Final results

For this solution the decision was to use the Qiskit Pulse for the IBM open science prize 2021 challenge, this module allowed us to explore the possibilities of precise quantum control.

The best achieved fidelity for the full state tomography on the ibmq_jakarta were:

1. **0.8437** fidelity with zero noise extrapolation (fidelity without error mitigation techniques, but including measurement filter was 0.5278). The experiment was run on 20th March 2022 (job id: 623754a3d97bff37d8693572). The circuit was:
    - 11 Trotter steps,
    - RZX based circuit design,
    - having additional calibration runs using qiskit calibration routines,
    - including Carr-Purcel dynamical decoupling sequence with 4 pulses per sequence,
    - with a custom measurement classifier based on k-Nearest Neighbors algorithm and modified amplitudes of default measurement pulses,
    - a measurement filter was applied,
    - Zero noise extrapolation was done based on 6 full tomography scans with different noise scales.
2. **0.8260** and **0.8107** fidelity for a similar circuit as above but run on 26th March 2022 (job id: 623ec33ba2f72d3234dabd1a and 623ec0c974de0e833f85b645). All the settings were the same as above with the exception that no additional calibration routines were run (due to high queue on the machine) and 10 full tomography scans were used for the ZNE.

For the full details of the design circuit and results from other experiments please refer to the included paper.

Many other techniques to improve fidelity were tested but with negative or unclear results, more work and more experiments are needed to fully evaluate other techniques. A short summary of techniques which were not included in the final circuit but described in the full documents are:

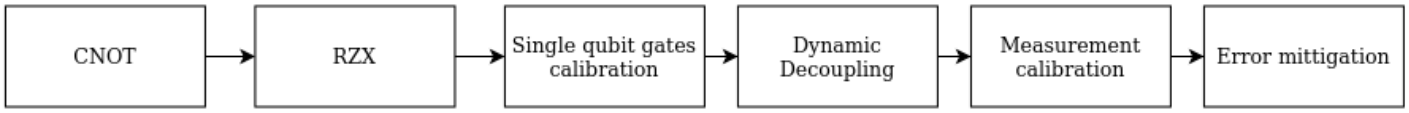
- [Promising] Full benchmark to decide on dynamical decoupling optimal pattern - due to limited time not enough circuits were run to show the optimal DD pattern (the final solution uses Carr-Purcel sequence).
- [Promising] RZX gate calibration - additional calibration routine for the RZX parameters. Few benchmarks were run but not included in the final solution due to limited availability of the quantum hardware.
- [Promising] Custom pulse schedules for off-resonance and amplitude error reduction (SCROFULOUS, CORPSE, etc.) - some of the results showed improvements of 2-3% for the fidelity, but calibration could be only performed manually and required few additional circuits to be run.
- [Promising - not explored] Qubits frequency calibration - additional calibration routine.
- [Promising - not explored] Probabilistic error cancellation - this technique should provide benefits but with the cost of additional circuits. This technique was not explored for this challenge.
- [Unclear] Trotter hamiltonian permutations - results on qiskit simulator did not show any improvements.
- [Too expensive] Piecewise constant optimization for pulse design - this technique requires a very high number of circuits to be run, which was not feasible for this challenge.
- [To specific] Algorithm mitigation techniques - To apply this technique with improvement to the final fidelity knowledge of the results in need (to choose the perfect trotter steps circuits for the extrapolation).

Abstract

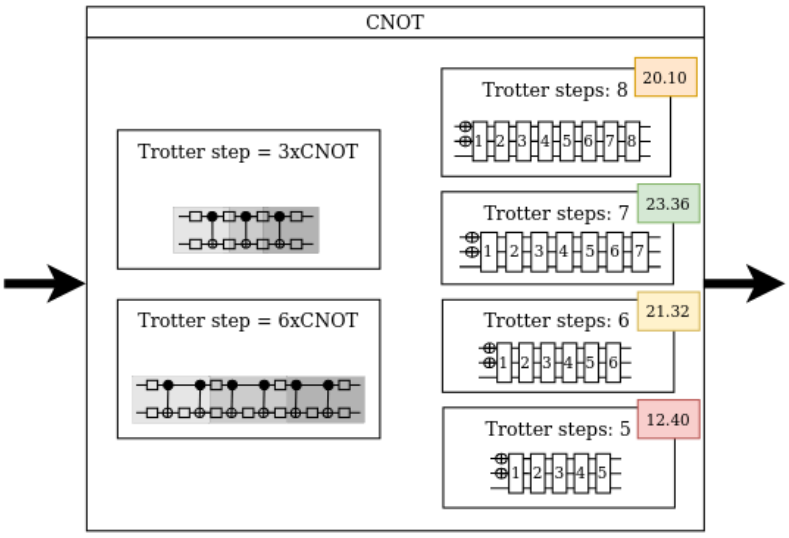
The goal of this paper is to explore error mitigation techniques for quantum simulation for the IBM open science 2021 challenge (*Qiskit-Community/open-Science-Prize-2021*, n.d.). We used the final state fidelity for the 3 qubit Ising model as our benchmark.
The hamiltonian of the challenge is as follows:

$$H_{\text{Heis}} = \sum_{\langle ij \rangle}^{N} J \left(\sigma_x^{(i)} \sigma_x^{(j)} + \sigma_y^{(i)} \sigma_y^{(j)} + \sigma_z^{(i)} \sigma_z^{(j)} \right).$$

We divided our implementation into following steps:

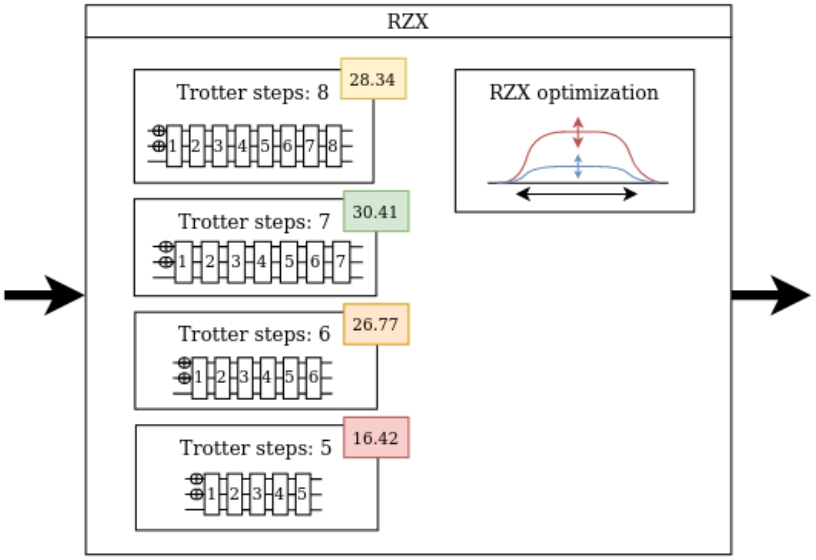


CNOT
For the initial exploration of the physics of quantum simulation we created a CNOT gate based model and benchmarked the impact of different number of trotter steps.



The naive 6 CNOT circuit achieved lower fidelity than the optimized 3 CNOT model. This is due to higher noise introduced by the additional 2 qubit gates.
7 Trotters steps perform the best, the number of trotter steps needs to be balanced so the digitization error is lower but not influenced by the circuit duration noise.
With CNOT based circuits we were able to achieve around **20-25%** fidelity.

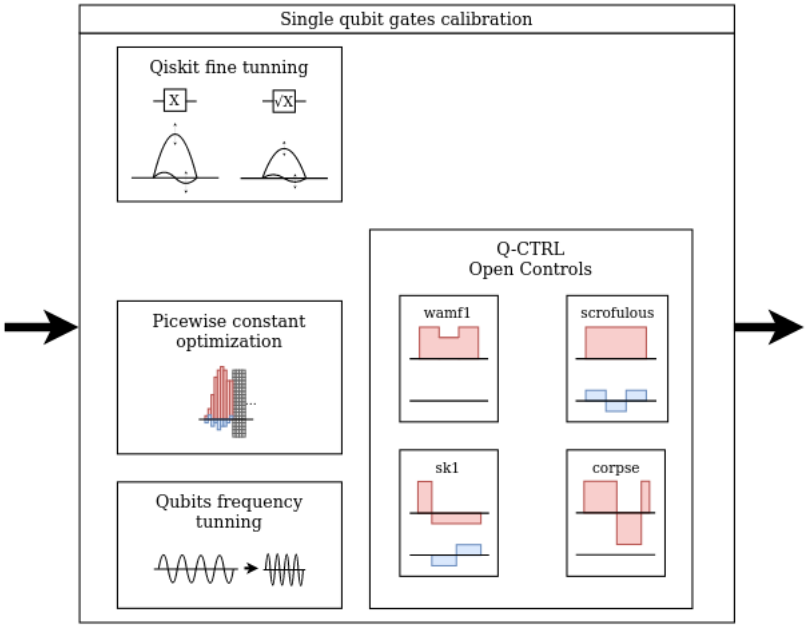
RZX
The next steps in the optimizalization was realized by replacing CNOT gates with RZX gates. The single 2 qubit Trotter step of Ising simulation can be modeled using only 3 RZX gates with single qubits operations. The RZX gates are constructed from CNOT gates by reducing the pulse's duration. This results in shortened overall circuit durations.



We were able to increase the final fidelity to around **30-32%**, using 7 trotter steps.

We additionally tested the RZX gate calibration. The default IBM RZX gates have quite high performance but fine tuning could improve the fidelity further. Due to limited availability of the quantum computer we were not able to properly perform RZX calibration so we skipped it from the final solution. The default qiskit RZX builder does use the default calibration for the X pulse, we create a code which modifies the RZX schedule by using the parameters from the calibrated X pulse.

Single qubit gates calibration.

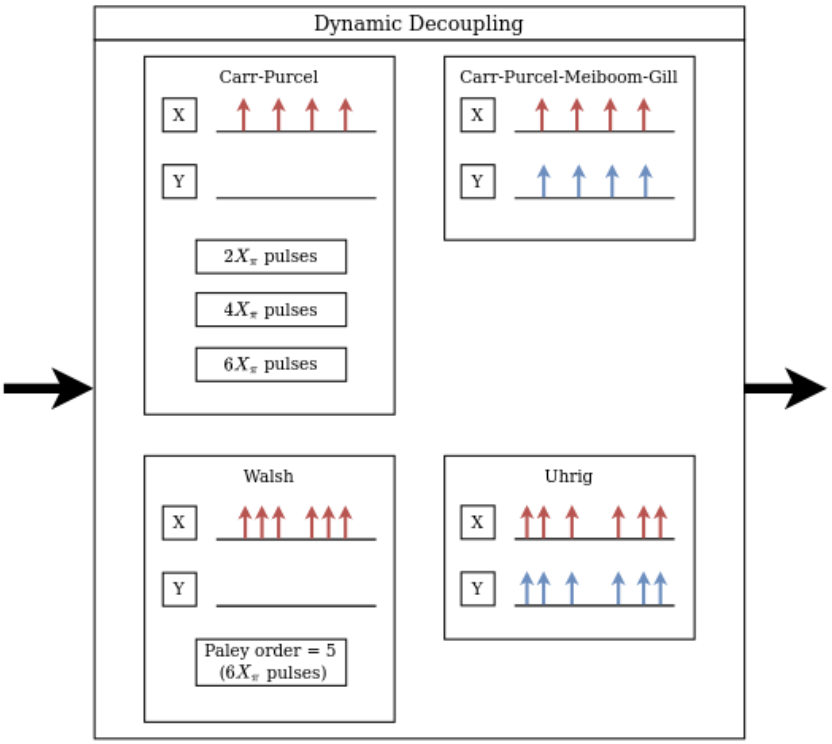


Qiskit provides built-in calibration routines for the single qubit gates - X and SX gates.
We used this routine to fine tune the gates with extended circuits count (from the default settings).
Fine tuning gives us around **1-3%** improvement in the overall simulation. This results depends on when the automatic calibration of the quantum computer was done. With longer wait time between automatic calibration running additional fine tuning calibration routine has the highest gains.
We decided to not perform qubit frequency calibration due to impact on the default ibm calibration. Changing qubit frequency without fine tuning results in lower fidelity.

Piecewise constant optimization techniques were checked and we ran circuits designed to create a custom X gate. But the performance of our gate never reached the default performance. The number of circuits needed to run this calibration is extremely high and not suitable for this challenge.

