Extending XACC for Quantum Optimal Control

Thien Nguyen, Anthony Santana, Alexander McCaskey
Computer Science and Mathematics
Oak Ridge National Laboratory
Oak Ridge, USA
nguyentm@ornl.gov

***Abstract*—Quantum computing vendors are beginning to open up application programming interfaces for direct pulse-level quantum control. With this, programmers can begin to describe quantum kernels of execution via sequences of arbitrary pulse shapes. This opens new avenues of research and development with regards to smart quantum compilation routines that enable direct translation of higher-level digital assembly representations to these native pulse instructions. In this work, we present an extension to the XACC system-level quantum-classical software framework that directly enables this compilation lowering phase via user-specified quantum optimal control techniques. This extension enables the translation of digital quantum circuit representations to equivalent pulse sequences that are optimal with respect to the backend system dynamics. Our work is modular and extensible, enabling third party optimal control techniques and strategies in both C++ and Python. We demonstrate this extension with familiar gradient-based methods like gradient ascent pulse engineering (GRAPE), gradient optimization of analytic controls (GOAT), and Krotov's method. Our work serves as a foundational component of future quantum-classical compiler designs that lower high-level programmatic representations to low-level machine instructions.**

I. Introduction

Extensible and modular software architectures for quantum-classical computing are proving essential for researchers that require a quick prototyping capability and workflow customization [1], [2]. Moreover, it has become increasingly evident that low-level, pulse-level access to nascent hardware will enable improved error mitigation and smart quantum-program generation technique [3]–[5]. There is a unique need for extensible software architectures that enable customization of the pulse-level programming, compilation, and execution workflows.

Recently we presented a novel update to the XACC system-level quantum-classical programming, compilation, and execution framework that enables direct pulse-level programming [6]. Specifically, we demonstrated an extension to the XACC quantum intermediate representation for pulse-level instructions, compiler scheduling routines, and an OpenPulse adherent simulation backend built on the QuaC open quantum dynamics simulator [7]. We believe this is foundational for the creation of higher-level quantum compiler technologies that streamline pulse-level programming research and development activities. In this work, we build off the initial XACC pulse-level software infrastructure to provide an extensible framework for typical (or custom) optimal quantum control strategies. Optimal quantum control techniques have been long-established, and a number of popular strategies exist based on typical gradient methods (GRAPE, GOAT, Krotov, etc.) [8]–[10]. Our goal here is to present a unique software architecture that extends the XACC framework to enable the implementation of these typical control strategies in a plug-and-play manner. Ultimately, we demonstrate how this extension can be used for prototypical quantum compilation routines that lower gate-level program representations to an optimal pulse-sequence.

We begin this work with a brief description of quantum optimal control and various gradient-based algorithms that we integrate with XACC (Sec. II). Next, we briefly provide a background on pulse-level programming and simulation in XACC (Sec. III), and describe in detail our extension to the IR transformation service framework enabling modular quantum optimal control strategies (Sec. IV). We finish with a detailed demonstration of the utility of this extension for a number of problems (Sec. V).

II. Quantum Optimal Control

To enable the control and programmability of physical qubits, one must be able to generate analog control signals that affect a desired unitary evolution with high degrees of precision. This is typically accomplished using a closed-loop feedback system with an iterative optimization algorithm acting as the controller. This task, quantum optimal control [11], seeks to map higher-level quantum programmatic representations to an optimal sequence of pulses that realizes the unitary evolution of the program.

Quantum control algorithms typically use a loss function created out of the difference between the target state and the evolved state at time τ as a fidelity metric to guide their training. A commonly used field of algorithms are gradient-based, in that they learn the optimal controls of the system by iteratively updating them in the direction of the loss function gradient. For piecewise constant pulses [12], this would mean iteratively updating the pulse amplitude at each time step. With Gaussian pulses [13], it would entail iterative updates to the mean and standard deviation of the distribution and with basis functions [14], [15]. The optimization routine is terminated once the target and evolved state reach maximum overlap with some predetermined precision.

In this section, we seek to detail prototypical examples of gradient-based quantum optimal control strategies that we leverage in this work.

a) GRAPE: A commonly used algorithm for quantum control is Gradient Ascent Pulse Engineering, or GRAPE [8]. GRAPE utilizes a discretized approximation of Schrodinger's equation to create piecewise constant control pulses. The total pulse time, τ, is broken up into N time-steps, all of duration Δt. GRAPE seeks to find the optimal pulse amplitude at each of the N time steps, typically subject to a value constraint on the amplitude. The time evolution of the system, in terms of its Hamiltonian, can thus be approximated as $\mathcal{U}(t_n) = exp[-i\,\mathcal{H}(t_n)\,\Delta t]$, giving evolution at time τ as:

$$\mathcal{U}(\tau) = \mathcal{U}(t_N)\,\mathcal{U}(t_{N-1})\,...\,\mathcal{U}(t_0) \tag{1}$$

The loss function that we seek to minimize, known as the infidelity, is:

$$\mathcal{L} = 1 - \frac{1}{d}\Big|Tr\big[\mathcal{U}^{\dagger}_{target}\,\mathcal{U}(\tau)\big]\Big|^2 \tag{2}$$

where d is the dimension of the system Hilbert space and $\mathcal{U}_{target}$ is a user-specified target quantum gate.

After initializing all N amplitudes, either randomly or with a pulse provided by the user, the pulse is fed to the hardware/simulator, and measurements of the system are taken. The fidelity is then recorded, and the pulse amplitudes are updated in small steps towards the direction of greatest ascent of the fidelity's gradient. The step size is a hyperparameter that may be adjusted by the user, and is used to prevent the optimizer from over or under stepping its updates. Optimization is terminated once Eq. 2 is at a minima (up to some tolerance).

b) GOAT: Gradient Optimization of Analytic conTrols, or GOAT [9], is another popular gradient based technique for optimal control. Unlike GRAPE, GOAT is not limited to piecewise constant controls and, therefore, can be used to create piecewise continuous pulses with bandwidth constraints. In our implementation, we optimize pulses within the GOAT framework out of a superposition of (K) Gaussian pulses of the form:

$$\Omega(t) = \sum_{k=0}^{K} exp(-(t-\tau_k)/\sigma_k^2) \tag{3}$$

where τ_k and σ_k are the pulse duration and standard deviation of each of the K Gaussians respectively. The loss function takes the same form as in GRAPE (Eq. 2), but with the time ordered Unitary evolution now being:

$$\mathcal{U}(\tau) = \mathcal{T}exp\Big[-\frac{i}{\hbar}\int_0^{\tau}\mathcal{H}(t)dt\Big] \tag{4}$$

