Software tool-set for automated quantum system identification and device bring up

Anurag Saha Roy1,2,3, Kevin Pack1,2, Nicolas Wittler1,2, and Shai Machnes1,3

1Peter Grünberg Institute – Quantum Computing Analytics (PGI 12), Forschungszentrum Jülich, D-52425 Jülich, Germany
2Theoretical Physics, Saarland University, 66123 Saarbrücken, Germany
3Qruise GmbH, 66113, Saarbrücken, Germany

Abstract—We present a software tool-set which combines the theoretical, optimal control view of quantum devices with the practical operation and characterization tasks required for quantum computing. In the same framework, we perform model-based simulations to create control schemes, calibrate these controls in a closed-loop with the device (or in this demo —by emulating the experimental process) and finally improve the system model through minimization of the mismatch between simulation and experiment, resulting in a digital twin of the device. The model based simulator is implemented using TensorFlow, for numeric efficiency, scalability and to make use of automatic differentiation, which enables gradient-based optimization for arbitrary models and control schemes. Optimizations are carried out with a collection of state-of-the-art algorithms originated in the field of machine learning. All of this comes with a user-friendly Qiskit interface, which allows end-users to easily simulate their quantum circuits on a high-fidelity differentiable physics simulator.

Index Terms—Quantum Optimal Control, Machine Learning, Quantum Computing, Superconducting Qubits, Benchmarking, Calibration, System Identification

I. INTRODUCTION

Development of quantum technology devices, in particular quantum computers is an enormously difficult task. Focusing for the moment on quantum computation, one can estimate NISQ (Near-term Intermediate-Scale Quantum Computers, [1]) devices become generally useful as we approach 100 qubits. But to be able to use this number of qubits in a meaningful way, a quantum circuit must also have sufficient depth. For most useful applications [2][3][4], a back of the envelope calculation thus suggests that an entangling gate error of $10^{-4}$ is an important step towards useful quantum computers, and progress towards that goal has been slow.

Whether aiming at error-corrected quantum computation or applications on NISQ devices, we posit one must significantly improve the methods by which we characterize quantum devices, and how we calibrate quantum controls. One common approach, known as ad-hoc [5], consists of deriving controls in simulation and subsequent fine-tuning in experiment. This scenario leaves a lot to be desired: The need for calibration proves the model we have of the system is inaccurate, and the final calibrated pulses may perform well, but we do not know why, as we have no model which explains the changes in controls which have occurred during calibration. To make matters worse, we have not learned anything about the system in the process, and we are not in a better position to improve system design in the next hardware iteration.

To resolve this unsatisfactory situation, the authors have developed a novel methodology, C3 — Control, Calibration and Characterization [6], which tightly integrates the calibration and characterization processes, to achieve better modeling of the system and hence better optimal control. This manuscript details the open-source (Apache 2.0) software implementation of this new methodology. And while there are several open-source software packages which perform either model-based optimal control, control calibration or limited system characterization [7], [8], [9], [10], [11], to our knowledge C3 is the first which combines all three in an integrated fashion, providing better insights into the sources of limitations in fidelities, and hence hints as to what should be addressed in the next hardware iteration.

The manuscript is organized as follows: We present the optimal control task of realizing a quantum computing gate-set on a single qubit in Section II. To demonstrate the full C3 procedure, we will consider two similar models, the simulation model, to which we have full descriptive access, and the blackbox model, which stands-in for a real experiment and thus only provides limited access by measurement. We also discuss the implementation of the underlying quantum simulation. In Section III we simulate the procedure of calibrating the gate-set by taking measurements on the blackbox. On the basis of the data taken during calibration, we perform model learning in Section IV, by simulating the calibration experiments on the simulation model to bring the two to agreement. Finally, we discuss the Qiskit [12] interface in Section V and give an overview of the software architecture in Section VI.

As a companion piece, an example Jupyter notebook that contains the code and reproduces the plots in this manuscript is accessible through this link - bit.ly/c3-full-example. This is released as part of the source code on Github at q-optimize/c3.