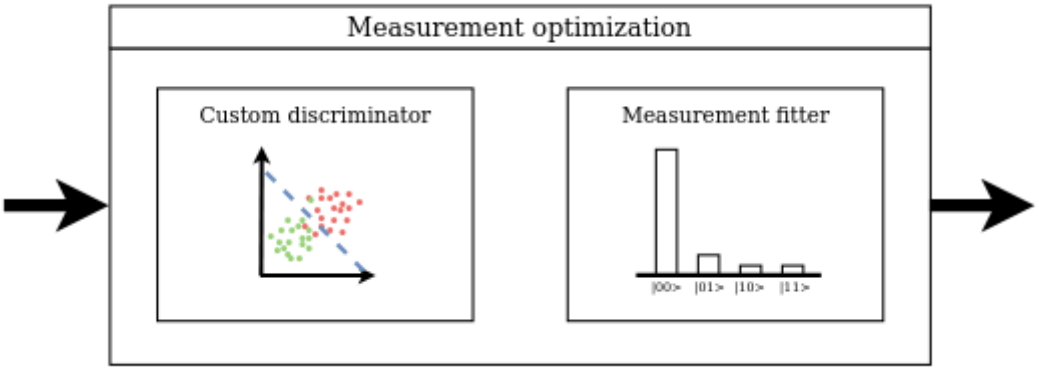
Using predefined pulse schedules we saw fidelity improvements in order of **1-3%**. The best results were seen with a SCROFULOUS pattern. The custom pulses required a few runs of manual calibration and with the limited availability of the quantum platform - we decided to skip this routine from the final solution.

Dynamic Decoupling
Applying X pulses can reduce the decoherence of the qubit, this is the idea behind Dynamic Decoupling (DD).



Many different DD patterns were designed, so we decided to benchmark the pattern in the Q-CTRL Open Control python package. Only X gate pulses were used due to unavailability of a calibrated Y pulse on IBM computers.
Qiskit provides a Pass Manager object which can apply DD patterns but we see problems with custom modified schedules which we created for the X gate calibration in RZX schedule and custom measurement classifier which will be explained later. We create our custom function which adds DD pulses into a preexisting schedule.
Dynamic decoupling provides a significant boost in the performance of the quantum circuit - **~3-5%**. Due to limited quantum hardware availability we were not able to properly benchmark all DD patterns to find the optimal one.
From only a few runs we decided to use the Carr-Purcell pattern with 4 pulses.

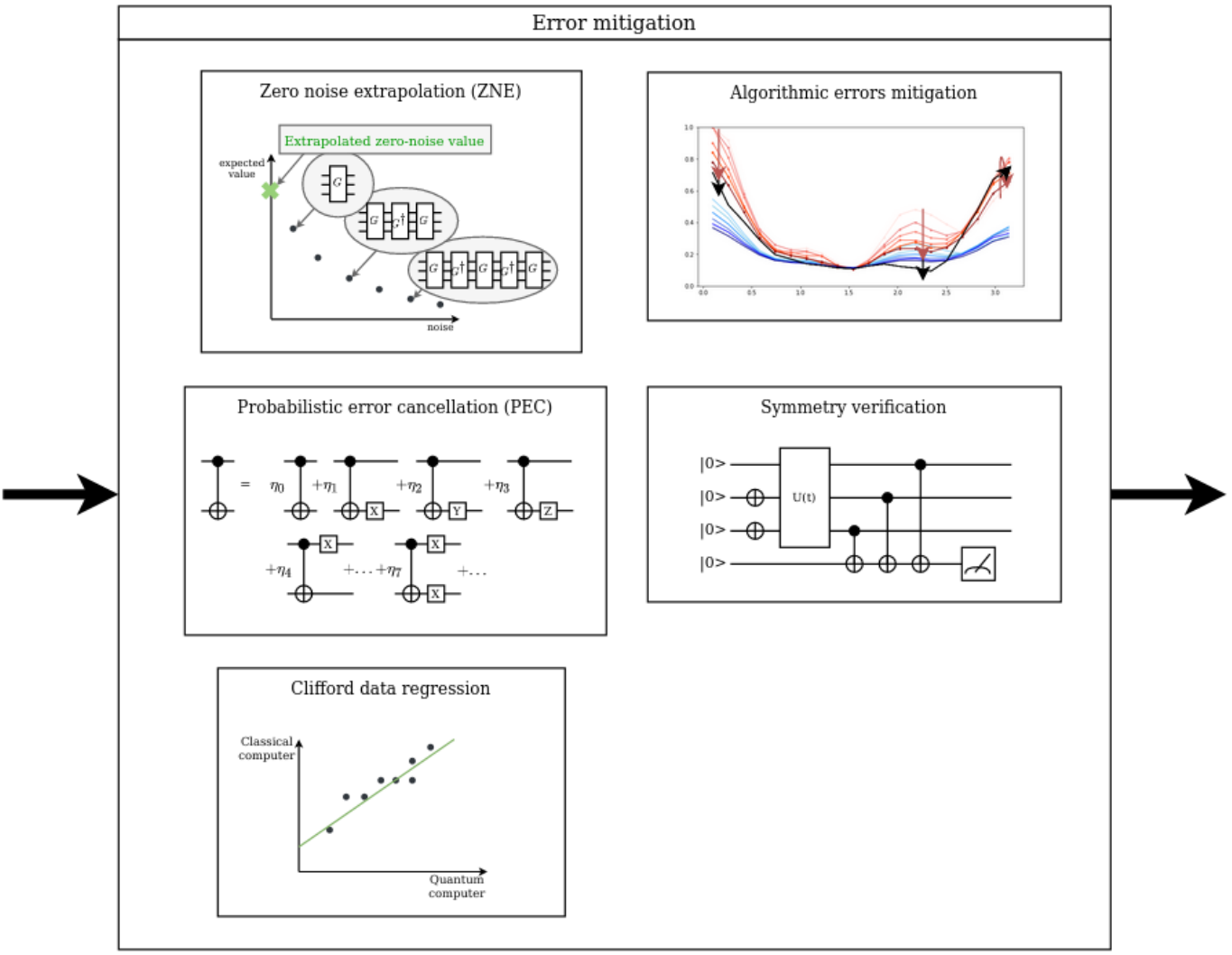
Measurement optimization
Measurement has very low fidelity in quantum circuits, but it often needs to be applied only once for the circuit.



We can improve the measurement fidelity by creating our own classifier, which based on the readout measurement classifies the final output into ground or excited state. We used the k-Nearest neighbor algorithm to improve the measurement outcome - we achieved an improvement of **1-2%** in the final fidelity. Other classifiers algorithms did not show significant improvements. Default readout fidelity for ibmq_jakarta is around 90% but our classifier improves the numbers to ~93%.

Qiskit provides a measurement fitter routine which performs full Hilbert space measurement and creates a filter which reduces the bias of the system. This routine brings high improvement in the final fidelity for a cost of 1 additional circuit. The improvement is around **2-5%**.

Error mitigation
As the final step in optimizing the quantum circuit we can apply quantum error mitigation (QEM) techniques - the goal of those techniques is to filter out the noise in post processing. Some of those techniques require many additional circuits to be run, this is the trade-off offered by QEM - more circuits will result in better final state estimation.



Due to limited availability of the quantum hardware we benchmarked techniques which require only a few additional circuits to be run.

The simplest technique is Zero noise extrapolation (ZNE), which can perform very well with as low as 4 additional circuits. We achieved very high improvement using ZNE reaching up to **84%** fidelity.
We created our own function to scale the noise of the circuit by applying identity operators in the ising steps. The different noise scaling steps (up to 10 steps) were combined in a single quantum execution routine, which results in faster execution.

Different extrapolation function were tested and we found out best performance from exponential function ($A\,e^{-x} + B$) with minimal values set to 0.
We benchmarked the ZNE extensively using quantum simulation to make sure that the choice of the noise and extrapolation step is generic and not fine tuned for this specific simulation.

Probabilistic error cancellation was not tested due to the significantly larger amount of circuits required.

Clifford data regression (CDR) technique was skipped for this challenge because 3 Qubit Ising simulation can be easily simulated on a classical computer, so the mitigation results can be too fine tuned to this specific problem.

Algorithmic error mitigation was benchmarked but we saw that this technique under estimates the errors. This could be improved with better selection of the trotter steps used for the extrapolation but we argue that fine tuning trotter steps is too specific to this problem, and cannot be easily generalized.

Symmetry verification can lead to further improvement of the final fidelity but we did not have any time to properly analyze and apply this technique.

About this document

The goal of this document is to explain techniques explored during our work on ibm open science 2021 challenge, this included the analysis/experiments which did not work.
Exploration of specific technique we divided in the following sections (in case section is not applicable to the technique it was skipped):

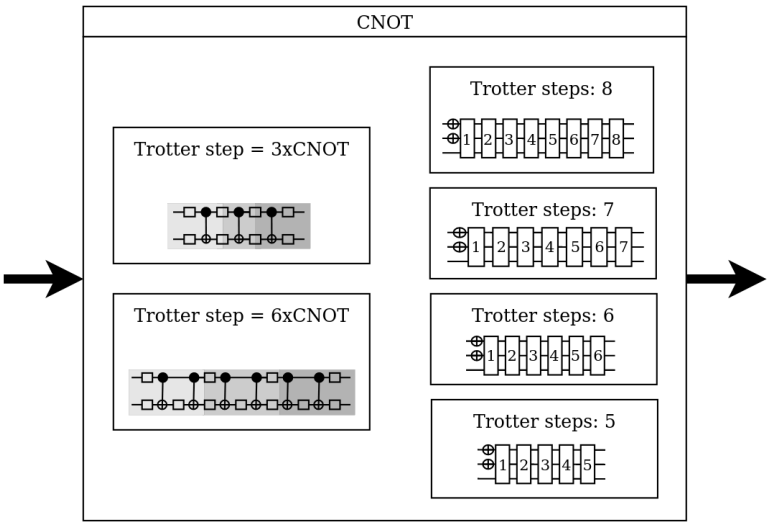
- Generic introduction
  Basic description of the technique used. We did not rely on mathematical description to simplify the document, for deeper analysis we provided the links to the recommended publication.

- Application to Ising simulation
  Here we explore specific settings applicable to the Ising simulation using trotterization. This assumes some knowledge about the problem that we want to analyze. We wanted to make sure that the analyzed technique is generic enough that it can be applied to bigger Ising simulation problems.

- Experiments
  Explanation of the code used to run the experiments on ibm quantum platform.
  Results of our benchmarks. The fidelity values shall not be compared between experiments if not explicitly specified because circuits were run with different default calibration at different dates (sometimes many days apart).
  In case we did not had enough time for fully exploring the technique we mention what could be further explored.

- Application to other project
  We want to answer the questions:
  1. Is this technique applicable to other problems / quantum circuits than the ising simulation?
  2. What are the trades-off?
  3. When can the technique be applied, what are the preconditions?

CNOT based circuit



Generic introduction

Simulation of quantum system is using Schrodinger equation in form $\frac{d|\psi>}{dt} = -iH|\psi>$, where the H is the Hamiltonian operator. Given a known Hamiltonian the problem we need to find the time evolution operator $U(t) = exp(-iHt)$ this operator is defining a unitary matrix, which can be run on a quantum computer to get the result at specified time (Tacchino et al., n.d.).

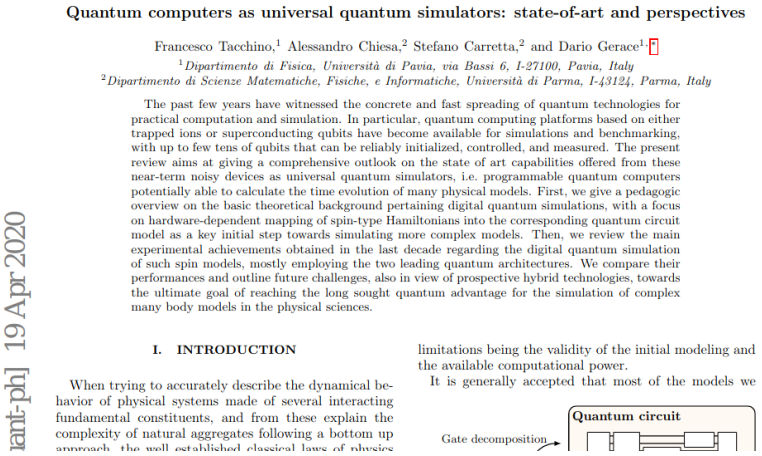
In generic term creating the time evolution operator requires $O(2^{2N})$ operations which increase exponentially (Lloyd). The number of circuits in case we have only involve few-body interaction can be reduce using so called Suzuki-Trotter decomposition, where the O is the digitization error:

$$e^{-i\sum_l H_l t} = \lim_{n\to\infty} \left(\prod_l e^{-iH_l t/n}\right)^n + O\left(\frac{t^2}{n}\right)$$

Papers for further reading

1. **Quantum computers as universal quantum simulators: state-of-art and perspectives**
   Francesco Tacchino, Alessandro Chiesa, Stefano Carretta and Dario Gerace
   https://arxiv.org/pdf/1907.03505.pdf

   This paper is one of the best introductions for people not familiar with Quantum simulation. It explores the basic topics and provides examples to see the techniques in action.