For a specified amount of pulses, GOAT seeks to optimize over the set $\mathcal{A}_k = \{\tau_k, \sigma_k\}$. Beginning with either a randomly initialized or user-provided $\mathcal{A}_k$, GOAT uses the second-order derivative optimizer, L-BFGS [16], to iteratively minimize the loss function (Eq. 2). The unitary evolution (Eq. 4) is then computed using the third order Runge-Kutta numerical integration algorithm [17]. Training is again terminated once Eq. 2 is minimized.

c) Krotov: As opposed to GRAPE and GOAT, which use concurrent updates to optimization parameters, Krotov [10], [18] uses sequential updates to guarantee monotonic convergence, all without the need for gradient calculations [19]. Application of the Krotov method requires reformulation of the system in terms of density matrices, $\rho(t)$. The system evolves in time according to:

$$\frac{\partial}{\partial t}\rho(t) = \frac{1}{\hbar}\mathcal{L}(t)\rho(t) \tag{5}$$

where $\mathcal{L}(t)$ is the Liouvillian. For the application of quantum gates to the system, the problem may be viewed as controlling the unitary time evolution of the set of basis states, $\{|\psi_k(t)\rangle\}$. Krotov seeks to minimize a functional, $J\big[\{|\psi_k(t)\rangle\}, \{\Omega_i(t)\}\big]$ with constraints and boundary conditions imposed through Lagrangian multipliers, and the loss function being $\nabla_{\psi,\Omega}J = 0$ [20]. Beginning with either a randomly initialized or user-specified pulse, Krotov sequentially updates the controls by:

$$\Omega_i(t) = \Omega_{i-1}(t) + \Delta\Omega_i(t) \tag{6}$$

until the functional is globally minimized.

III. Pulse-level Programming in XACC

XACC is a system-level software framework enabling typical heterogeneous quantum-classical computing workflows [1]. XACC can be decomposed into front-end, middle-end, and backend layers, enabling the translation of quantum kernels to a core polymorphic intermediate representation (IR), translations and optimizations on that IR, and hardware-agnostic

backend execution. XACC exposes standard interfaces at all levels of this hierarchy. An interface or service of note for this work is the IR Transformation, which enables one to define general transformations on the intermediate representation. This is useful for quantum-compile time routines, and as we show in this work, mapping digital circuit representations to optimal pulse sequences. XACC has been leveraged in a number of experimental demonstrations of quantum-classical computing and general benchmarking [21]–[23].

Recently, XACC has been updated with support for analog quantum programming. Here, we briefly summarize the key features of pulse-level programming in XACC, but interested readers are referred to [6] for a more comprehensive description.

A. Analog Instructions

At its core, XACC puts forward a polymorphic Intermediate Representation (IR) as an extensible data structure that encapsulates quantum programming semantics – from single quantum gates to composite quantum circuits. IR data structures are constructed by front-end compiler plugins, processed by middle-end IR transformation plugins (e.g., circuit optimization) and then executed on available quantum backends, e.g., remote quantum hardware or simulators.

Since the emergence of pulse-level programming, we have extended this key infrastructure of the framework to handle analog-like instructions. Specifically, we added additional fields to capture discrete pulse samples, its start time, and the target channel. Pulse instructions can then be parsed from vendor-provided pulse libraries (JSON objects), constructed manually by providing data arrays or programmatically by using a native XACC pulse generation utility which automatically discretizes commonly-used pulse shapes.

An important aspect of pulse-level programming is the ability to automatically lower digital gates into sequences of pulses. The polymorphic hierarchy of XACC IR enables a unified representation of composite instructions, i.e. groups of other instructions (basic instructions or composite instructions), hence digital gate instructions can be replaced by a pulse composite IR which consists of multiple pulse instructions.

B. Digital-to-Analog Transpiling

Standard IR lowering from digital to analog is included in the basic pulse extension of XACC. When used with a pulse-capable backend, such as the QuaC simulator (Sec. III-C) the framework will use the backend-associated default pulse library to transpile digital gates into pulses.

Pulse libraries typically only contain pulse sequence definitions for a pre-determined set of universal gate sets, e.g. single qubit U gates and two-qubit CNOT/CZ gates between neighboring qubits [24]. In XACC, we support a much wider range of gates. Thus, for those gates that do not have direct pulse sequence definitions, XACC transpiles them into gates drawn from the backend gate set to enable pulse conversion. Pulse sequences associated with gates are time-shifted accordingly to maintain the atomicity of quantum gates.

The result of this lowering procedure is a purely-analog composite instruction consisting of pulse instructions on different channels at different start times. This combinatorial approach toward digital-to-analog lowering is the fundamental building block of the XACC pulse programming environment upon which the quantum-control-based approach that we present here is built. Not only does quantum optimal control provide a means to derive basic pulses to construct a pulse library, but it also enables novel use cases, such as custom pulse implementation or sub-circuit optimization, which we discuss in Sec. IV.

C. QuaC Accelerator Backend

In addition to a wide variety of gate-based simulation backends that are currently available in XACC, we have also implemented an OpenPulse-compatible simulation backend based on the QuaC (Quantum in C) quantum dynamics solver [7]. This analog backend enables users to experiment with pulse-level programming as well as to develop and verify custom digital-analog transformation procedures, e.g. those that are put forward in this manuscript.

Key components of the QuaC pulse backend are:

- A high-performance time-stepping solver based on the PETSc library which has built-in support for MPI parallelization [25]. This allows us to optimize the simulator performance on platforms ranging from laptops to computer clusters.
- An OpenPulse-compatible frontend that can process pulse-level backend information in OpenPulse format. This includes system dynamics (Hamiltonian and qubit dimensionality), drive/control channel configurations, and pulse library.
- A pulse generation utility which supports automatic discretization of analytical pulse envelopes.

The QuaC pulse backend implements the standardized `Accelerator` interface of XACC, hence it can be used as a drop-in replacement for any other existing gate-based backends as well as in the Pythonic programming environment.

IV. Software Architecture

In the XACC framework, quantum optimal control capabilities are tightly integrated into the end-to-end programming model, as illustrated in Fig. 1. More specifically, with the core intermediate representation (`IR`) providing a universal data structure for describing both digital and analog quantum instructions, we are able to encapsulate quantum optimal control strategies as general transormations of the IR, specifically via a new pulse-level IR Transformation service providing a flexible gate-to-pulse lowering/compilation procedure.

Since the input IR can represent individual gates as well as whole quantum circuits, e.g. variational ansatz state-preparation, Quantum Fourier transform, etc., this new pulse-level transformation infrastructure of XACC can provide substantial improvements in areas such as execution fidelities and efficiencies (total time of gate operation).

It is worth noting that, while quantum optimal control modules are utilized internally within the pulse-level IR transformation workflow, they can also be used independently as a service. For example, one can manually define the target unitary and the system definition and then invoke any quantum optimal control module that the framework provides.