II. OPTIMAL CONTROL

When employing optimal control techniques in the practical operation of quantum devices, it is desirable to have access to a model of the device for analysis and the derivation of advanced control schemes. From a numerical perspective, algorithms that make use of the gradient of the goal function, e.g. L-BFGS [13], can speed up optimizations significantly compared...
to gradient-free algorithms [14]. Access to gradients of a quantum dynamics simulation is not trivial and thus procedures such as Krotov [15], GRAPE [16] and GOAT [14] require a specific formulation of the control problem in goal function and system descriptions. In the approach presented here, we make use of the numerics package Tensorflow, developed for machine learning applications, to provide both high compute performance and the access to automatic differentiation which lifts most limitations in formulating the problem.

A. The problem statement

For demonstration purposes, we will concern ourselves in this paper with the realization of a gate-set of \( \pi/2 \) rotations around the \( x \) and \( y \) axis of the Bloch sphere of a single qubit. This system is described by the Hamiltonian

\[
H(t)/\hbar = \omega b^\dagger b - \frac{\delta}{2}(b^\dagger b - 1)b^\dagger b + c(t)(b + b^\dagger),
\]

where \( b \) and \( b^\dagger \) are the bosonic ladder operators, \( \omega \) and \( \delta \) the qubit frequency and anharmonicity and \( c(t) \) the time-dependent control field.

The following snippet creates the qubit as an object in the C³ package.

```python
q1 = Qubit(
    name="q1",
    desc="Qubit 1",
    freq=Quantity(
        value=3e9,
        min_val=4.995e9,
        max_val=5.005e9,
        unit="Hz 2pi",
    ),
    anhar=Quantity(
        value=380e6,
        min_val=120e6,
        max_val=1280e6,
        unit="Hz 2pi",
    ),
)
```

Properties that will be optimized are instantiated from the Quantity class, which handles physical units and limits. Similarly, the drive is created by

```python
drive = Drive(
    name="d1",
    desc="Drive 1",
    comment="Drive line 1 on qubit 1",
    connected="[Q1]",
    hamiltonian_func=hamiltonians.x_drive,
)
```

where the connected parameter indicates this drive acts on the qubit created above. When the model is created with

```python
model = Model([q1, [drive]],)
```

the matrix representation of the operators in (1) are created and stored for later use. The operation \( U \) on the qubit, affected by the control field, is obtained by the formal solution of the Schrödinger equation

\[
U(T) = \exp\left(-\frac{i}{\hbar} \int_0^T H(t) dt \right).
\]

The control task now consists of finding the control fields \( c(t) \) which implement the gate-set

\[
\mathcal{G} = \{X_{\pi/2}, Y_{\pi/2}, X_{-\pi/2}, Y_{-\pi/2}\},
\]

Fig. 1. Figure and caption adapted from [6]: The process of simulating experimental procedure for signal processing and readout. The control function is specified by some function \( \varepsilon \) and specifies the line voltage \( u(t) \) by an arbitrary waveform generator (AWG) with limited bandwidth. Electrical properties of the setup, such as impedances, are expressed as a line transfer function \( \varphi \), resulting in a control field \( c(t) = \varphi[u(t)] \), as in Eq. (1).

\[
\mathcal{F}[U(t)] = \left| \frac{1}{\dim U} \text{Tr}\{U^\dagger(t)U_{\text{ideal}}\} \right|^2
\]

which we average over all operations of the gate-set to compute the goal function value.

B. Signal generation

The control field \( c(t) \), as seen by the qubit and written in the Hamiltonian, is typically different from the programmed field in signal generators of an experimental setting. The mathematical background is described in [6]. The process is implemented as a generator using a number of devices to realize the signal chain.

In the generator, a number of devices are arranged into signal chains by specifying their inputs and finally, their voltage signals are converted to energy (or frequency) scale to be put into the Hamiltonian.

C. Simulating quantum dynamics

The simulation of quantum dynamics is at the core of both the optimal control and model learning applications. To efficiently simulate a sequence of gates in the gate-set in a Markovian context, it makes sense to compute the Unitary representations of the gate-set only once and multiply them as
needed to create the sequence. We employ a technique similar to the virtual Z [17] phenomenon by multiplying the unitary representations of each gate by \( \exp(\omega_t T b^* b) \) which realigns the \( x, y \) axis of the qubit and the drive field.