Quantum computers as universal quantum simulators: state-of-art and perspectives

Francesco Tacchino,[1] Alessandro Chiesa,[2] Stefano Carretta,[2] and Dario Gerace[1,*]
[1]Dipartimento di Fisica, Università di Pavia, via Bassi 6, I-27100, Pavia, Italy
[2]Dipartimento di Scienze Matematiche, Fisiche, e Informatiche, Università di Parma, I-43124, Parma, Italy

The past few years have witnessed the concrete and fast spreading of quantum technologies for practical computation and simulation. In particular, quantum computing platforms based on either trapped ions or superconducting qubits have become available for simulations and benchmarking, with up to few tens of qubits that can be reliably initialized, controlled, and measured. The present review aims at giving a comprehensive outlook on the state of art capabilities offered from these near-term noisy devices as universal quantum simulators, i.e. programmable quantum computers potentially able to calculate the time evolution of many physical models. First, we give a pedagogic overview on the basic theoretical background pertaining digital quantum simulations, with a focus on hardware-dependent mapping of spin-type Hamiltonians into the corresponding quantum circuit model as a key initial step towards simulating more complex models. Then, we review the main experimental achievements obtained in the last decade regarding the digital quantum simulation of such spin models, mostly employing the two leading quantum architectures. We compare their performances and outline future challenges, also in view of prospective hybrid technologies, towards the ultimate goal of reaching the long sought quantum advantage for the simulation of complex many body models in the physical sciences.

I. INTRODUCTION

When trying to accurately describe the dynamical behavior of physical systems made of several interacting fundamental constituents, and from these explain the complexity of natural aggregates following a bottom up approach, the well established classical laws of physics limitations being the validity of the initial modeling and the available computational power.

It is generally accepted that most of the models we

Application to Ising simulation

Ising simulation is describe using the following Hamiltonian:

$$H = \sum_{i,j=1,\,\alpha,\beta=x,y,z}^{N} h_{\alpha\beta,ij}\sigma_\alpha^{(i)}\sigma_\beta^{(j)} + \sum_{i=1,\,\alpha=x,y,z}^{N} h_{\alpha,i}\sigma_\alpha^{(i)}$$

In the simplest form using only 2 qubit the equation is:

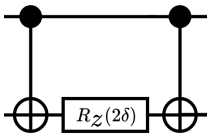
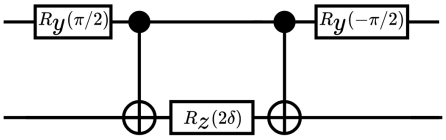
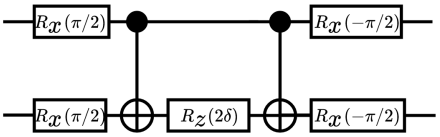
$$H = J \sum_{<i,j>} \sigma_x^{(i)}\sigma_x^{(j)} + \sigma_y^{(i)}\sigma_y^{(j)} + \sigma_z^{(i)}\sigma_z^{(j)}$$

We can split the time evolution operator using Suzuki-Trotter decomposition into form:

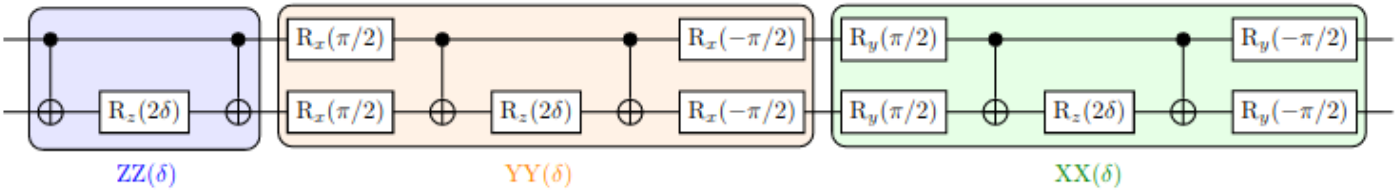
$$U(t) = e^{-i\delta(\sigma_x^{(1)}\sigma_x^{(2)}+\sigma_y^{(1)}\sigma_y^{(2)}+\sigma_z^{(1)}\sigma_z^{(2)})} = XX(\delta)YY(\delta)ZZ(\delta),$$

$$ZZ(\delta) = e^{-i\delta\sigma_z\otimes\sigma_z},\ XX(\delta) = e^{-i\delta\sigma_x\otimes\sigma_x},\ YY(\delta) = e^{-i\delta\sigma_y\otimes\sigma_y}$$

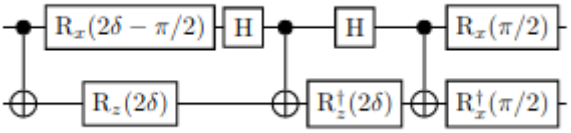
Those operators can be realized using the following quantum circuits:

$ZZ(\delta)$	$XX(\delta)$	$YY(\delta)$
		

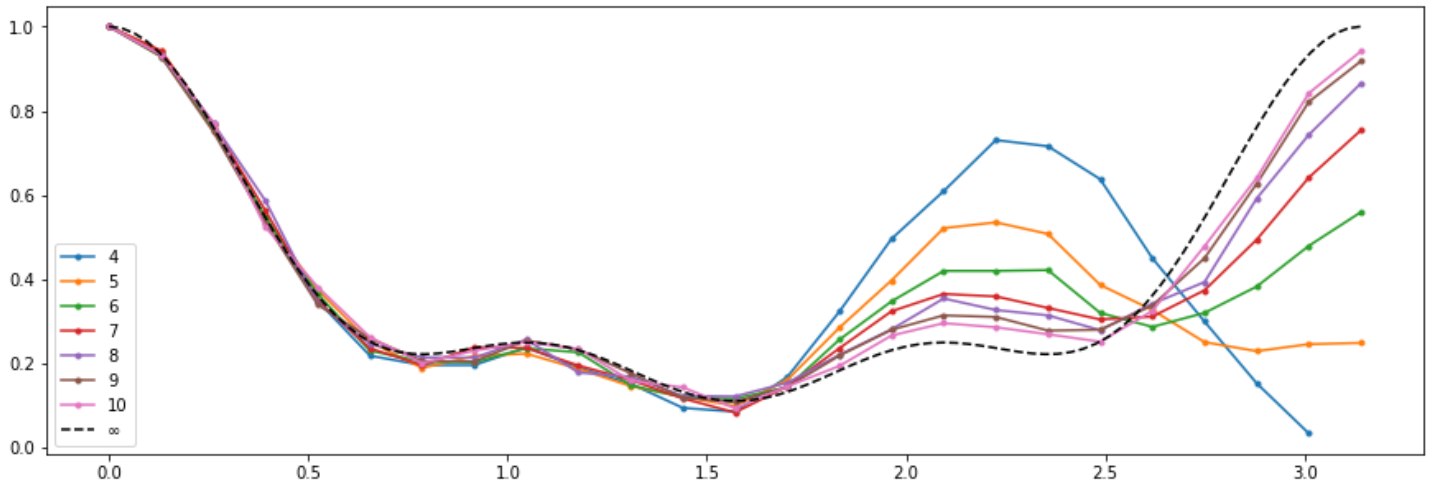
So the basic circuits for single trotter step will be as follows (Tacchino et al):



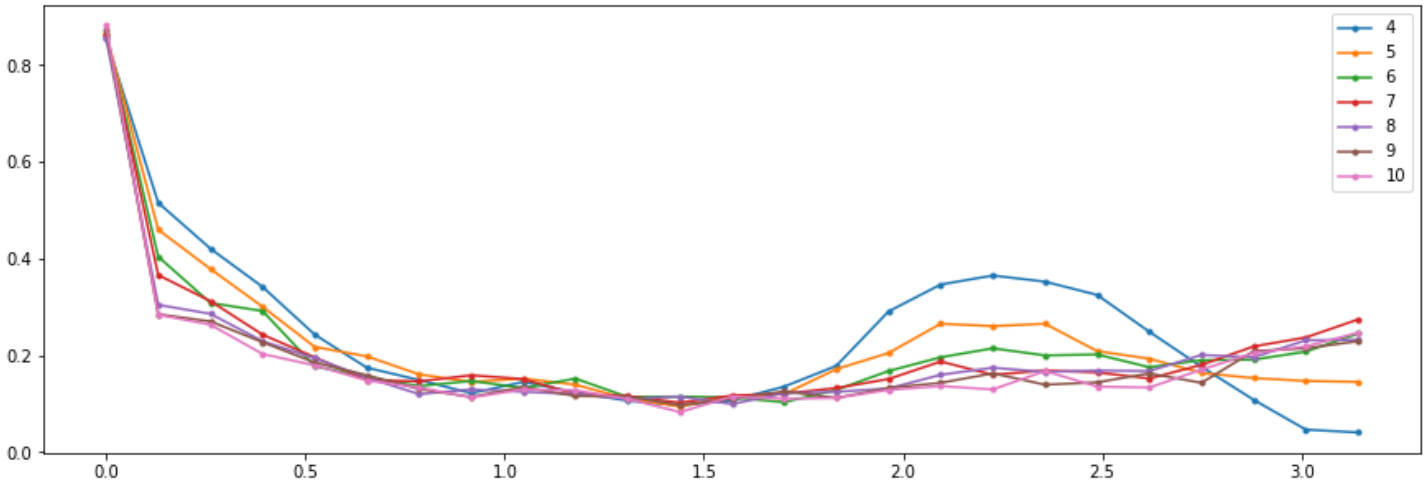
As described in (Vidal et al.) all 2 qubit systems can be modeled using only 3 CNOT gates. This allows us to simplify the circuit:



Quantum simulation using Suzuki-Trotter decomposition induces digitalization error $O(\frac{t^2}{n})$, whete t is the simulation time and n number of trotter steps.
We can see this by plotting the simulation of 3 qubit Ising system with different trotter steps (we measure the expectation value of the |110> state):



Due to the inherent noise of quantum circuits there will be an optimal value for the trotter steps where the accuracy of simulation is high enough and the noise is low.
Simulation with added noise looks as follows:



Experiments

We decided to execute the optimized 3 cnot model with Trotter steps ranging from 5 to 8. No other optimization techniques were added.

To generate the ising simulation circuits we use following code:

```
def ising_sim(trotter_steps, target_time, optimization_level, circ_type,
              calibrated=False):
```

This function generates the state tomography circuits for the Ising simulation. Trotter steps and target time are the input arguments used to generate the circuit. circ_type can be 'cnot' or 'rzx' calibration argument defines if a calibrated instance map shall be used.

```
    q0 = 1
    q1 = 3
    q2 = 5
    num_qubits = 3
```

Define the used qubits.

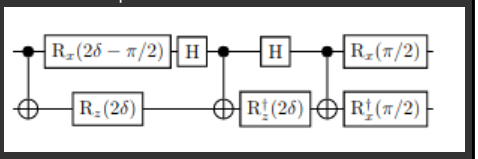
```
    Ising2Q_qr = QuantumRegister(2)
    Ising2Q_qc = QuantumCircuit(Ising2Q_qr, name='Ising_2Qubits')
```

We will split the full Ising simulation into smaller 2 qubits Ising circuits.

```
    if circ_type == 'cnot':
        t = Parameter('t')
```

t is δ parameters from the circuit image.