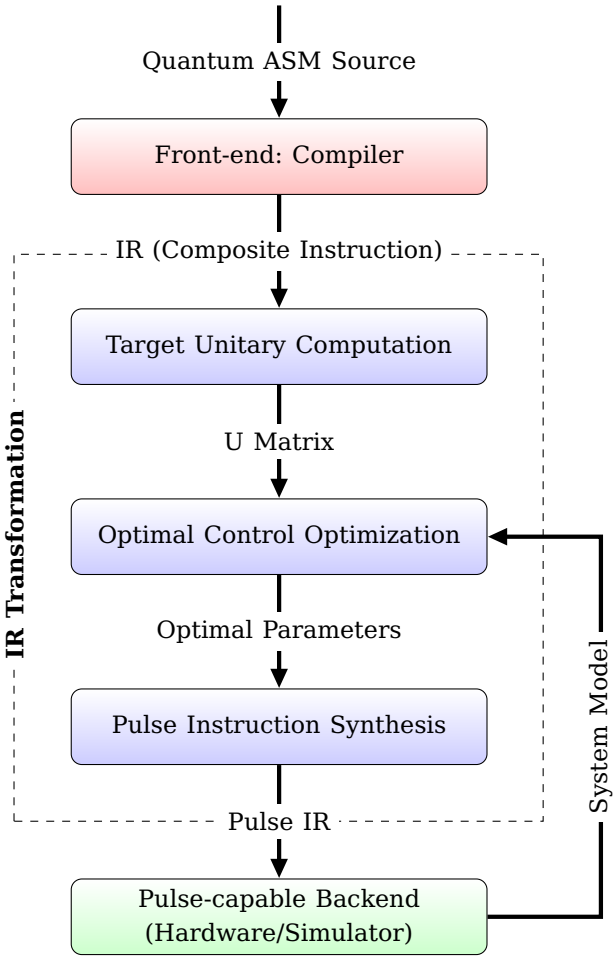

Fig. 1: XACC pulse level IR transformation flow.

In the following, we will describe the three main components of the pulse-level IR transformation workflow, as illustrated in Fig. 1, in greater detail.

A. IR Transformation

Built upon the concept of compiler optimization routines from classical computing, XACC defines the `IRTransformation` interface as the backbone of the middle-end pipeline. This allows modular and customizable multi-pass transformations of quantum kernels parsed by the front-end. As shown in Fig. 2, we already have built-in support for several gate-based transformations such as gate optimization and qubit placement.

In this work, we provide a new `IRTransformation` service called `PulseTransform`, which bridges digital and pulse `IR`'s via quantum optimal control. When invoked with an `apply()` call, the `PulseTransform` service is provided with a `CompositeInstruction` describing a quantum circuit and an instance of a pulse-capable back-end, such as the XACC QuaC simulator (see Fig. 1 & 2). The backend supplies the system dynamics information which is required by downstream control modules to compute analog driving signals.

One key functionality that the top-level `PulseTransform` service performs is to convert arbitrary gate-based circuits into their equivalent unitary matrix. Thus, the framework can transform either individual gates or multi-gate circuits into monolithic pulse programs representing the underlying total unitary evolution.

As described in the next section, we also implement a wide variety of quantum optimal control modules as well as provide a user-friendly interface to integrate custom pulse optimizers, any of which can be used in this digital-to-analog IR transformation pipeline. By specifying the method name in the input `HeterogeneousMap` of the `apply()` call, the corresponding optimal control plugin will be delegated (Fig. 2) to perform the optimization task. Also, the computed target unitary matrix along with system definitions are sent on to downstream optimization plugins as needed.

B. Quantum Control Optimizer

The QCOR language specification [26] put forward the `Optimizer` data structure which provides a common interface for all classical optimization services, and XACC has provided the first definition and implementation of it. In a conventional optimization setting, an `Optimizer` implementation will perform a multi-dimensional optimization of a target cost function

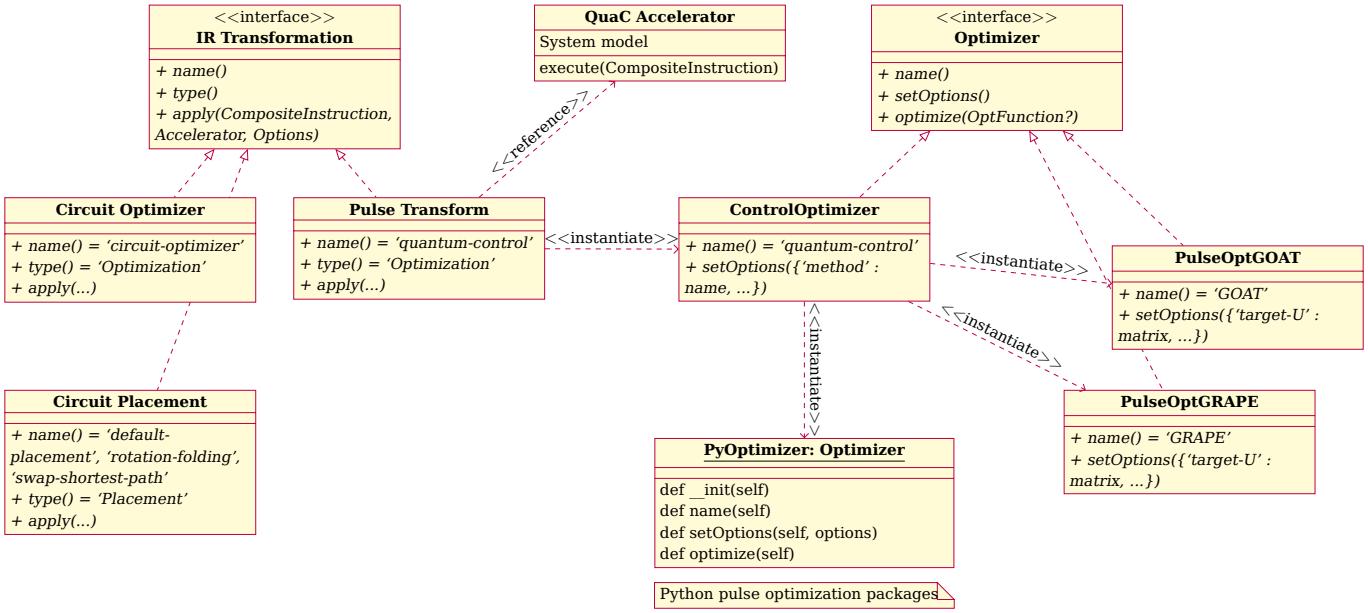

Fig. 2: XACC pulse-level IR transformation software architecture: single service entrance via the Pulse Transform plugin; the specific optimal control method to be used for circuit-to-pulse transformation is provided as an option; both native (C++) and Python pulse optimization methods can be invoked via this unified API.