### D. Optimization

To manipulate the control parameters, we use the `ParameterMap` that contains methods to collect optimizable parameters from both model and control components. In the example, we parametrize the control function with three parameters, the amplitude of the pulse, the DRAG parameter \( \delta \), the frequency offset which modulates the resonance and \( \delta \text{framechange} \), a phase which allows fine-tuning of the rotation.

The optimization is then carried out by the module

```python
opt = OptimalControl(
    dir_path=log_dir,
    fid_func=unitary_infid_set,
    fid_subspace="Q1",
    pmap=parameter_map,
    algorithm=lbgfs,
    options="maxfun" : 150,
    run_name="better_X90"
)
```

by specifying a fidelity function, including the subspace in the case of larger Hilbert space. Both the fidelity and the algorithm can be chosen from a library included in the package or from a custom user implementation with the same signature.

### III. Calibration

Having created an initial starting pulse through the usage of the tools in the optimization phase, we now continue with the refinement of said pulse in a closed feedback loop, the calibration step.

#### A. Automated Calibration

In traditional calibration, a portfolio of experiments executed in a specific sequence is used to gain knowledge of the different parameters of the quantum mechanical system and control pulse necessary to implement a desired control. This obviously does not scale to 100s of qubits. Henceforth, we advocate an automated approach to calibration where we try to minimize a goal function, which acts as a figure of merit for the desired operation. This approach is by no means new, and has been used successfully in the past to fine tune control pulses as can be seen in [18].

In this example, we decided to use ORBIT [19] as a goal function to minimize, and in the following chapter it will be shown how an ORBIT experiment can be set up in the \( \mathbb{C}^3 \) code. The task of calibration itself can be further simplified as a black box optimization, since we have to assume that our initial model of the system is not accurate. Here, we use CMA-ES, an advanced gradient free optimization algorithm.

#### B. ORBIT as Loss Function

The creation of the ORBIT sequence in \( \mathbb{C}^3 \) is done through a helper function, namely `single_length_RB`, contained in the `qt-utils` module.

```python
qt_utils.single_length_RB(
    RB_number=1, RB_length=5, target=0
)
```

The function has access to an internal list of decompositions of Clifford gates made up of available elemental operations. In this case, these elemental operations are the gates specified in \( \mathcal{G} \) given in Eq. 3. Which elemental operations are available usually depends on the experimental hardware. The function then returns a list of instruction names corresponding to the previously defined available instructions, which can be sent to the simulation as in this example or to a real quantum computer. We use the mean over the population probabilities of the desired final state as the final result. This concludes the evaluation of the ORBIT sequences for the simulation.

#### C. Setup of The Optimization Algorithm

After the preparation of the loss function, we continue with the setup of the optimizer algorithm. We rely for the minimization of the loss function on the covariance matrix adaptation evolution strategy or CMA-ES algorithm provided by the python package pycma [20]. Several relevant hyperparameters for the optimizer are also set up at this stage, the detailed description of which is beyond the scope of this manuscript.

```python
alg_options = {
    "popsize" : 10,
    "maxfevals" : 300,
    "init_point" : "True",
    "tolfun" : 0.01,
    "spread" : 0.1
}
```

### D. The Calibration Class

After all preparations are done, we can finally continue by setting up the actual calibration object, which takes the previously created ORBIT loss function and the hyperparameters of the optimizer as input.

```python
# Create a temporary directory to store logfiles, modify as needed
log_dir = "c3example_calibration"

opt = Calibration(
    dir_path=log_dir,
    run_name="ORBIT_cal",
    eval_func=ORBIT_wrapper,
    pmap=parameter_map,
    exp_right=simulation,
    algorithm=cmaes,
    options=alg_options
)
```

With the final preparations completed, we can now begin the calibration process simply by calling the method

```python
opt.optimize_controls()
```

Fig. 2 shows the final result of the calibration step. We can see how the values of the loss function decrease indicating a successful calibration, which results in an improvement in the final gate fidelity. It is noteworthy that the return value of the ORBIT loss function is not a gate fidelity or infidelity.
This means, that any improvement of the goal function cannot be directly expressed as improvement in gate fidelity. Hence, even small improvements in the goal function can lead to big improvements in the gate fidelity.

