M. Negoro, M. Kitagawa, and K. Fujii. 2018. Quantum circuit learning. *Phys. Rev. A* 98 (Sep 2018), 032309. Issue 3. https://doi.org/10.1103/PhysRevA.98.032309

[27] Mario Motta, Chong Sun, Adrian TK Tan, Matthew J O'Rourke, Erika Ye, Austin J Minnich, Fernando GSL Brandão, and Garnet Kin-Lic Chan. 2019. Determining eigenstates and thermal states on a quantum computer using quantum imaginary time evolution. *Nature Physics* (2019), 1–6.

[28] Pauline J. Ollitrault, Abhinav Kandala, Chun-Fu Chen, Panagiotis Kl. Barkoutsos, Antonio Mezzacapo, Marco Pistoia, Sarah Sheldon, Stefan Woerner, Jay M. Gambetta, and Ivano Tavernelli. 2020. Quantum equation of motion for computing molecular excitation energies on a noisy quantum processor. *Phys. Rev. Research* 2 (Oct 2020), 043140. Issue 4. https://doi.org/10.1103/PhysRevResearch.2.043140

[29] P. J. J. O'Malley, R. Babbush, I. D. Kivlichan, J. Romero, J. R. McClean, R. Barends, J. Kelly, P. Roushan, A. Tranter, N. Ding, B. Campbell, Y. Chen, Z. Chen, B. Chiaro, A. Dunsworth, A. G. Fowler, E. Jeffrey, E. Lucero, A. Megrant, J. Y. Mutus, M. Neeley, C. Neill, C. Quintana, D. Sank, A. Vainsencher, J. Wenner, T. C. White, P. V. Coveney, P. J. Love, H. Neven, A. Aspuru-Guzik, and J. M. Martinis. 2016. Scalable Quantum Simulation of Molecular Energies. *Phys. Rev. X* 6 (Jul 2016), 031007. Issue 3. https://doi.org/10.1103/PhysRevX.6.031007

[30] Robert M. Parrish, Edward G. Hohenstein, Peter L. McMahon, and Todd J. Martínez. 2019. Quantum Computation of Electronic Transitions Using a Variational Quantum Eigensolver. *Phys. Rev. Lett.* 122 (Jun 2019), 230401. Issue 23. https://doi.org/10.1103/PhysRevLett.122.230401

[31] FM Peeters and JT Devreese. 1984. Upper bounds for the free energy. A generalisation of the Bogolubov inequality and the Feynman inequality. *J. Phys. A: Mathematical and General* 17, 3 (1984), 625.

[32] Bo Peng and Karol Kowalski. 2021. Variational quantum solver employing the PDS energy functional. arXiv:2101.08526 [quant-ph]

[33] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'brien. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nat. Commun.* 5 (2014), 4213.

[34] Matthew G. Quesne, Fabrizio Silveri, Nora H. de Leeuw, and C. Richard A. Catlow. 2019. Advances in Sustainable Catalysis: A Computational Perspective. *Frontiers in Chemistry* 7 (2019), 182. https://doi.org/10.3389/fchem.2019.00182

[35] Kaushik Raha, Martin B. Peters, Bing Wang, Ning Yu, Andrew M. Wollacott, Lance M. Westerhoff, and Kenneth M. Merz. 2007. The role of quantum mechanics in structure-based drug design. *Drug Discovery Today* 12, 17 (2007), 725–731. https://doi.org/10.1016/j.drudis.2007.07.006

[36] Maria Schuld, Ville Bergholm, Christian Gogolin, Josh Izaac, and Nathan Killoran. 2019. Evaluating analytic gradients on quantum hardware. *Phys. Rev. A* 99 (Mar 2019), 032331. Issue 3. https://doi.org/10.1103/PhysRevA.99.032331

[37] Kazuhiro Seki and Seiji Yunoki. 2020. Quantum power method by a superposition of time-evolved states. arXiv:2008.03661 [quant-ph]

[38] Hideo Sekino and Rodney J. Bartlett. 1984. A linear response, coupled-cluster theory for excitation energy. *International Journal of Quantum Chemistry* 26, S18 (1984), 255–265. https://doi.org/10.1002/qua.560260826 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/qua.560260826

[39] Aaron Sisto, David R. Glowacki, and Todd J. Martinez. 2014. Ab Initio Nonadiabatic Dynamics of Multichromophore Complexes: A Scalable Graphical-Processing-Unit-Accelerated Exciton Framework. *Accounts of Chemical Research* 47, 9 (2014), 2857–2866. https://doi.org/10.1021/ar500229p arXiv:https://doi.org/10.1021/ar500229p PMID: 25186064.