($argmin_{x \in \mathbb{R}^n} f(x)$). This function is listed as the (input `OptFunction` parameter of the optimize method of the `Optimizer` interface in Fig. 2.

The `Optimizer` interface is thus amenable to quantum optimal control problems underpinning analog control synthesis. For instance, one can consider control values at each discrete time step as parameters to be optimized and use any fidelity measures, such as the trace distance, as the function to be optimized. To maximize flexibility, modularity, and reusability of these quantum optimal control sub-routines, we design a two-level interface for the quantum optimal control service.

At the top-level, we define a `ControlOptimizer` service which can be invoked by its generic name, `quantum-control`. The caller then provides a `HeterogeneousMap` which contains a method field indicating which optimal control method to be used along with an arbitrary set of additional parameters. The `ControlOptimizer` will then look up in the XACC service registry for the specified optimal control module, initialize it with method-specific parameters (either user-provided or default values customized for each method), and delegate the `optimize()` call to the concrete implementation provided by each optimal control module.

At the time of writing this manuscript, we have implemented the `GOAT` and `GRAPE` pulse optimization methods within the XACC framework as native plugins. Hence, these two methods are available universally on any XACC installations. We also provide a Python binding interface (see `PyOptimizer` in Fig. 2) through which one can wrap Pythonic quantum optimal control modules, e.g. QuTiP [27], and contribute them as services to be used in the XACC IR transformation workflow. In the demonstration section, we will demonstrate the use of the Krotov package, which depends on QuTiP, as a backend optimizer for a pulse-level IR transformation.

C. Pulse Instruction Synthesis

As illustrated in Fig. 1, the final output of the pulse-level `IRTransformation` is a pulse `CompositeInstruction` which contains pulse instructions. Those analog instructions are defined in terms of arrays of complex-valued samples representing control signals at a backend specific sampling rate of dt. Depending on the specific optimal control method that was used in the previous step, the results may not immediately be in the right format.

For example, analytical optimization approaches, such as GOAT, produce a set of functional parameters representing time-continuous signals. In that case, the Pulse Transform service will evaluate those envelope functions and generate the corresponding data samples to construct output pulse instructions. On the other hand, time-series based methods, e.g. piecewise constant optimization, should already generate the optimized pulses as sample arrays which can be used to construct pulse `IR`'s directly.

```python
# Run Quantum Optimal Control GOAT method to
# find gaussian pulse that approximates the X
# gate on qubit 0
goat = xacc.getOptimizer('quantum-control', {
    'method':'GOAT',
    'dimension': 1,
    'target-U':'X0',
    'control-params':['sigma'],
    'control-funcs':['exp(-t^2/(2*sigma^2))'],
    'control-H':['X0'],
    'max-time': 100,
    'initial-parameters':[8.0]
})

optimal_sigma = goat.optimize()[1][0]
```

Fig. 3: Using the `ControlOptimizer` service, i.e. the XACC Optimizer plugin named "quantum-control", to optimize for a π-pulse (X gate) using the GOAT method. Since this is an analytical method, we need to provide the functional form of controls and the `optimize()` function will return the optimal parameters for those input control functions.

```
// Get QuaC accelerator (pulse-capable backend):
// systemModel contains Hamiltonian & channel configs.
auto quaC = xacc::getAccelerator("QuaC", {
  std::make_pair("system-model", systemModel),
});
auto compiler = xacc::getCompiler("xasm");
// Using the XASM compiler to compile ASM to IR
auto ir = compiler->compile(R"(__qpu__ void circ(qbit q){
  H(q[0]);
})", quaC);
auto program = ir->getComposite("circ");
// Pulse IR transformation configs, using GRAPE:
xacc::HeterogeneousMap configs {
  std::make_pair("method", "GRAPE"),
  std::make_pair("max-time", 10)
};
// Get the pulse-level IR Transformation service
auto opt = xacc::getIRTransformation("quantum-control");
// Apply the transformation on the gate-level program
opt->apply(program, quaC, configs);
// After the transformation, the program is converted
// optimal pulse instructions which
// can be simulated on the QuaC backend
auto qubitReg = xacc::qalloc(1);
quaC->execute(qubitReg, program);
```

Fig. 4: Using the pulse-level `IRTransformation` service to convert a quantum circuit, in this case, just a Hadamard gate, into an optimal pulse instruction using the GRAPE optimizer.

V. Demonstration

A. *ControlOptimizer API*

In this example, we show how the underlying quantum control optimizer plugins can be used directly. This type of usage fits physics-based experiments whereby the system dynamics, target unitary, and method-specific parameters need to be provided and are fully customizable by users.

Fig. 3 is a Python snippet demonstrating the way that underlying quantum optimal control modules (plugins) are invoked. As shown in Fig. 2, all of those modules are sub-components of the high-level `ControlOptimizer` which is a generic XACC Optimizer with name "quantum-control" (Fig. 3, line 4.)

In this example, we request the *GOAT* (**G**radient **O**ptimization of **A**nalytic con**T**rols) optimizer by specifying its key in the `method` field (Fig. 3, line 5.) The XACC `HeterogeneousMap` utility allows us to pass flexible data-structures in a type-safe manner to the underlying native (C++) module. This makes the Python-C++ integration seamless as demonstrated in this example.

Those control parameters after the `method` field in the `getOptimizer()` call are method-specific. For example, since GOAT is an analytical method, users need to specify the functional form of control envelopes. It is also worth pointing out that due to the fact that this is a direct invocation of an optimal control module, there are quite a few parameters that need to be specified, such as those related to the system dynamics (Hamiltonian) and the target unitary. When these underlying optimal control modules are invoked within the `IRTransformation` workflow (Fig. 1), most of these parameters will be derived automatically by the high-level `IRTransformation` service.

B. *PulseTransform API*

As a system-level framework, optimal control modules within XACC are tightly integrated with the end-to-end compilation, transformation, execution software workflow. Fig. 4 demonstrates the use of our built-in *GRAPE* optimizer to derive an optimal pulse shape that implements a quantum circuit. The input circuit is given as an assembly source (XASM dialect) which is compiled into digital `IR` by XACC's `Compiler` service.

This digital gate-based circuit, in this case, just a single quantum gate, is given to the pulse-level `IRTransformation` service along with a reference to a pulse-capable backend and a set of configuration parameters (see line 23, Fig. 4.) In this example, we want to request the gate-to-pulse transformation via the GRAPE method, which only requires a single parameter (maximum time horizon, `max-time`.) The rest of the necessary parameters, e.g. the number of data samples (GRAPE is a time-series based method), the Hamiltonian, the target unitary, etc. , are deduced by the `IRTransformation` service which has access to the

execution backend and the IR of the digital quantum circuit.

The input digital program is lowered to a corresponding analog program at the end of the `apply()` procedure. This pulse program can then be executed on the backend as shown in lines 27-28 of Fig. 4.