IV. MODEL LEARNING

Model based Optimal Control can only be as good as the model, which means there is always scope for further refining the parameters of your model and also enriching your existing model with additional components. An intuitive approach to solving the former challenge (of tuning your existing model with additional components) is to use Machine Learning techniques for extracting the model parameters from experimental data.

A. The Dataset

The dataset for learning model parameters can be generated from any experiment that can be reproduced in simulation. For this example, we consider the data (stored as a key-value pairs) from the ORBIT Calibration step, during which we obtained pairs of gate sequences and corresponding measurement results.

| TABLE I | DATASET FOR MODEL LEARNING |
|---------|-----------------------------|
| params | seqs | results | results_std | shots |
| [376.655 mV, -954.644 m, -50.333 MHz 2pi, -1.479 mrad] | (0.0129, 1000) | (0.0108, 1000) | (0.0129, 1000) | (0.0108, 1000) |

In Table I, params is the parameters of the pulse during that specific run and seqs stores the exact ORBIT sequence that was used; while results, results_std and shots tell us about the measurement outcome, standard deviation and number of shots respectively. The dataset also has the required metadata, such as the opt_map which contains information about the specific pulse parameters being optimized during that calibration run.

B. Model and Loss Function

We use the same model as described in the Optimal Control section since the goal of this step is to fine tune the model parameters by learning from the calibration data. We simulate the ORBIT sequence as contained in the dataset using this model and then compare the measurement outcomes of the simulation and the experiment. This comparison takes place in the form of a loss function[6] which is outlined in Eq. 5:

$$ f_3(\beta) = f_{ll}(D|\beta) = \frac{1}{2N} \sum_{n=1}^{N} \left[ \frac{(m_n - \bar{m}_n)^2}{\sigma_n} - 1 \right] $$

| that captures how well the model prediction $\bar{m}_n$, with standard deviation $\sigma_n$, agrees with the recorded values $m_n$, in the dataset $D$ for the model parameter values $\beta = (\omega_1, \delta_1, ...)$.

For the purpose of learning, we choose a subset of the entire data since simulating the whole dataset would be a significant computational challenge, as shown in the code snippet with accompanying explanatory comments.

```python
datafiles = {"orbit": DATAFILE_PATH} # path to the dataset
run_name = "simple_model_learning" # name of the optimization run
dir_path = "ml_logs" # path to save the learning logs
goal = "cmaes" # algorithm for learning
options = {
    "cmaes": {
        "popsize": 12,
        "init_point": "True",
        "stop_at_convergence": 10,
        "ftarget": 4,
        "spread": 0.05,
        "stop_at_sigma": 0.01,
    },
    "lbfgs": {
        "maxfun": 50, "disp": 0,
    }
} # options for the algorithms
sampling = "high_std" # how data points are chosen from the total dataset
batch_sizes = {"orbit": 2} # how many data points are chosen for learning
state_labels = {
    "orbit": [1, 2],
} # the excited states of the qubit model, in this case it is 3-level

To start the Model Learning process, an object of the relevant class is instantiated with the parameters defined in the previous code snippet, followed by invoking the canonical run() function on this object.

```
An interesting aspect to note here is the choice of the cma_pre_lbfgs algorithm for optimization. This algorithm is a mix of gradient-free (blackbox) global optimization and gradient-based local optimization. The CMA-ES undertakes a global optimization run and once it is approximately close to the global minima region, it switches over to a gradient based L-BFGS run to identify the final minima.

![Fig. 3. The Model Learning process for the Qubit Frequency (Q1-freq) —showing the initial search with CMA-ES followed by L-BFGS —as it converges to the true value of 5.001 GHz after starting from the assumed value of 5 GHz in approximately 200 iterations of the learning step.](image)

V. QISKIT INTERFACE

An important aspect of this software library is its intuitive and flexible interface which is achieved with the help of the Qiskit interface, the implementation of which is discussed below.