[40] Daniel G. A. Smith, Lori A. Burns, Andrew C. Simmonett, Robert M. Parrish, Matthew C. Schieber, Raimondas Galvelis, Peter Kraus, Holger Kruse, Roberto Di Remigio, Asem Alenaizan, Andrew M. James, Susi Lehtola, Jonathon P. Misiewicz, Maximilian Scheurer, Robert A. Shaw, Jeffrey B. Schriber, Yi Xie, Zachary L. Glick, Dominic A. Sirianni, Joseph Senan O'Brien, Jonathan M. Waldrop, Ashutosh Kumar, Edward G. Hohenstein, Benjamin P. Pritchard, Bernard R. Brooks, Henry F. Schaefer, Alexander Yu. Sokolov, Konrad Patkowski, A. Eugene DePrince, Uğur Bozkaya, Rollin A. King, Francesco A. Evangelista, Justin M. Turney, T. Daniel Crawford, and C. David Sherrill. 2020. PSI4 1.4: Open-source software for high-throughput quantum chemistry. *The Journal of Chemical Physics* 152, 18 (2020), 184108. https://doi.org/10.1063/5.0006002 arXiv:https://doi.org/10.1063/5.0006002

[41] AV Soldatov. 1995. Generalized variational principle in quantum mechanics. *Int. J. Mod. Phys. B* 9, 22 (1995), 2899–2936.

[42] John F. Stanton and Rodney J. Bartlett. 1993. The equation of motion coupled–cluster method. A systematic biorthogonal approach to molecular excitation energies, transition probabilities, and excited state properties. *The Journal of Chemical Physics* 98, 9 (1993), 7029–7039. https://doi.org/10.1063/1.464746 arXiv:https://doi.org/10.1063/1.464746

[43] Jonathan Stevens. 2017. Virtually going green: The role of quantum computational chemistry in reducing pollution and toxicity in chemistry. *Physical Sciences Reviews* 2, 7 (2017). https://doi.org/doi:10.1515/psr-2017-0005

[44] James Stokes, Josh Izaac, Nathan Killoran, and Giuseppe Carleo. 2020. Quantum Natural Gradient. *Quantum* 4 (May 2020), 269. https://doi.org/10.22331/q-2020-05-25-269

[45] Qiming Sun, Timothy C. Berkelbach, Nick S. Blunt, George H. Booth, Sheng Guo, Zhendong Li, Junzi Liu, James D. McClain, Elvira R. Sayfutyarova, Sandeep Sharma, Sebastian Wouters, and Garnet Kin-Lic Chan. 2017. PySCF: the Python–based simulations of chemistry framework. , e1340 pages. https://doi.org/10.1002/wcms.1340 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/wcms.1340

[46] Ho Lun Tang, V. O. Shkolnikov, George S. Barron, Harper R. Grimsley, Nicholas J. Mayhall, Edwin Barnes, and Sophia E. Economou. 2020. qubit-ADAPT-VQE: An adaptive algorithm for constructing hardware-efficient ansatze on a quantum processor. arXiv:1911.10205 [quant-ph]

[47] Han-Sen Zhong, Hui Wang, Yu-Hao Deng, Ming-Cheng Chen, Li-Chao Peng, Yi-Han Luo, Jian Qin, Dian Wu, Xing Ding, Yi Hu, Peng Hu, Xiao-Yan Yang, Wei-Jun Zhang, Hao Li, Yuxuan Li, Xiao Jiang, Lin Gan, Guangwen Yang, Lixing You, Zhen Wang, Li Li, Nai-Le Liu, Chao-Yang Lu, and Jian-Wei Pan. 2020. Quantum computational advantage using photons. *Science* 370, 6523 (2020), 1460–1463. https://doi.org/10.1126/science.abe8770 arXiv:https://science.sciencemag.org/content/370/6523/1460.full.pdf

[48] Linghua Zhu, Ho Lun Tang, George S. Barron, Nicholas J. Mayhall, Edwin Barnes, and Sophia E. Economou. 2020. An adaptive quantum approximate optimization algorithm for solving combinatorial problems on a quantum computer. arXiv:2005.10258 [quant-ph]