C. External Python Package Integration

In an effort to take advantage of quantum optimal control modules which are currently available either as open-source software, e.g., QuTiP, or as commercial solutions, e.g. Q-CTRL [28], the XACC framework provides a set of Python bindings which can be used to wrap external Python modules as plugins (see Fig. 2.) The primary task of the wrapper is to translate standardized data which the `IRTransformation` plugin sends on to the user-requested quantum control method into the implementation-specific data format. For instance, after reading the input digital circuit, the `IRTransformation` module will generate the target unitary matrix represented by a complex-valued vector. This bare array may need to be marshaled into the expected data format of the external Python package.

As an example, we have implemented a wrapper for the *Krotov* package which depends upon QuTiP. From the top-level, i.e. `IR Transformation`, the usage is completely analogous to other built-in optimal control modules as can be seen in Fig. 6. The specific method key, in this case, 'krotov' (line 14), is registered by the wrapper service. The data generated by the Krotov package while performing the pulse optimization

```
1  # Get the XASM compiler
2  xasmCompiler = xacc.getCompiler('xasm');
3  # Composite to be transform to pulse: H gate = Y^(1/2) - X
4  ir = xasmCompiler.compile('''__qpu__ void f(qbit q) {
5      Ry(q[0], pi/2);
6      X(q[0]);
7  }''', qpu);
8  program = ir.getComposites()[0]
9  # Run the pulse IRTransformation
10 optimizer = xacc.getIRTransformation('quantum-control')
11 optimizer.apply(program, qpu, {
12     # Using the Python-contributed pulse optimizer (Krotov
13     'method': 'krotov',
14     'max-time': T
15 })
```

Fig. 6: Using the pulse `IRTransformation` service with an external Python plugin (Krotov) as the optimizer. In this example, we optimize for, effectively, an Hadamard gate (expressed as a $XY^{1/2}$ circuit.)

procedure is illustrated in Fig. 5. Starting with an arbitrary guess pulse, it drives the pulse envelope to an optimal shape via a method-specific update policy to implement the target unitary. It is worth noting that, in this example, we purposely specify the Hadamard gate as a $Y^{1/2} - X$ gate sequence to demonstrate the total unitary computation functionality of the `IRTransformation` service. The underlying optimal control module will only receive the computed target unitary matrix as a black box.

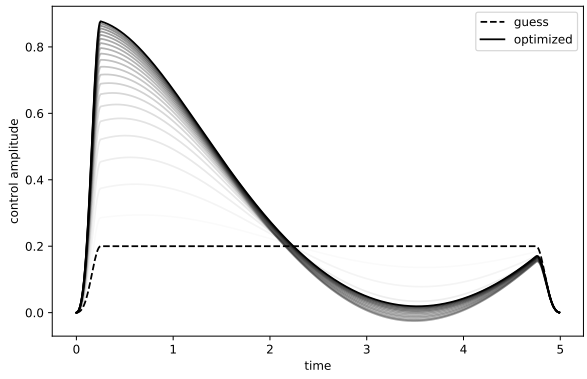

Fig. 5: Optimal control signal generated by the Krotov pulse optimizer plugin when invoked by the XACC `IRTransformation` service to optimize for an Hadamard unitary. The dash line is the initial pulse (randomly selected) and the blurred lines represent optimization iterations until it converges to the optimized pulse (the solid line.)

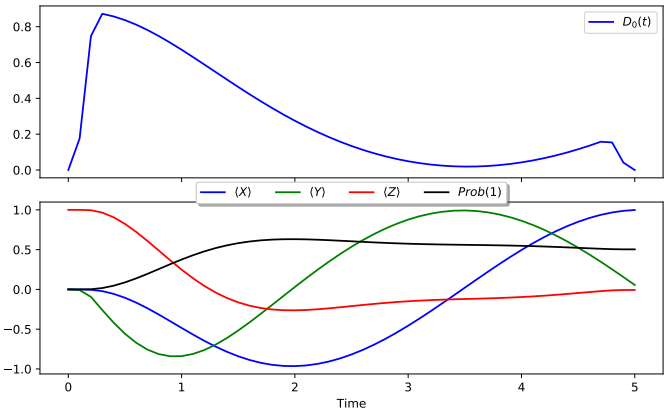

Fig. 7: Execution results on the QuaC backend for the optimal pulse derived by the `IRTransformation` service using the Krotov optimizer. The `IRTransformation` transforms a Hadamard-gate equivalent circuit ($Y^{1/2} - X$) into an analog pulse (*top* panel). The time-domain response on the QuaC backend (expectation values of the number operator and Pauli operators) in the *bottom* panel confirmed a Hadamard response.

The optimal pulse `IR` after transformation can then be executed on a pulse-capable backend (like lines 27-28 of Fig. 4.) The verification results for the Krotov-optimized pulse on our QuaC simulator backend is shown in Fig. 7. The pulse shape, as executed on the backend, is the output from the Krotov optimizer. The expectation values of X, Y, and Z operators indicate that this pulse indeed performs a Hadamard transformation (the initial state is $|0\rangle$.)

D. Comparison Across Techniques

One key benefit of having multiple pulse optimizers implemented as plugins in a micro-services approach is that users can easily examine different methods in a plug-and-play manner. For instance, we can request an X gate to be transformed into pulses in a similar way to the code snippet in Fig. 6 (with a different XASM kernel string), in which the method field can be either '`GOAT`', '`GRAPE`', or '`Krotov`'. Correspondingly, the appropriate plugin will be invoked to perform the optimization task with the same set of inputs, i.e. the system Hamiltonian dynamics encapsulated by the `qpu` instance and the target unitary of the gate-based `program`.

The generated pulses and the excited state population responses are shown in Fig. 8. It is worth noting that the generated pulses strongly depend on the initial guess pulses which are Gaussian, square, and random for GOAT, Krotov, and GRAPE, respectively.

Similarly, we can also introduce non-ideal effects such as finite qubit decay [29] or local-oscillator detuning [30] (from the exact qubit frequency) to the simulator backend and examine the performance of generated pulses under such circumstances. The results for dissipative and detuning experiments are shown in Fig. 9 and 10, respectively. The varying nature of the pulse generated by the randomly-initialized GRAPE method results in loss of fidelity under both the dissipative condition (Fig. 9) and static detuning (Fig. 10).