Create the quantum circuit:




```
        Ising2Q_qc.cnot(0,1)
        Ising2Q_qc.rx(2 * t - np.pi/2, 0)
        Ising2Q_qc.h(0)
        Ising2Q_qc.rz(2 * t, 1)
        Ising2Q_qc.cnot(0,1)
        Ising2Q_qc.h(0)
        Ising2Q_qc.rz(-2 * t, 1) # dagger
        Ising2Q_qc.cnot(0,1)
        Ising2Q_qc.rx(np.pi/2,0)
        Ising2Q_qc.rx(-np.pi/2,1) # dagger

    Ising2Q = Ising2Q_qc.to_instruction()

    qc = QuantumCircuit(7)
```

Create the final circuits with all qubits

```
    qc.x([q1,q2])  # DO NOT MODIFY (|q_5,q_3,q_1> = |110>)
```

Initiate the circuit in the |110> state

```
    for step in range(trotter_steps):
        qc.append(Ising2Q, [q0, q1])
        qc.append(Ising2Q, [q1, q2])
```

For every trotter step add the 2 Qubit Ising simulation - first time with 1st (q0) and 3rd (q1) qubits and next with 3nd (q1) and 5th (q2) qubits.

```
    inst_map_ = inst_map_cal if calibrated else inst_map_ncal
```

Select the global inst map calibration.

```
    if circ_type == 'cnot':
        if trotter_steps > 0:
            qc = qc.bind_parameters({t: target_time/trotter_steps})
```

For the cnot circuit bind the delta (t) parameter.

```
    qpt_qcs = state_tomography_circuits(qc, [q0, q1, q2])
```

Create the state tomography circuits.

```
    qpt_qcs = transpile(qpt_qcs, backend, optimization_level=optimization_level,
                        basis_gates=inst_map_.instructions)
```

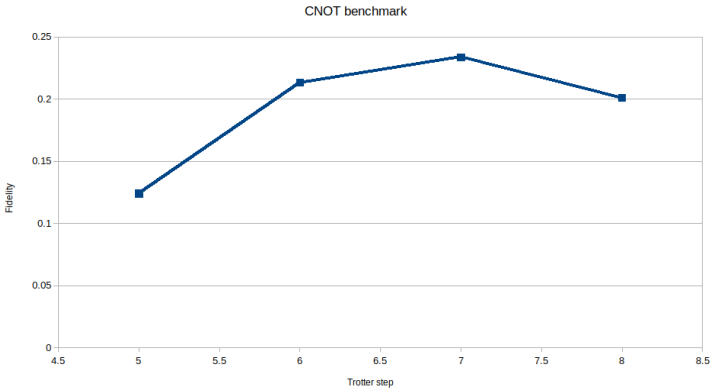
Transpile the circuit into quantum gates available on the device.

```
    return qpt_qcs, inst_map_
```

Return the tomography circuits and the "calibrated or not" instruction map for further usage.

Our results for the CNOT circuit on real hardware:

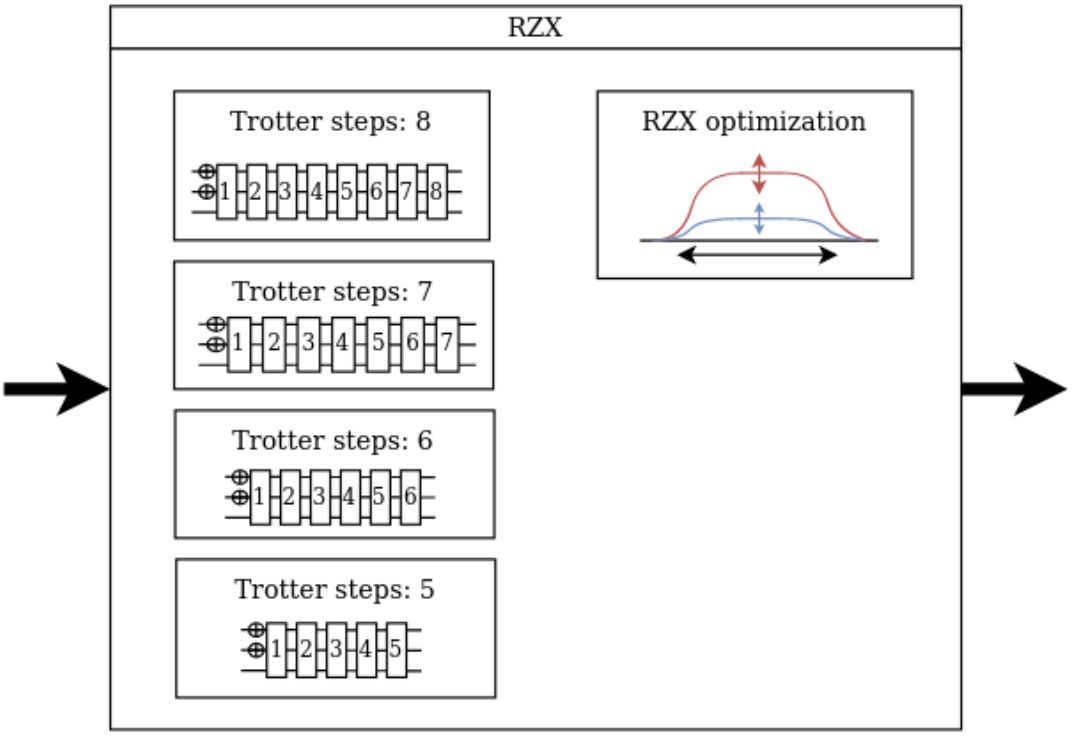
Trotter steps	Fidelity
8	0.201
7	0.2336
6	0.2132
5	0.124



Application to other problems

Technics presented in this section are very generic for the Ising simulations but many other techniques were developed for quantum simulation like Quantum Walk, Qubitization, Fractional Query, etc.
Higher orders of the Trotter formula can bring improvement in specific cases (Childs et al.).
The 3 CNOT decomposition can be very useful for other quantum circuits.

RZX based circuit



Generic introduction

The CNOT gate is not native to superconducting quantum computers. The interaction between 2 qubits is modeled by a so-called cross resonance gate, which Hamiltonian is in the form (Satoh et al):

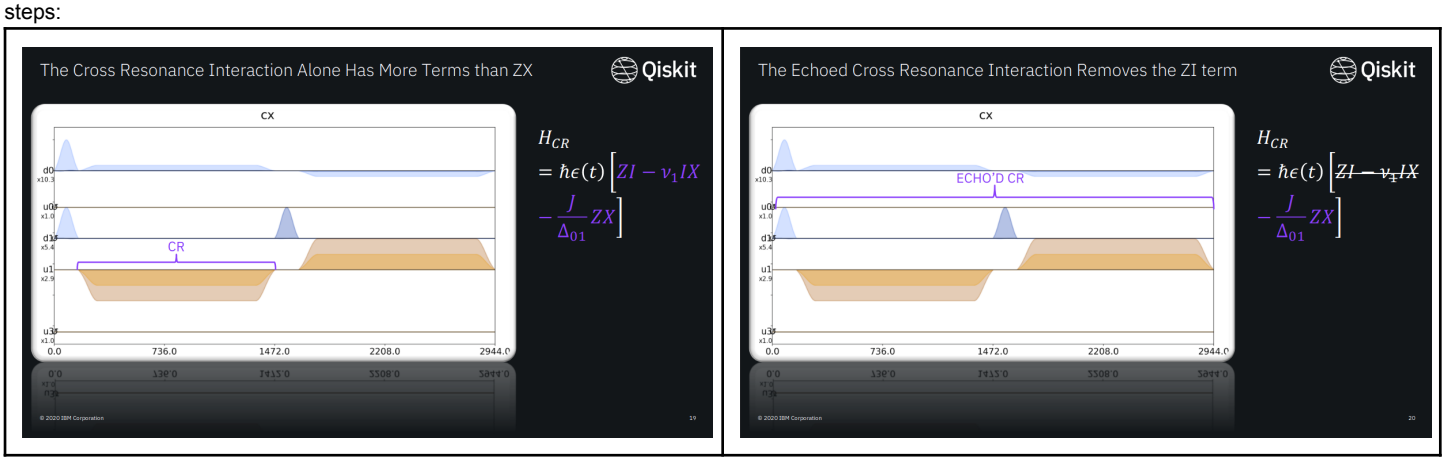
$$H_{CR} = \sum_{P=I,X,Y,Z} \frac{\omega_{ZP}(A,\phi)}{2} Z \otimes P + \sum_{Q=X,Y,Z} \frac{\omega_{IQ}(A,\phi)}{2} I \otimes Q,$$

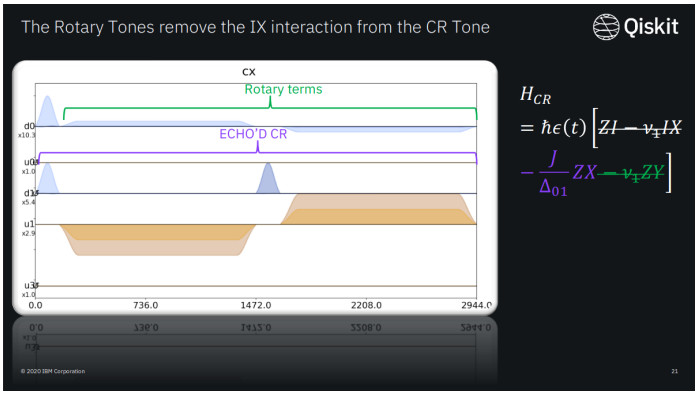
where ω_{ZP} and ω_{IQ} represent the interaction strength, which are the functions of the amplitude A and the phase ϕ of the microwave pulse.

The RZX gate is defined by the hamiltonian:

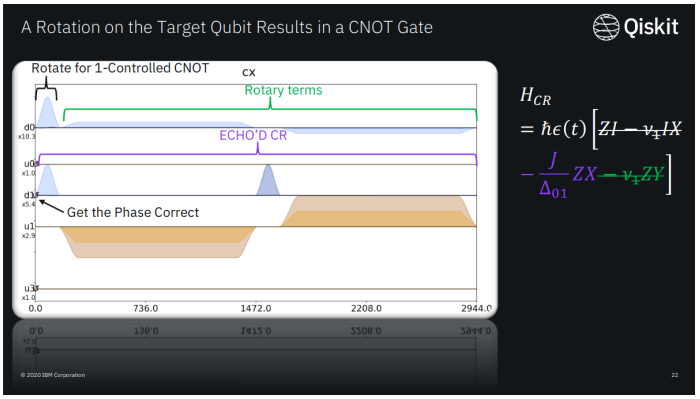
$$H_{ZX} = \frac{\omega_{ZX}(A,\phi)}{2} Z \otimes X$$

This is basically the cross resonance gate but with some terms canceled out (Earnest-Noble), the get the pure gate we need to apply some cancelation steps:

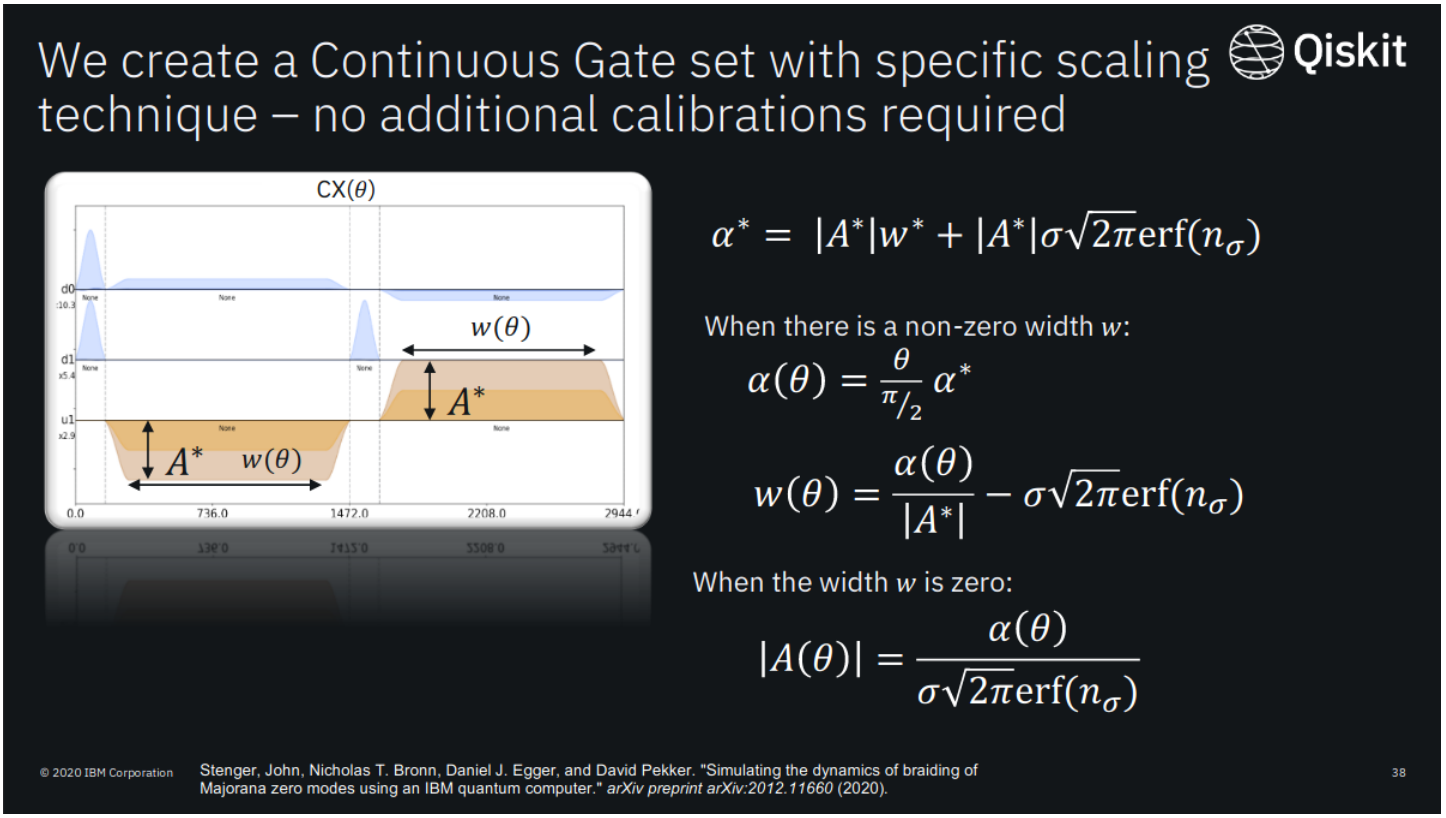




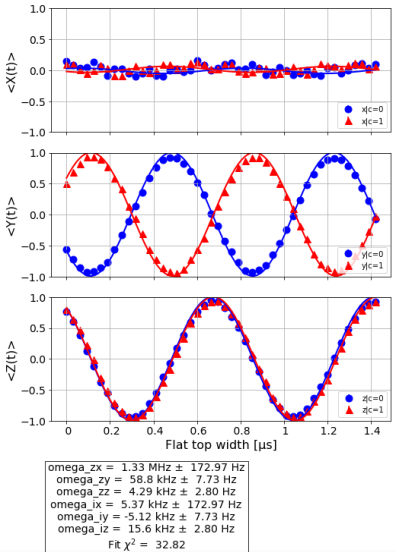
From the cross resonance gate we can get the CNOT gate by adding a rotation on the target qubit:



We can create the RZX gate by scaling the control pulse based on the CNOT calibrations. This technique give us a well calibrated RZX gate out of the box:



The CNOT and RZX gates can be further fine tuned. (Malekakhlagh and Magesan) (Xu et al).
Qiskit provides a Hamiltonian experiment `EchoedCrossResonanceHamiltonian()`(“`EchoedCrossResonanceHamiltonian – Qiskit Experiments 0.2.0 documentation`”) to visualize the performance of the cross resonance gate. Which returns the result in the plot of X, Y and Z axis measurements with the control set to ground or excited state.



Papers for further reading

1. **Pulse-efficient circuit transpilation for quantum applications on cross-resonance-based hardware**
   Nathan Earnest Caroline Tornow and Daniel J. Egger
   https://arxiv.org/pdf/2105.01063.pdf

   Paper explaining the cross resonance gate on superconductive quantum computers and its application to create other custom 2 qubit gates.

Pulse-efficient circuit transpilation for quantum applications on cross-resonance-based hardware

Nathan Earnest,[1] Caroline Tornow,[2,3] and Daniel J. Egger[3,*]
[1]*IBM Quantum – IBM T.J. Watson Research Center, Yorktown Heights, New York 10598, USA*
[2]*Institute for Theoretical Physics, ETH Zurich, Switzerland*
[3]*IBM Quantum – IBM Research – Zurich, Säumerstrasse 4, 8803 Rüschlikon, Switzerland*
(Dated: May 4, 2021)

We show a pulse-efficient circuit transpilation framework for noisy quantum hardware. This is achieved by scaling cross-resonance pulses and exposing each pulse as a gate to remove redundant single-qubit operations with the transpiler. Crucially, no additional calibration is needed to yield better results than a CNOT-based transpilation. This pulse-efficient circuit transpilation therefore enables a better usage of the finite coherence time without requiring knowledge of pulse-level details from the user. As demonstration, we realize a continuous family of cross-resonance-based gates for $SU(4)$ by leveraging Cartan's decomposition. We measure the benefits of a pulse-efficient circuit transpilation with process tomography and observe up to a 50% error reduction in the fidelity of $R_{ZZ}(\theta)$ and arbitrary $SU(4)$ gates on IBM Quantum devices. We apply this framework for quantum applications by running circuits of the Quantum Approximate Optimization Algorithm applied to MAXCUT. For an 11 qubit non-hardware native graph, our methodology reduces the overall schedule duration by up to 52% and errors by up to 38%.

I. INTRODUCTION

Quantum computers have the potential to impact a broad range of disciplines such as quantum chemistry [1], finance [2, 3], optimization [4, 5], and machine learn-

the set of two-qubit gates [36–38]. Such gates can in turn generate other multi-qubit gates more effectively than when the CNOT gate is the only two-qubit gate available [37]. However, creating these gates comes at the expense of additional calibration which is often im-

2. **Procedure for systematically tuning up crosstalk in the cross resonance gate**
   Sarah Sheldon, Easwar Magesan, Jerry M. Chow, and Jay M. Gambetta
   https://arxiv.org/pdf/1603.04821.pdf

   This paper explains the idea behind cross resonance gate calibration.

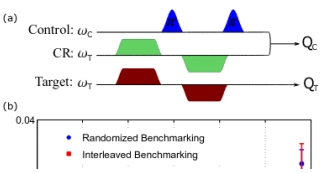
Procedure for systematically tuning up crosstalk in the cross resonance gate

Sarah Sheldon, Easwar Magesan, Jerry M. Chow, and Jay M. Gambetta
IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA
(Dated: March 16, 2016)

We present improvements in both theoretical understanding and experimental implementation of the cross resonance (CR) gate that have led to shorter two-qubit gate times and interleaved randomized benchmarking fidelities exceeding 99%. The CR gate is an all-microwave two-qubit gate offers that does not require tunability and is therefore well suited to quantum computing architectures based on 2D superconducting qubits. The performance of the gate has previously been hindered by long gate times and fidelities averaging 94-96%. We have developed a calibration procedure that accurately measures the full CR Hamiltonian. The resulting measurements agree with theoretical analysis of the gate and also elucidate the error terms that have previously limited the gate fidelity. The increase in fidelity that we have achieved was accomplished by introducing a second microwave drive tone on the target qubit to cancel unwanted components of the CR Hamiltonian.

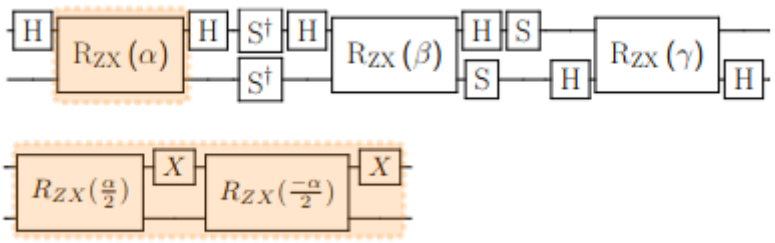
The cross resonance (CR) gate is an entangling gate for superconducting qubits that uses only microwave control [1, 2] and has been the standard for multi-qubit experiments in superconducting architectures using fixed-frequency transmon qubits [3, 4]. Superconducting qubits arranged with shared quantum buses [5] allow qubit networks to be designed with any desired connectivity. This flexiblity of design also translates into a flexibility of control and many choices in entangling gate implementations. The CR gate is one choice of two-qubit gate that uses only microwave control, as opposed to using magnetic flux drives to tune two qubits into a specific resonance condition to entangle, as in the controlled-Phase gate [6, 7], or to tune a coupler directly [8–11]. The CR gate requires a small static coupling of the qubit pair that slightly hybridizes the combined system and one additional microwave drive. The relatively low overhead of the CR scheme (the additional control line is combined

and novel tune-up procedure that reduce the gate time by a factor of two with corresponding fidelities over 99%. Our increased understanding of the CR Hamiltonian expands upon previous studies, which have primarily used a simple qubit model [1, 12]. We show that such models are incomplete and do not fully capture the dynamics of the complete two transmon qubit system.



Application to Ising simulation

Similar to 3 CNOT decomposition we have an algorithm (Tucci) to design any 2 qubit algorithm using 3 RZX gates (Earnest-Noble). This results in the following optimized circuit:



Due to the shorter duration of RZX gate, in comparison to CNOT, this circuit shall have better performance due to lower overall duration.

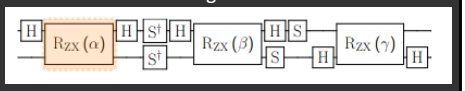
Experiments

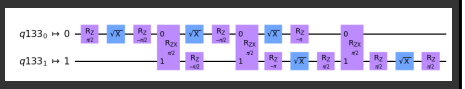
We modified the `ising_sim()` function to include the RZX gate circuit.

```
def ising_sim(trotter_steps, target_time, optimization_level, circ_type,
              calibrated=False):
    ...

    Ising2Q_qr = QuantumRegister(2)
    Ising2Q_qc = QuantumCircuit(Ising2Q_qr, name='Ising_2Qubits')

    if circ_type == 'cnot':
        ...
    if circ_type == 'rzx':
        rzx_gate = Gate('rzx_', 2, [])

        Ising2Q_qc.rz(np.pi/2, 0)
        Ising2Q_qc.sx(0)
        Ising2Q_qc.rz(-np.pi/2, 0)
        Ising2Q_qc.append(rzx_gate, [0, 1])
        Ising2Q_qc.sx(0)
        Ising2Q_qc.rz(-np.pi/2, 0) # Sdg
        Ising2Q_qc.rz(-np.pi/2, 1) # Sdg
        Ising2Q_qc.append(rzx_gate, [0, 1])
        Ising2Q_qc.sx(0)
        Ising2Q_qc.rz(-np.pi, 0)
        Ising2Q_qc.rz(-np.pi, 1)
        Ising2Q_qc.sx(1)
        Ising2Q_qc.rz(np.pi/2, 1)
        Ising2Q_qc.append(rzx_gate, [0, 1])
        Ising2Q_qc.h(1)

    Ising2Q = Ising2Q_qc.to_instruction()

    qc = QuantumCircuit(7)
    qc.x([q1,q2])  # DO NOT MODIFY (|q_5,q_3,q_1> = |110>)
    for step in range(trotter_steps):
        qc.append(Ising2Q, [q0, q1])
        qc.append(Ising2Q, [q1, q2])

    inst_map_ = inst_map_cal if calibrated else inst_map_ncal

    if circ_type == 'rzx':
        rzx_q0q1_sched = rzx_cal((q0,q1), 2*target_time/trotter_steps, x_cal=True)
        rzx_q1q2_sched = rzx_cal((q1,q2), 2*target_time/trotter_steps, x_cal=True)

        qc.add_calibration('rzx_', (q0,q1), rzx_q0q1_sched)
        qc.add_calibration('rzx_', (q1,q2), rzx_q1q2_sched)

        inst_map_.add('rzx_', (q0,q1), rzx_q0q1_sched)
        inst_map_.add('rzx_', (q1,q2), rzx_q1q2_sched)

    if circ_type == 'cnot':
        ...

    qpt_qcs = state_tomography_circuits(qc, [q0, q1, q2])
    qpt_qcs = transpile(qpt_qcs, backend, optimization_level=optimization_level,
                        basis_gates=inst_map_.instructions)
    return qpt_qcs, inst_map_
```

See the CNOT based circuit Experiments chapter for full description of this function.

Create a custom rzx gate.





The difference between those two pictures comes from the qiskit transpiler. Those circuits are equivalent.

For the rzx gate we need to add a custom schedule from rzx_cal() custom function. We create the schedule for the rzx qubit 1 & 2 and rzx qubit 2 & 3.

qc.add_calibration() applies our schedule to the custom Gate.

We add it to the instant calibration map for the transpiler.