A. The qiskit backend in c3-toolset

This library includes a c3.qiskit submodule which allows users to have a drop-in replacement for running their existing qiskit circuits on the high-fidelity c3-toolset physics simulator. A minimum example for this is below:

```python
qc = QuantumCircuit(1)
qc.append(RX90pGate(), [0])
c3_provider = C3Provider()
c3_backend = c3_provider.get_backend("c3_qasm_physics_simulator")
c3_backend.set_c3_experiment(simulation)
c3_job_unopt = c3_backend.run(qc)
result_unopt = c3_job_unopt.result()
res_pops_unopt = result_unopt.data()['state_pops']
```

The histograms obtained from running this circuit, as shown in Fig. 4 reflect the un-optimized nature of the gates, along with some leakage to higher levels of the transmon. The same circuit when run after the Optimal Control step, returns result populations much closer to ideally expected values.

B. Pulse Level Control

The qiskit interface also gives more advanced users complete pulse level control, allowing them to execute routines such as pulse-level noise mitigation or replicate complete calibration runs on the digital twin. This pulse level access is provided through two different interfaces. Firstly, the user can supply additional options to the qiskit backend.run() call, which allows them to define various pulse parameters, such as the amplitude, frequency, detuning etc specific to this particular circuit. Otherwise, users can also provide a completely arbitrary piece-wise constant waveform as pairs of In-phase and Quadrature values; as a parameter for custom user-defined gates.

VI. SOFTWARE ARCHITECTURE

The c3-toolset software is based on three main building blocks (Model, Generator, Instructions) that form the foundation of all the modelling and calibration tasks, and depending on the use-case, some or all of these blocks might be useful, together with the tools necessary to perform optimization tasks.

A. Model

A theoretical Physics-based model of the Quantum Processing Unit. This is encapsulated by the Model class which consists of objects from the chip and tasks library. chip contains Hamiltonian models of different kinds of qubit realisations, along with their couplings while tasks let you perform common operations such as qubit initialisation or readout.

B. Generator

A digital twin of the electronic control stack associated with the Quantum Processing Unit. The Generator class contains the required encapsulation in the form of devices which help model the behaviour of the classical control electronics taking account of their imperfections and physical realisations.

C. Instruction

Once there is a software model for the QPU and the control electronics, one would need to define Instructions or operations to be perform on this device. Pulse shapes are described through a Envelope along with a Carrier, which are then wrapped up in the form of Instruction objects.
D. Parameter Map

The ParameterMap helps to obtain an optimizable vector of parameters from the various theoretical models previously defined. This allows for a simple interface to the optimization algorithms which are tasked with optimizing different sets of variables used to define some entity, e.g. optimizing pulse parameters by calibrating on hardware or providing an optimal gate-set through model-based quantum control.

E. Experiment

With the building blocks in place, we can bring them all together through an Experiment object that encapsulates the device model, the control signals, the instructions and the parameter map. Note that depending on the use only some of the blocks are essential when building the experiment.

F. Optimizers

At its core, c3-toolset is an optimization framework and all of the three steps - Open-Loop, Calibration and Model Learning can be defined as an optimization task. The optimizers contain classes that provide helpful encapsulation for these steps.

G. Libraries

The c3.libraries sub-module includes various helpful library of components that are used somewhat like Lego pieces when building the bigger blocks, e.g. hamiltonians for the chip present in the Model or envelopes defining a control pulse.

VII. OUTLOOK

We have described C³, an open-source software package for the characterization, control planning and calibration for quantum computers and other quantum technology devices. Since publishing the initial paper [6], significant developments have been undertaken by a number of contributors on github.com/q-optimize/c3 to facilitate application in several projects and physical platforms. Development is ongoing, focusing on improving performance and flexibility. In the near future we are planning on adding support for NV-centers, Rydberg atoms and trapped ion systems. C³ will also be the basis on which additional capabilities will be constructed. Co-design will allow one to simultaneously optimize model and control parameters, to find the model which will, when fabricated, provide the best performance. Sensitivity analysis can report on the distance between the model and the data in terms of standard deviations, and how this distance changes as a function of varying values of model parameters. Experiment Design will allow us to plan which data must be taken, i.e. which experiments must be performed. In order to efficiently reduce error bars of a specified model parameter. Finally, we note that much of what has been described in the C³ methodology is not actually specific to quantum systems, and may be adapted to other realms of physics using the suitable digital twin.