The results presented here are model- and

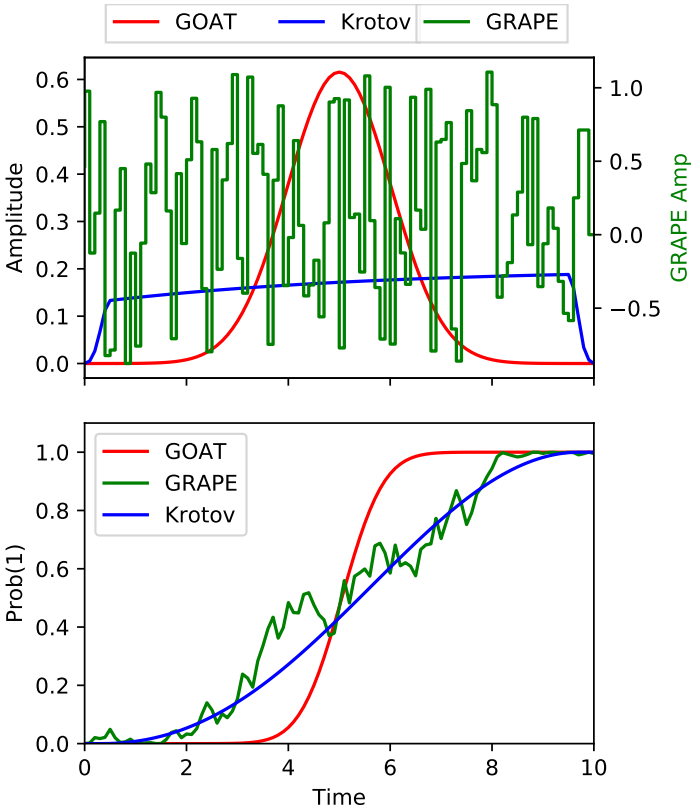

Fig. 8: Comparison between pulses generated by GOAT, GRAPE, and Krotov optimizers when performing `IR-Transformation` for an X gate: (top) pulse envelopes and (bottom) excited state population (initial state = $|0\rangle$).

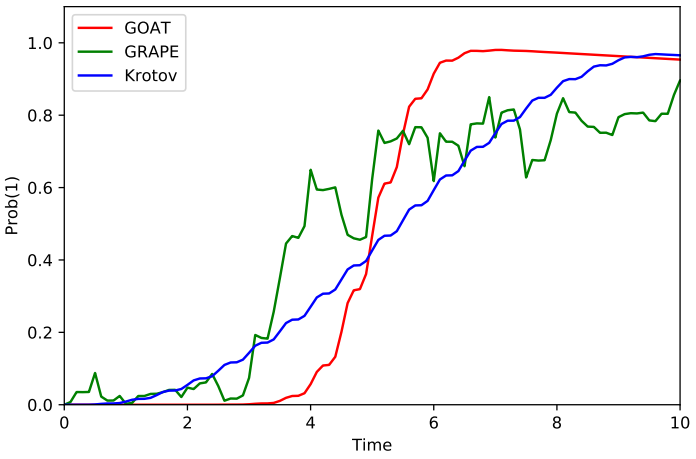

Fig. 9: Comparison between X-gate pulses generated by GOAT, GRAPE, and Krotov optimizers with qubit decay ($T_1 = 10 \times T_{gate}$).

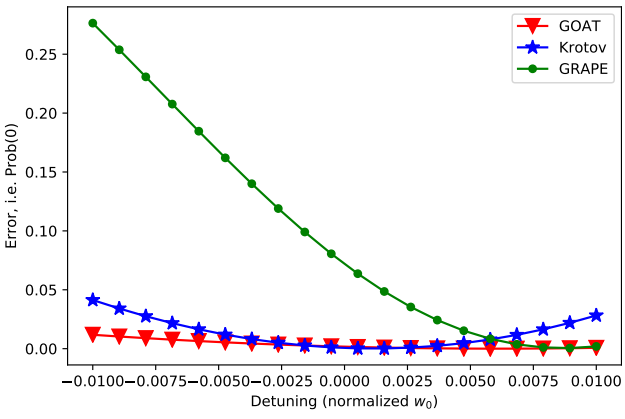

Fig. 10: Comparison between X-gate pulses generated by GOAT, GRAPE, and Krotov optimizers with static detuning ($\omega_{LO} = (1 + \delta)\omega_0$).

configuration-dependent but demonstrating the capability and utility of the new quantum control extension to the XACC framework. The modularity and compatibility of XACC services enable users to quickly evaluate different quantum control solutions on a unified API.

E. Multi-qubit Circuit

By integrating pulse optimization modules into the IR transformation workflow, as shown in Fig. 1, we can transform the whole quantum circuit consisting of multiple gates into a monolithic pulse program implementing the overall unitary. This might be beneficial, e.g., for frequently used sub-circuits because one can derive a unified set of pulses across multiple channels to achieve the overall unitary of the sub-circuit, rather than assembling individual gate pulses.

In the following example, we use the GRAPE pulse optimizer (Fig. 11, line 9) to convert a two-qubit Quantum Fourier Transform (QFT) circuit into a pulse program. It is worth noting that the XACC framework has built-in support for circuit generation of common algorithms. Hence, the QFT circuit (line 1-4 in Fig. 11) is automatically expanded to the gate sequence.

In this example, we consider a generic Hamiltonian model in the form

$$H = \sum_{k\in\{x,y,z\}} u_k(t)\sigma_k^{(0)}\sigma_k^{(1)} + \sum_{i=0}^{1}\sum_{k\in\{x,y,z\}} d_k^{(i)}(t)\sigma_k^{(i)}, \quad (7)$$

where $u_k(t), d_k^{(i)}(t)$ are control terms that our optimal control plugin aims to optimize. This simplified model is for illustrative purposes only and does not correspond to any particular physical system.

The pulse waveforms for $d_k^{(i)}(t)$ channels are shown in Fig. 12. Similarly, we also obtained pulses for $u_k(t)$ channels which are not shown here for brevity. This pulse program can then be verified on the QuaC simulator backend, i.e. the `qpu` instance which was given to the optimizer (Fig. 11, line 8).

```
1  # Use XACC QFT circuit generator
2  qft = xacc.getComposite('qft')
3  # Expand QFT circuit for 2 qubits
4  qft.expand({ 'nq' : 2 })
5  # Run the pulse IRTransformation
6  optimizer = xacc.getIRTransformation('quantum-control')
7  optimizer.apply(qft, qpu, {
8      'method': 'GRAPE',
9      'max-time': T,
10     'dt': dt
11 })
```

Fig. 11: Python code snippet demonstrates the pulse-level IR transformation of a two-qubit Quantum Fourier Transform circuit into a pulse program using the GRAPE method.