The RZX calibration is generated using the following code:

```
RzxCal = RZXCalibrationBuilder(backend=backend, instruction_schedule_map=inst_map_cal)
```

Qiskit provides a Builder object which can be used to create custom RZX gates.

```
def get_closest_multiple_of_16(num):
    return int(num + 8 ) - (int(num + 8 ) % 16)
```

IBM systems accept duration in multiples of 16, so we create a helper function.

```
def rzx_cal(qubits, theta, amp_mul=1.0, amp=None, amp_t=None, duration=None,
            sigma=None, x_cal=False, waveform=None):
```

This function returns the RZX schedule based on the input angle (theta), additional parameters can be provided for custom calibration.

```
    theta_ = Instruction('theta', 7, 7, params=[theta])
    def_rzx_cal = RzxCal.get_calibration(theta_, qubits)
```

Create the RZX angle and generate the rzx schedule.

```
    amp_dr_, amp_ctr_, sigma_, width_, duration_, ctr_ch_, dr_ch_ = None, None, None, \
                                                                    None, None, None, None
    x_amp_, x_beta_, x_sigma_, x_duration_, x_dr_ch_ = None, None, None, None, None
```

We will extract the default algorithms from the rzx schedule

```
    for instr_ in def_rzx_cal.instructions:
        if isinstance(instr_[1], pulse.instructions.play.Play):
            if isinstance(instr_[1].operands[0], pulse.GaussianSquare):
                  if isinstance(instr_[1].channel, pulse.channels.ControlChannel):
                   # CR control
                   if amp_ctr_ is None:
                       amp_ctr_ = instr_[1].operands[0].amp
                       ctr_ch_ = instr_[1].channel.index
               else:
                   #CR drive
                   if amp_dr_ is None:
                       amp_dr_ = instr_[1].operands[0].amp
                       sigma_ = instr_[1].operands[0].sigma
                       duration_ = instr_[1].operands[0].duration
                       width_ = instr_[1].operands[0].width
                       dr_ch_ = instr_[1].channel.index

            if isinstance(instr_[1].operands[0], pulse.Drag):
                # X pulses
                if x_amp_ is None:
                    x_duration_ = instr_[1].operands[0].duration
                    x_amp_ = instr_[1].operands[0].amp
                    x_beta_ = instr_[1].operands[0].beta
                    x_dr_ch_ = instr_[1].channel.index
                    x_sigma_ = instr_[1].operands[0].sigma
```

Go through all RZX instructions and look for the GaussianSquare pulse.

If the pulse is acting on ControlChannel,

For the Control Channel,
if the control channel parameters are not yet set,
copy the pulse amplitude and the channel.

For the Drive Channel,
if the drive channel parameters are not yet set,
copy all the pulse parameters to local objects.

If the pulse is Drag - this is the X gates,
copy the data x pulse data if the parameters are not not yet set.

```
    if amp is not None:
        if amp[0] is not None:
            amp_ctr_ = amp[0]+amp_ctr_.imag*1j
        if amp[1] is not None:
            amp_ctr_ = amp_ctr_.real+amp[1]
    if amp_t is not None:
        if amp_t[0] is not None:
            amp_dr_ = amp_t[0]+amp_dr_.imag*1j
        if amp_t[1] is not None:
            amp_dr_ = amp_dr_.real+amp_t[1]
```

If the input amplitude is provided, update the local pulse parameters with the input amplitude.

```
    if duration is not None:
        duration = get_closest_multiple_of_16(duration)
        width_ = duration/duration_ * width_
        duration_ = duration
```

If the duration is provided, update the local pulse parameters - for the duration set it to the closest multiple of 16. Calculated pulse width based on the new duration.

```
    if sigma is not None:
        sigma_ = sigma
```

If sigma is provided, update the local pulse parameters.

```
    if x_cal:
        x_cal = inst_map_cal.get('x', x_dr_ch_)
        x_amp_, x_beta_, x_sigma_, x_duration_ = x_cal.instructions[0][1].operands[0].amp,\
                                        x_cal.instructions[0][1].operands[0].beta,\
                                        x_cal.instructions[0][1].operands[0].sigma,\
                                        x_cal.instructions[0][1].operands[0].duration
```

If the x gate calibration is provided, copy its parameters to the local object, by default RzxCal.get_calibration() uses the ibmq machine default X gate calibrations.

```
    amp_ctr_ *= amp_mul
    amp_dr_ *= amp_mul
```

Update the pulse amplitude based on the amplitude multiplier provided as input.

```
    rzx_cal = Schedule(
              (0, Play(GaussianSquare(duration=duration_, amp=amp_dr_, sigma=sigma_,
                                      width=width_), DriveChannel(dr_ch_))),
              (0, Play(GaussianSquare(duration=duration_, amp=amp_ctr_, sigma=sigma_,
                                      width=width_), ControlChannel(ctr_ch_))),
              (duration_, Play(Drag(duration=x_duration_, amp=x_amp_, sigma=x_sigma_,
                                    beta=x_beta_), DriveChannel(x_dr_ch_), name='Xp_d5')),
```

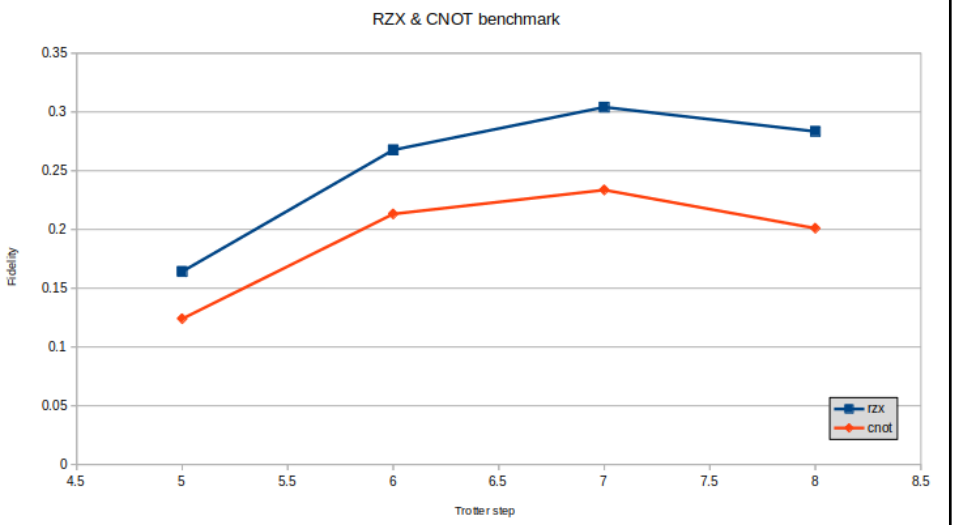
Create a new RZX schedule with the local parameters.

```
            (duration_+x_duration_, Play(GaussianSquare(duration=duration_,
                                        amp=-amp_dr_, sigma=sigma_, width=width_),
                                    DriveChannel(dr_ch_))),
            (duration_+x_duration_, Play(GaussianSquare(duration=duration_,
                                        amp=-amp_ctr_, sigma=sigma_, width=width_),
                                    ControlChannel(ctr_ch_))),
            (2*duration_+x_duration_, Play(Drag(duration=x_duration_, amp=x_amp_,
                                               sigma=x_sigma_, beta=x_beta_),
                                    DriveChannel(x_dr_ch_), name='Xp_d5')),
        name="rzx(theta)")

    return rzx_cal
```

Return the rzx schedule.

Our results for the RZX circuit on real hardware (RZX and CNOT benchmarks were run at the same time so we can compare the results):

Trotter steps	RZX Fidelity	CNOT Fidelity
8	0.2834 +41%)	0.201
7	0.3041 (+30%)	0.2336
6	0.2677 (+26%)	0.2132
5	0.1642 (+32%)	0.124



We can clearly see the fidelity improvements due to the RZX gate.

To check the performance of amplitude calibration of the RZX pulse, we benchmarked the fidelity of RZX sequences:

```
from qiskit.ignis.verification.tomography import state_tomography_circuits,
                                                 StateTomographyFitter
def rzx_experiment(qubits, repetitions, control, target, target_time,
                   trotter_steps, amp_mul=1.0):
    q0 = qubits[0] # control
    q1 = qubits[1] # target

    qc = QuantumCircuit(7)

    if control == 1:
        qc.x(q0)
    if target == 1:
        qc.x(q1)

    rzx_gate = Gate('rzx_', 2, [])
    for _ in range(repetitions):
        qc.append(rzx_gate, [q0, q1])

    rzx_q0q1_sched = rzx_cal((q0,q1), 2*target_time/trotter_steps, amp_mul=amp_mul,
                             x_cal=True)
    qc.add_calibration('rzx_', (q0,q1), rzx_q0q1_sched)
    inst_map_cal.add('rzx_', (q0,q1), rzx_q0q1_sched)

    qc = state_tomography_circuits(qc, [q0, q1])
    qc_transpile = transpile(qc, backend, optimization_level=0,
                             basis_gates=inst_map_cal.instructions)
    return qc_transpile
```

This function returns a circuit with “repetitions” number of RZX gates in sequence.

Initiate the state of the control and target qubits.

Create a custom RZX gate.
Create the RZX sequence.

Apply the custom amplitude to the RZX sequence.

Run state tomography.

Next we want to have the noiseless state known for the RZX sequence:

```
from qiskit.quantum_info import Statevector
```

```python
def rzx_target_state(repetitions, control, target, target_time, trotter_steps):

    qc = QuantumCircuit(2)
    if control == 1:
        qc.x(0)
    if target == 1:
        qc.x(1)

    for _ in range(repetitions):
        qc.rzx(2*target_time/trotter_steps, 0,1)

    target_state_rzx = Statevector.from_instruction(qc)
    return target_state_rzx
```

This function returns a state vector for the RZX sequence. This will be used to calculate the fidelity of the final circuit.

Initiate the state of the control and target qubits.

Create the RZX sequence.

Calculate and return the state vector.

For our experiments we will run multiple RZX sequences with different target and state values:

```python
def gen_experiments(qubits_, target_time, trotter_steps, amp_mul):
    experiments = []
    circuits = []
    for exp_idx in range(0, 8):
        repetitions = (exp_idx+1)*4
        target_state = rzx_target_state(repetitions=repetitions, control=0,
                                        target=0, target_time=np.pi,
                                        trotter_steps=trotter_steps)
        experiments.append({'qubits': qubits_, 'repetitions': repetitions,
                            'amp_mul': amp_mul, 'control': 0, 'target': 0,
                            'target_state': target_state})
        circuits.append(None)

        target_state = rzx_target_state(repetitions=repetitions, control=0,
                                        target=1, target_time=np.pi,
                                        trotter_steps=trotter_steps)
        experiments.append({'qubits': qubits_, 'repetitions': repetitions,
                            'amp_mul': amp_mul, 'control': 0, 'target': 1,
                            'target_state': target_state})
        circuits.append(None)

        target_state = rzx_target_state(repetitions=repetitions, control=1,
                                        target=0, target_time=np.pi,
                                        trotter_steps=trotter_steps)
        experiments.append({'qubits': qubits_, 'repetitions': repetitions,
                            'amp_mul': amp_mul, 'control': 1, 'target': 0,
                            'target_state': target_state})
        circuits.append(None)

        target_state = rzx_target_state(repetitions=repetitions, control=1,
                                        target=1, target_time=np.pi,
                                        trotter_steps=trotter_steps)
        experiments.append({'qubits': qubits_, 'repetitions': repetitions,
                            'amp_mul': amp_mul, 'control': 1, 'target': 1,
                            'target_state': target_state})
        circuits.append(None)
    return experiments, circuits

qubits_ = [1,3]
target_time = np.pi
trotter_steps = 11
amp_mul = 1.1

experiments, circuits = gen_experiments(qubits_, target_time, trotter_steps,
amp_mul)

def execute_experiments():
    qcs_ = []
    for idx, exp in enumerate(experiments):
        qc = rzx_experiment(qubits=exp['qubits'], repetitions=exp['repetitions'],
                            control=exp['control'], target=exp['target'],
                            target_time=target_time, trotter_steps=trotter_steps,
                            amp_mul=exp['amp_mul'])
```

Function to create the experiments with different RZX sequences like: 4xRZX, 8xRZX, 12xRZX, 16xRZX, 20xRZX, 24xRZX, 28xRZX, 32xRZX, 36xRZX.
We run those circuits will all target and control combinations.

Setup the benchmark default parameters.

Create the experiments as RZX sequences.

Run the experiments on quantum hardware.

```
        circuits[idx] = qc
        qcs_.extend(qc)

    job = execute(qcs_, backend, meas_level=2, shots=10000, optimization_level=0,
                  inst_map=inst_map_cal)

execute_experiments()
```

We plot the fidelity of the circuit vs number of RZX gates in sequences:

```
def state_tomo(result, st_qcs, target_state_rzx):
    tomo_fitter = StateTomographyFitter(result, st_qcs)
    rho_fit = tomo_fitter.fit(method='lstsq')
    fid = state_fidelity(rho_fit, target_state_rzx)
    return fid

num_tomo_circ_2q = 3**2

fids = []
for idx, exp in enumerate(experiments):
    target_state_rzx = exp['target_state']

    results = copy.deepcopy(jobs.result())
    results_dict = results.to_dict()
    results_dict['results'] = \
       results.to_dict()['results'][idx*num_tomo_circ_2q:(idx+1)*num_tomo_circ_2q]
    results = results.from_dict(results_dict)

    fid = state_tomo(results, circuits[idx], exp['target_state'])
    fids.append(fid)
    color = 'blue' if exp['target'] == 0 and exp['control'] == 0 else ''
    color = 'green' if exp['target'] == 1 and exp['control'] == 0 else color
    color = 'red' if exp['target'] == 0 and exp['control'] == 1 else color
    color = 'cyan' if exp['target'] == 1 and exp['control'] == 1 else color
    plt.scatter(exp['repetitions'], fid, color=color)

plt.legend(["c=|0> t=|0>", "c=|0> t=|1>", "c=|1> t=|0>", "c=|1> t=|1>"])
plt.show()

print(float(sum(fids))/len(fids))
```

Calculate state fidelity based on the results from quantum circuits and the noiseless state vector of the circuit.