ACKNOWLEDGEMENTS

This work was supported by the European Commission through the OpenSuperQ project (Grant Nr. 820363), by the Germany Ministry of Science and Education (BMBF) through projects VERTICONS and DAQC (Grant Nr. 13N14872 and 13N15688) and the Helmholtz Validation Fund project "Qurise" (HVF-00096).

REFERENCES

[1] J. Preskill, “Quantum computing in the NISQ era and beyond,” Quantum, vol. 2, p. 79, aug 2018. [Online]. Available: https://doi.org/10.22331/q-2018-08-06-79
[2] S. Bravyi, D. Gosset, and R. König, “Quantum advantage with shallow circuits,” Science, vol. 362, no. 6412, pp. 308–311, 2018.
[3] S. Beauregard, “Circuit for shor’s algorithm using 2n+ 3 qubits,” arXiv preprint quant-ph/0205095, 2002.
[4] R. Suzuki, K. Mitarai, and K. Fujii, “Computational power of one-and two-dimensional dual-unitary quantum circuits,” Quantum, vol. 6, p. 631, 2022.
[5] D. J. Egger and F. K. Wilhelm, “Adaptive hybrid optimal quantum control for imprecisely characterized systems,” Physical review letters, vol. 112, no. 24, p. 240503, 2014.
[6] N. Wittler, F. Roy et al., “Integrated tool set for control, calibration, and characterization of quantum devices applied to superconducting qubits,” Physical Review Applied, vol. 15, no. 3, p. 034080, 2021.
[7] T. Alexander, N. Kanazawa et al., “Qiskit pulse: programming quantum computers through the cloud with pulses,” Quantum Science and Technology, vol. 5, no. 4, p. 044006, Aug 2020. [Online]. Available: http://dx.doi.org/10.1088/2058-9565/aba404
[8] H. Ball, M. J. Biercuk et al., “Software tools for quantum control: Improving quantum computer performance through noise and error suppression,” 2020.
[9] S. Bade, G. Winer et al., “qqa-platform/qqa-libs: Open sourced libraries,” May 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4769470
[10] B. Li, S. Ahmed et al., “Pulse-level noisy quantum circuits with qutip,” arXiv preprint arXiv:2105.09902, 2021.
[11] M. M. Müller, R. S. Said et al., “One decade of quantum optimal control in the chopped random basis,” arXiv preprint arXiv:2104.07687, 2021.
[12] S. G. Mitchell et al., “Qiskit: An open-source framework for quantum computing,” 2021.
[13] J. Nocedal, “Updating quasi-newton matrices with limited storage, Mathematics of Computation, vol. 35, no. 151, pp. 773–773, sep 1980. [Online]. Available: http://www.jstor.org/stable/2007977
[14] M. S. Anis, A. Mitchell et al., “Tunable, flexible, and efficient optimization of control pulses for practical qubits,” Phys. Rev. Lett., vol. 120, p. 150401, Apr 2018. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.120.150401
[15] D. M. Reich, M. Novoa et al., “Monotonically convergent optimization in quantum control using krotov’s method,” The Journal of Chemical Physics, vol. 153, no. 10, p. 104103, 2012. [Online]. Available: http://dx.doi.org/10.1063/1.3691827
[16] N. Khaneja, T. Reiss et al., “Optimal control of coupled spin dynamics: design of nmr pulse sequences by gradient ascent algorithms,” Journal of Magnetic Resonance, vol. 172, no. 2, pp. 296 – 305, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022238X04700696
[17] D. C. McKay, C. J. Wood et al., “Efficient z gates for quantum computing,” Physical Review A, vol. 96, no. 2, p. 022330, 2017.
[18] M. Werninghaus, D. J. Egger et al., “Leakage reduction in fast superconducting qubit gates via optimal control,” npj Quantum Information, vol. 7, no. 1, Jan 2021. [Online]. Available: https://doi.org/10.1038/s41534-020-00346-2
[19] J. Kelly, R. Barends et al., “Optimal Quantum Control Using Randomized Benchmarking,” Phys. Rev. Lett., vol. 112, p. 240504, Jun 2014. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.112.240504
[20] N. Hansen, Y. Akimoto, and P. Baudis, “CMA-ES/pycma on Github,” Zenodo, DOI:10.5281/zenodo.2559634, Feb. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2559634

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.