F. Demo with physical hardware

The pulse IR transformation service of XACC also provides a means for users to optimize for particular gates that suit their needs. Hardware vendors usually provide a minimal set of pulses which implement a target universal gate set to which other gates are decomposed. Using the XACC pulse IR transformation

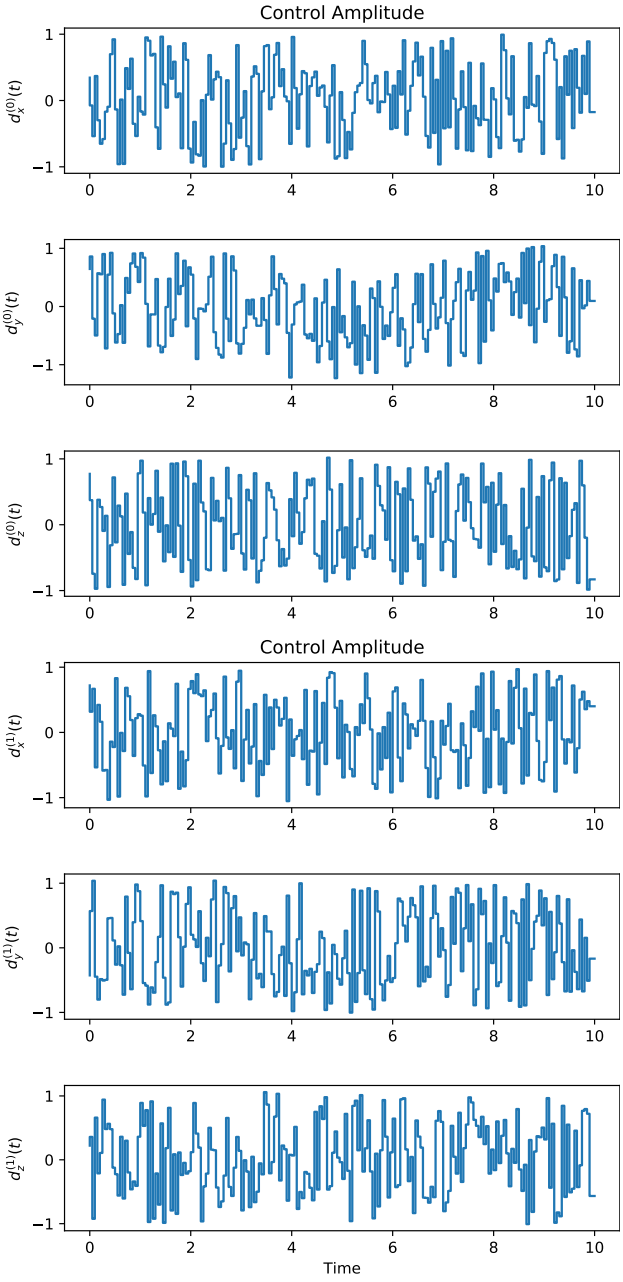

Fig. 12: Control signals for single-qubit operator terms in Eq. (7).

service, users can instead opt for a more custom approach as illustrated in Fig. 13 whereby a parametric rotation gate at a specific angle is directly lowered to a pulse instruction.

Using IBM Qiskit [2] we can examine the equivalent pulse sequences compiled by the vendor-provided pulse library to implement those parametric gates. These pulses are what will be applied to the underlying hardware when users submit gate-based circuits via XACC, Qiskit, or other front-end tools to the cloud backend.

Fig. 14 is a comparison between pulses that are optimized by the XACC GOAT optimizer to implement $R_x(\theta)$ gates at specific θ values vs. default pulse sequences generated by the Qiskit transpiler for the IBM one-qubit Armonk device. On average, pulses optimized by the XACC optimizer have comparable fidelity to the default ones. When using this IR transformation method for longer gate sequences, we can potentially have a significant gain in terms of execution time and fidelity.

VI. Discussion and Future Work

We have expanded upon XACC's pulse-level programming capacity by integrating several common quantum optimal control algorithms into our framework. With an emphasis on ease of implementation, we provide users the ability to take their quantum circuits and compile them to an optimally shaped control pulse. We currently support algorithms such as GRAPE, GOAT, and Krotov, in both C++ and Python, with plans to offer more algorithms in the future. Additionally, we hope to exploit the MPI parallelization feature of QuaC in order to use cluster based computing resources such as Oak Ridge's Summit Supercomputer [31]. This would allow for accurate simulation and subsequent control optimization routines to be run on larger quantum systems. Our long-term focus is both on maintaining our database of calibrated quantum hardware system models, as well as expanding the number of hardware providers supported by our framework for pulse-level control.

```
xasmCompiler = xacc.getCompiler('xasm');
ir = xasmCompiler.compile(
'''__qpu__ void Kernel(qbit q, double theta) {
    Rx(q[0], theta);
}''', qpu);
program = ir.getComposites()[0]
for angle in angles:
    evaled = program.eval([angle])
    # Run the pulse IRTransformation
    optimizer=xacc.getIRTransformation('quantum-control')
    optimizer.apply(evaled, qpu, { <options>})
```

Fig. 13: Pulse optimization for parametric gates.

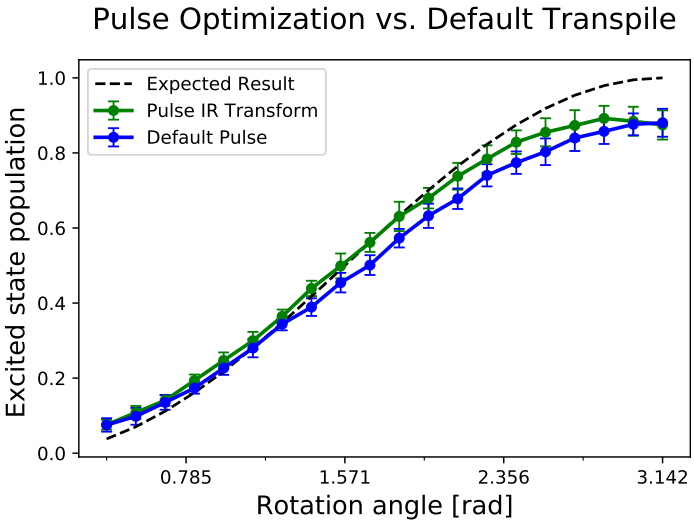

Fig. 14: Results from running $Rx(\theta)$ gate with different rotation angles on the IBMQ `armonk` (one qubit) backend. In the runs using XACC pulse optimization (green), rotation gate at a specific angle is transformed into a Gaussian-shaped envelope pulse (GOAT method). In default transpile runs (blue), default Qiskit transpiler is used. For each angle, 1024 shots are used to compute the excited state probability. There are 10 runs for each angle.

Acknowledgements

This work has been supported by the US Department of Energy (DOE) Office of Science Advanced Scientific Computing Research (ASCR) Quantum Computing Application Teams (QCAT), Quantum Testbed Pathfinder (QTP), and Accelerated Research in Quantum Computing (ARQC). A. S. was supported by an appointment to the Oak Ridge Associated Universities (ORAU) Post-Bachelors Program, sponsored by the U.S. Department of Energy and administered by the Oak Ridge Institute for Science and Education. ORNL is managed by UT-Battelle, LLC, for the US Department of Energy under contract no. DE-AC05-00OR22725. The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan.