Iterate through all experiments (that is RZX sequences and target/control values).

Extract the specific experiments from the results list.

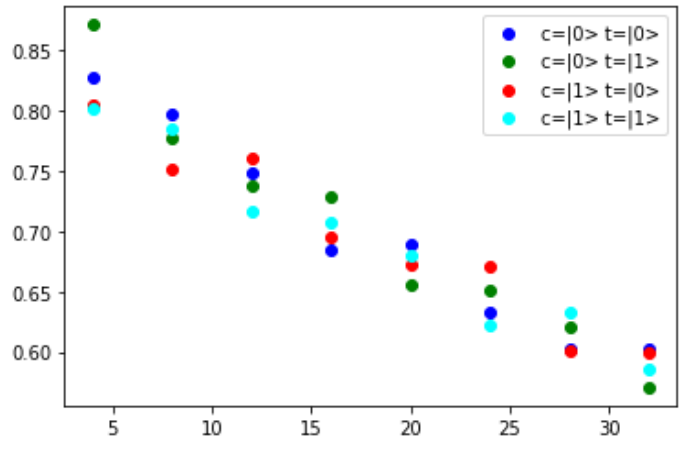
Calculate fidelity.

Display the target/control values in different colors.

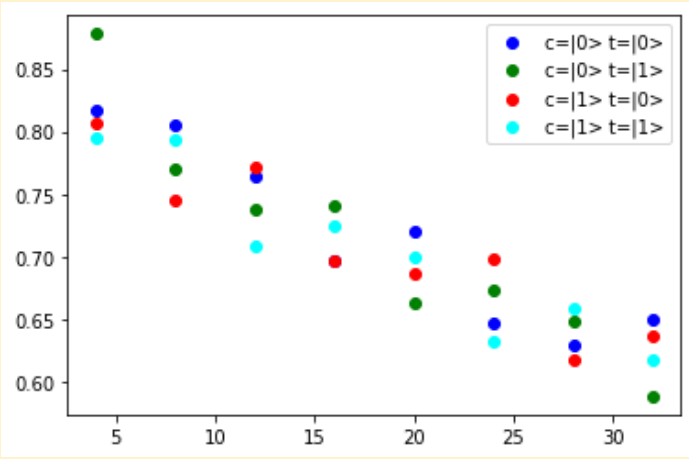
Display final fidelity as the average value from all experiments.

The results from the benchmark are as follows:

Amplitude multiplier = 0.98
Average fidelity = 0.6961

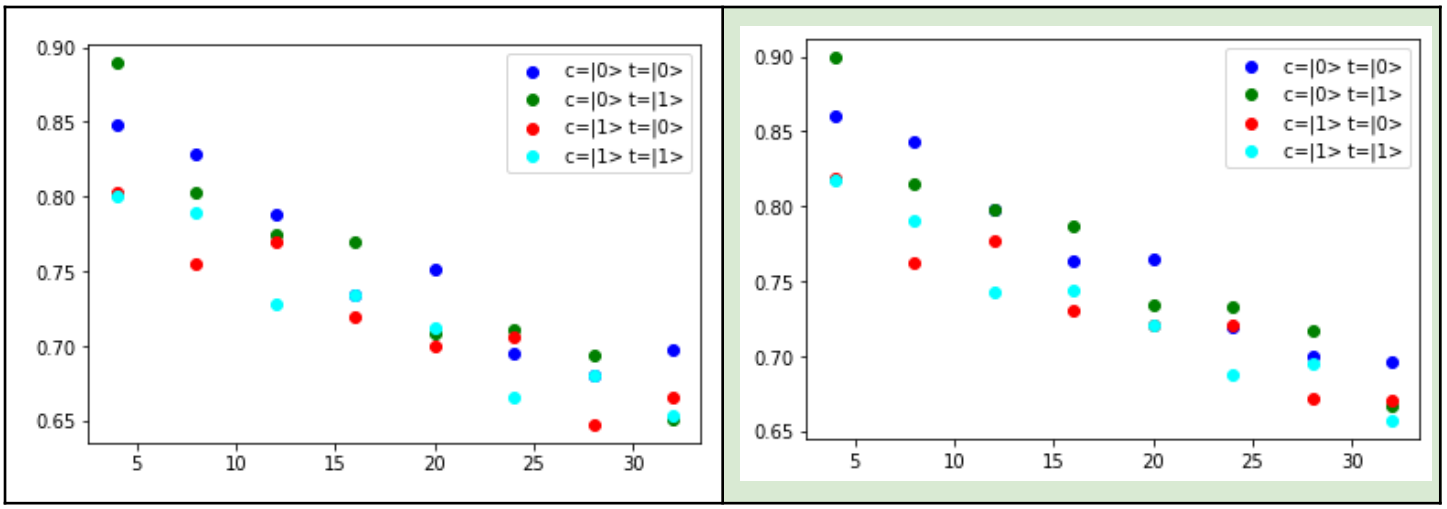


Amplitude multiplier = 1.0
Average fidelity = 0.71



Amplitude multiplier = 1.02
Average fidelity = 0.7361

Amplitude multiplier = 1.04
Average fidelity = 0.7506

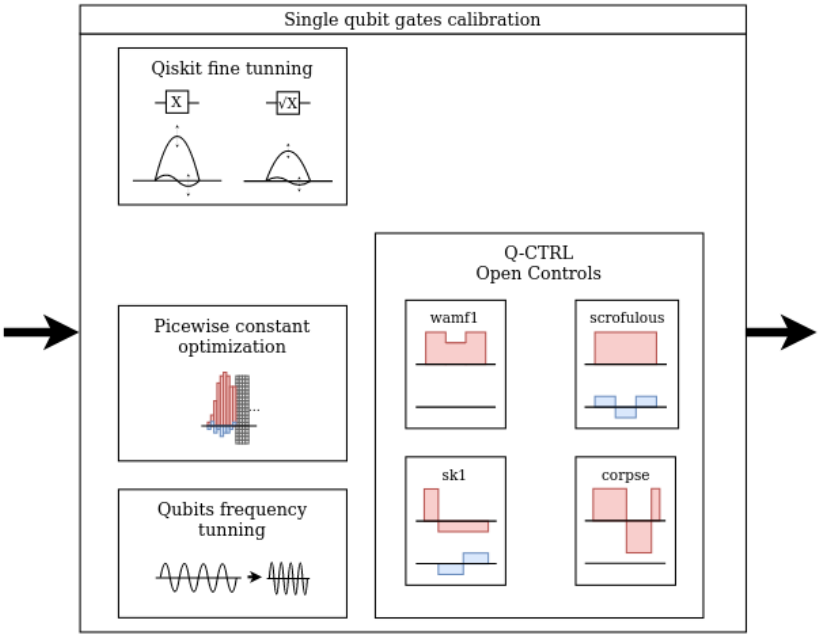


We can see that the default amplitude settings for the RZX pulse can be optimized. But this requires few benchmarks to be run, due to limited availability we were not able to include RZX calibration in the final result.

Application to other problems

The direction of the CNOT gate is important because on many systems only one direction is the native one. The non native direction is slower and requires additional pulses. So creating the quantum circuits this could be taken into account to improve the accuracy.

Single qubit gates calibration

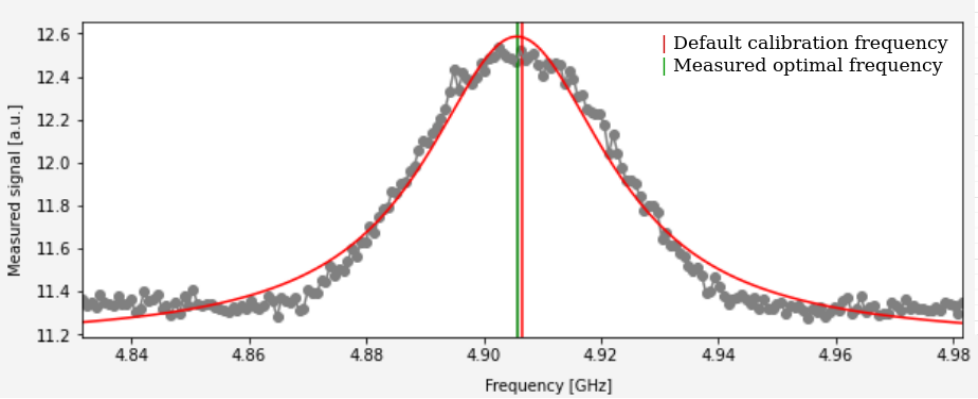


Generic introduction

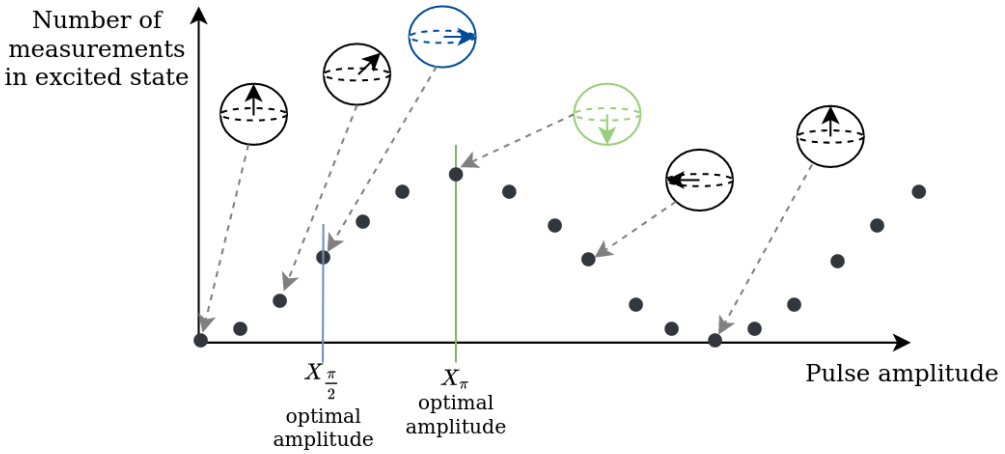
Qiskit calibration routines

The quantum circuit is susceptible to shifts of its internal parameters over the time (Proctor et al.), this results in decalibration of the quantum computers overtime. IBM quantum computers are calibrated every 24 hours (“System properties”).
The default calibration is very close to perfect, but it can be improved with longer calibration routines.

The simplest example of a calibration parameter is the Qubit frequency (“Calibrating Qubits with Qiskit Pulse”), the calibration procedure in the simplest form is a frequency sweep to determine the frequency at which the qubit shifts to the excited state.
Example of the measurement output:

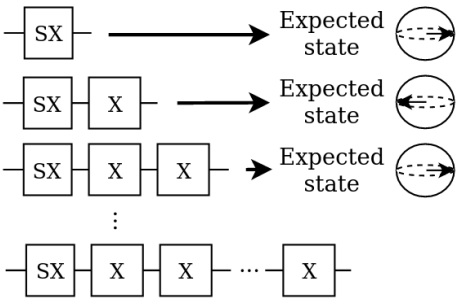


The Rabi (Menke) experiment can determine the amplitudes of the X_{π} and $X_{\pi/2}$ pulses. Those pulses represent the basic gates X and √X (SX) gates, which are the basic building blocks of quantum circuits because with an additional RZ gate they can represent all single qubit rotation operators.
For this experiment we change the amplitude of the pulse applied on the tested qubit. Higher amplitude will cause the rotation in the X plane of the Bloch sphere. We want to find the amplitude of the pulse which results in a pure excited state.

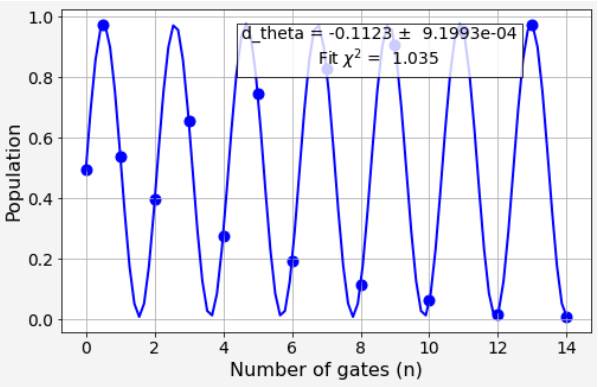


Rabi experiment will result in multiple amplitudes which generate the X_{π} and $X_{\pi/2}$ operations, pulses with lower amplitude are preferred because higher drive amplitude results in unwanted frequency harmonics which can cause the qubit to shift to higher excited levels.
The pulse duration is linked to the amplitude - shorter pulses require higher amplitudes, this creates a balancing act between decoherence noise (longer duration) and the pulse harmonics noise (higher amplitude).