References

[1] Alexander J McCaskey, Dmitry I Lyakh, Eugene F Dumitrescu, Sarah S Powers, and Travis S Humble. XACC: a system-level software infrastructure for heterogeneous quantum–classical computing. *Quantum Science and Technology*, 5(2):024002, feb 2020.

[2] Gadi Aleksandrowicz et al. Qiskit: An open-source framework for quantum computing, 2019.

[3] Yunong Shi, Nelson Leung, Pranav Gokhale, Zane Rossi, David I. Schuster, Henry Hoffman, and Fred T. Chong. Optimized Compilation of Aggregated Instructions for Realistic Quantum Computers. *arXiv e-prints*, page arXiv:1902.01474, February 2019.

[4] David C. McKay et al. Qiskit backend specifications for openqasm and openpulse experiments. *arXiv preprint arXiv:1809.03452*, 2018.

[5] Pranav Gokhale, Ali Javadi-Abhari, Nathan Earnest, Yunong Shi, and Frederic T. Chong. Optimized Quantum Compilation for Near-Term Algorithms with OpenPulse. *arXiv e-prints*, page arXiv:2004.11205, April 2020.

[6] Thien Nguyen and Alexander McCaskey. Enabling pulse-level programming, compilation, and execution in xacc. *arXiv preprint arXiv:2003.11971*, 2020.

[7] Otten Matthew and Finkel Hal. Quac âĂŤ quantum in c, 2020. https://github.com/0tt3r/QuaC.

[8] Re-Bing Wu, Bing Chu, David H. Owens, and Herschel Rabitz. Data-driven gradient algorithm for high-precision quantum control. *Phys. Rev. A*, 97:042122, Apr 2018.

[9] Shai Machnes, Elie Assémat, David Tannor, and Frank K. Wilhelm. Tunable, flexible, and efficient optimization of control pulses for practical qubits. *Phys. Rev. Lett.*, 120:150401, Apr 2018.

[10] David J. Tannor, Vladimir Kazakov, and Vladimir Orlov. *Control of Photochemical Branching: Novel Procedures for Finding Optimal Pulses and Global Upper Bounds*, pages 347–360. Springer US, Boston, MA, 1992.

[11] J Werschnik and E K U Gross. Quantum optimal control theory. *Journal of Physics B: Atomic, Molecular and Optical Physics*, 40(18):R175–R211, sep 2007.

[12] Ilan Degani, Antonella Zanna, Lene SÃęlen, and Raymond Nepstad. Quantum control with piecewise constant control functions. *SIAM J. Scientific Computing*, 31:3566–3594, 01 2009.

[13] S. G. Schirmer, A. D. Greentree, V. Ramakrishna, and H. Rabitz. Quantum control using sequences of simple control pulses, 2001.

[14] H. Ball and M.J Biercuk. Walsh-synthesized noise filters for quantum logic. *EPJ Quantum Technol*, 2.11, May 2015.

[15] Dennis Lucarelli. Quantum optimal control via gradient ascent in function space and the time-bandwidth quantum speed limit. *Phys. Rev. A*, 97:062346, Jun 2018.

[16] Jorge Nocedal. Updating quasi-newton matrices with limited storage. *Math. Comp.*, 35:773–782, 1980.

[17] F.Shampine P.BogackiL. A 3(2) pair of runge - kutta formulas. *Appl. Math. Lett.*, 2:321–325, 1989.

[18] JÃązsef SomlÃąi, Vladimir A. Kazakov, and David J. Tannor. Controlled dissociation of i2 via optical transitions between the x and b electronic states. *Chemical Physics*, 172(1):85 – 98, 1993.

[19] H. Jaddu and A. Majdalawi. An iterative method for solving constrained nonlinear optimal control problems using legendre polynomials. In *2015 10th Asian Control Conference (ASCC)*, pages 1–5, 2015.

[20] Michael H. Goerz et al. Krotov: A Python implementation of Krotov's method for quantum optimal control. *SciPost Phys.*, 7:80, 2019.

[21] Alexander J. McCaskey, Zachary P. Parks, Jacek Jakowski, Shirley V. Moore, Titus D. Morris, Travis S. Humble, and Raphael C. Pooser. Quantum chemistry as a benchmark for near-term quantum computers. *npj Quantum Information*, 5(1):99, 2019.

[22] E. F. Dumitrescu, A. J. McCaskey, G. Hagen, G. R. Jansen, T. D. Morris, T. Papenbrock, R. C. Pooser, D. J. Dean, and P. Lougovski. Cloud quantum computing of an atomic nucleus. *Phys. Rev. Lett.*, 120:210501, May 2018.

[23] N. Klco, E. F. Dumitrescu, A. J. McCaskey, T. D. Morris, R. C. Pooser, M. Sanz, E. Solano, P. Lougovski, and M. J. Savage. Quantum-classical computation of schwinger model dynamics using quantum computers. *Phys. Rev. A*, 98:032331, Sep 2018.

[24] David C. McKay, Christopher J. Wood, Sarah Sheldon, Jerry M. Chow, and Jay M. Gambetta. Efficient z gates for quantum computing. *Phys. Rev. A*, 96:022330, Aug 2017.

[25] Satish Balay et al. PETSc Web page. https://www.mcs.anl.gov/petsc, 2019.

[26] Tiffany M Mintz, Alexander J Mccaskey, Eugene F Dumitrescu, Shirley V Moore, Sarah Powers, and Pavel Lougovski. Qcor: A language extension specification for the heterogeneous quantum-classical model of computation. *arXiv preprint arXiv:1909.02457*, 2019.

[27] J.R. Johansson, P.D. Nation, and Franco Nori. Qutip 2: A python framework for the dynamics of open quantum systems. *Computer Physics Communications*, 184(4):1234 – 1240, 2013.

[28] Harrison Ball, Michael J. Biercuk, Andre Carvalho, Rajib Chakravorty, Jiayin Chen, Leonardo A. de Castro, Steven Gore, David Hover, Michael Hush, Per J. Liebermann, Robert Love, Kevin Nguyen, Viktor S. Perunicic, Harry J. Slatyer, Claire Edmunds, Virginia Frey, Cornelius Hempel, and Alistair Milne. Software tools for quantum control: Improving quantum computer performance through noise and error suppression, 2020.

[29] Roman Lutchyn, Leonid Glazman, and Anatoly Larkin. Quasiparticle decay rate of josephson charge qubit oscillations. *Phys. Rev. B*, 72:014517, Jul 2005.

[30] Matthew Ware, Blake R. Johnson, Jay M. Gambetta, Thomas A. Ohki, Jerry M. Chow, and B. L. T. Plourde. Cross-resonance interactions between superconducting qubits with variable detuning, 2019.

[31] Sudharshan S. Vazhkudai et al. The design, deployment, and evaluation of the coral pre-exascale systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC âĂŹ18. IEEE Press, 2018.