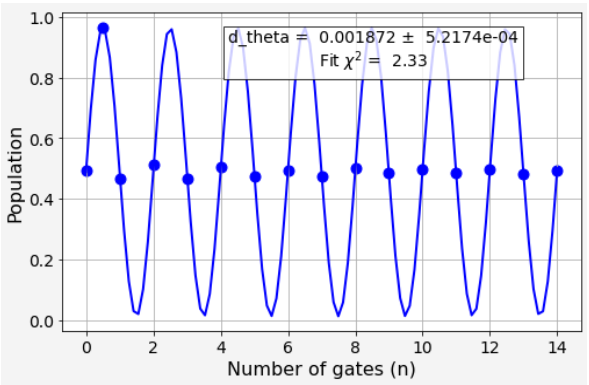
This calibration routine is very often done in 2 steps: Rough amplitude estimation and fine tuning.
Rough amplitude estimation is done with the Rabi experiment, where we focus on creating the single X and SX gates. We can narrow our amplitude sweep to fine tune the optimal amplitude, but at some moment we will not be able to find the maximum point due to measurement noise.
To improve the amplitude estimation we need to apply multiple X or SX gates. Every gate will apply the calibration error so we can clearly see the deviation from the expected states.
For X calibration we need to create the circuits which will result in the superposition state, that is circuits with a single SX gate, SX and X gates, SX and two X gates, etc.



Uncalibrated results shows a divergence in the measured state:



Where the calibrated gate shows that the number of excited state vs ground states is less susceptible to the number of gates in the circuit.



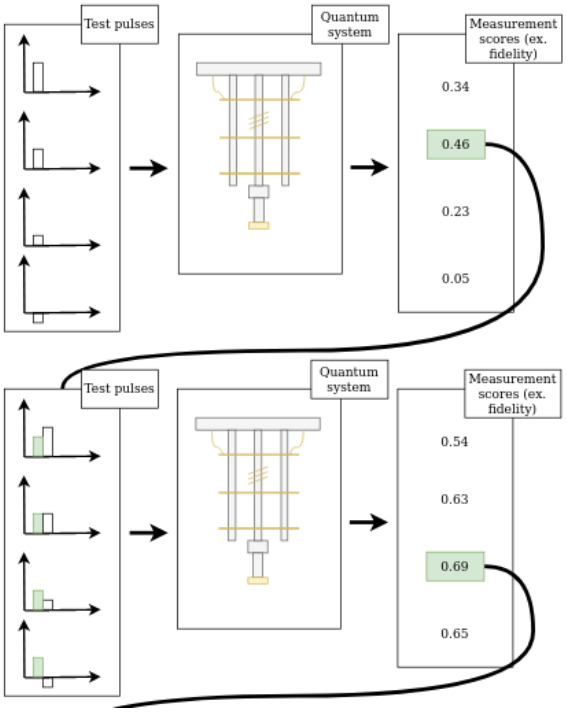
Similar fine tuning experiment can be performed for calibrating the SX gate.

To reduce the harmonics due to high pulse amplitude a so called Derivative Removal by Adiabatic Gate (Drag) pulse was design (Gambetta et al.) (Theis et al.):

This pulse is constructed based on Gausian pulse with an additional Gaussian derivative component and lifting applied (“Drag — Qiskit 0.34.2 documentation”).
Qiskit provides a function which performs the Drag calibration (“Calibrating single-qubit gates on ibmq_armonk — Qiskit Experiments 0.2.0 documentation”) for the rough estimation and fine tuning of the parameters. This calibration routine tries to find the optimal parameters of the Gaussian derivative component in the pulse.

PWC

We can go beyond the default pulses (Gaussian, Drag, etc.) to our own custom pulse waveforms. This is the idea behind Piecewise constant optimization. We create our pulses by directly defining the amplitude at every time interval.
The pulse design is automated in a closed loop environment:

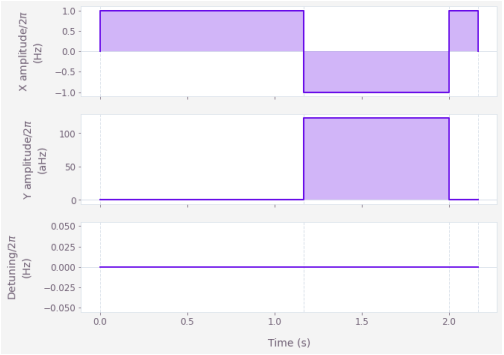


In the simplest case we start with test pulses constructed from a single pulse with different amplitudes, we create our circuit from the predefined pulses and execute it on a quantum computer. We then measure the outcome of the circuit and select the pulse with the best performance. Scoring can be based on outcome fidelity, gate tomography or number of outcomes in a specific state.
When we have our best pulse we repeat the sequence but we add a second random pulse to the sequence. We get the new measurements from which we select the sequence with the best score and continue by adding the next pulse.
The new pulses are added until we reach the expected duration for our gate. To increase further the gate performance we start again from the first pulse and change it slightly and benchmark the new sequence, then proceed to the second pulse and adjust it slightly to find the improvement in the gate. This can be repeated multiple times to get the needed fidelity.
The main drawback is the number of circuits which we need to run on the quantum computer. We are basically guessing at random so many of the circuits will result in lower performance. To reduce the number of runs we can applied gradient base techniques like Gradient Ascent Pulse Engineering (GRAPE (Goerz), BFGS-GRAPE [Link] (de Fouquieres et al.), d-GRAPE (Wu et al.), Chopped RAndom Basis (CRAB) (Doria et al.)) or apply deep reinforcement learning (Baum et al.).

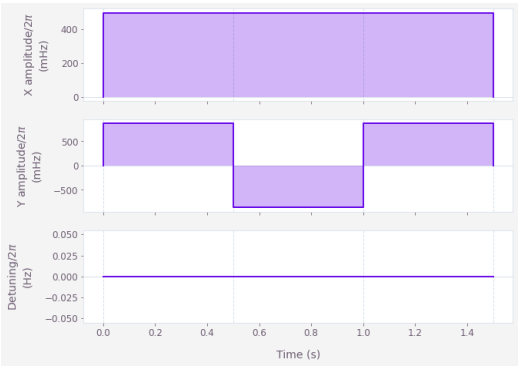
Predefine pulse schedules

Multiple predefined pulse schedules (Cummins et al.) were designed to reduce systematic error occuring in quantum systems. Errors can be viewed as imperfections in the Bloch sphere rotation and the composite pulses can be designed to reduce sensitivity of the system to those errors.
The most common error which can be improved using composite pulses are:

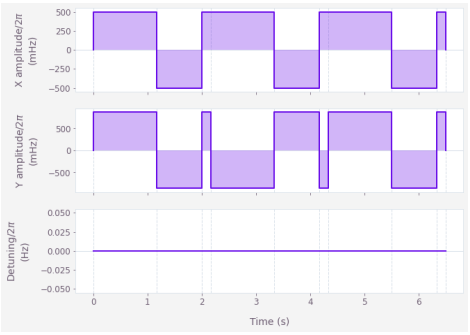
1. Off-resonance errors (Tycko) - errors of the qubit drive frequency estimation. Pulse example: CORPSE (Compensation for Off-Resonance with a Pulse SEquence), Walsh phase- (WPMF) modulated filters (Ball and Biercuk).
   CORPSE schedule:



2. Pulse amplitude error - errors of the amplitude of the pulse. Pulse example: SCROFULOUS (Short Composite ROtation For Undoing Length Over and Under Shoot) (Tycko), Walsh amplitude- (WAMF) modulated filters.
SCROFULOUS schedule:



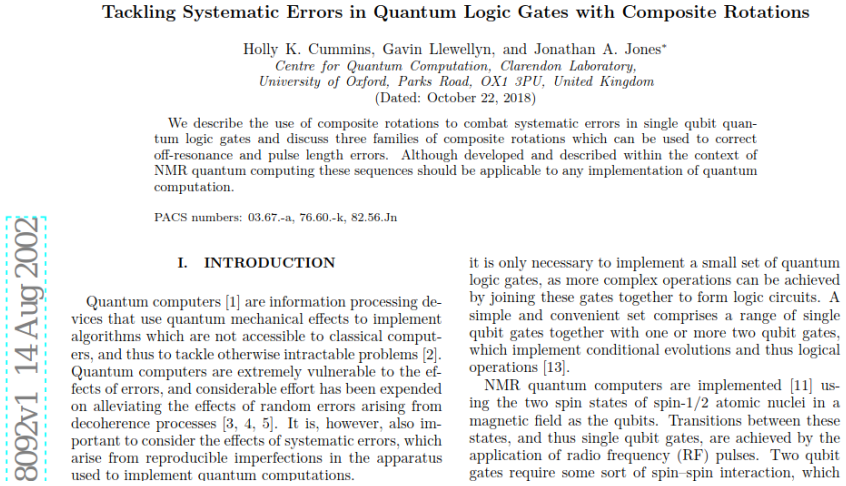
The off-resonance and pulse amplitude error pulses can be combined to simultaneously reduce both sources of errors (Ichikawa et al.)
CORPSE combine with SCROFULOUS:



Papers for further reading.

1. **Tackling Systematic Errors in Quantum Logic Gates with Composite Rotations**
Holly K. Cummins, Gavin Llewellyn, and Jonathan A. Jones
https://arxiv.org/pdf/quant-ph/0208092.pdf

Most of the custom pulse schedules described above were introduced in this paper. The authors design efficient pulse schedules to tackle off-resonance and pulse length errors and BB1 pulse. The mathematical justification for the pulse design is provided in this paper, but without any experimental results.



Tackling Systematic Errors in Quantum Logic Gates with Composite Rotations

Holly K. Cummins, Gavin Llewellyn, and Jonathan A. Jones*
Centre for Quantum Computation, Clarendon Laboratory, University of Oxford, Parks Road, OX1 3PU, United Kingdom
(Dated: October 22, 2018)

We describe the use of composite rotations to combat systematic errors in single qubit quantum logic gates and discuss three families of composite rotations which can be used to correct off-resonance and pulse length errors. Although developed and described within the context of NMR quantum computing these sequences should be applicable to any implementation of quantum computation.

PACS numbers: 03.67.-a, 76.60.-k, 82.56.Jn

I. INTRODUCTION

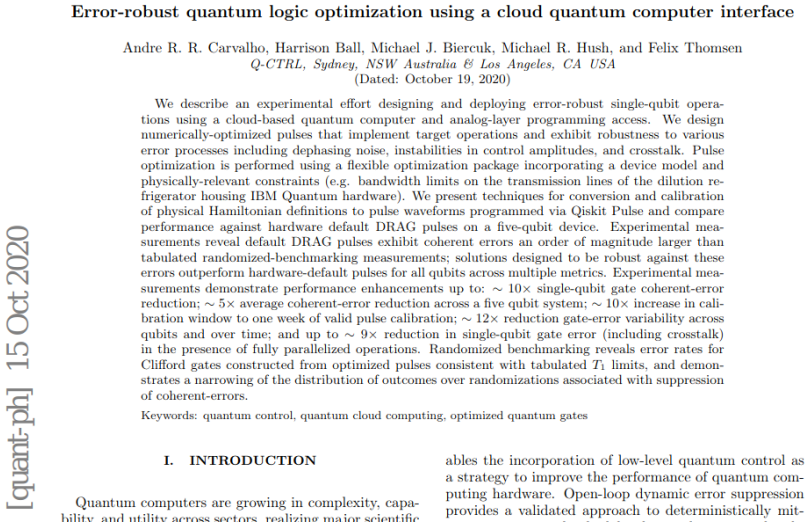
Quantum computers [1] are information processing devices that use quantum mechanical effects to implement algorithms which are not accessible to classical computers, and thus to tackle otherwise intractable problems [2]. Quantum computers are extremely vulnerable to the effects of errors, and considerable effort has been expended on alleviating the effects of random errors arising from decoherence processes [3, 4, 5]. It is, however, also important to consider the effects of systematic errors, which arise from reproducible imperfections in the apparatus used to implement quantum computations.

it is only necessary to implement a small set of quantum logic gates, as more complex operations can be achieved by joining these gates together to form logic circuits. A simple and convenient set comprises a range of single qubit gates together with one or more two qubit gates, which implement conditional evolutions and thus logical operations [13].

NMR quantum computers are implemented [11] using the two spin states of spin-1/2 atomic nuclei in a magnetic field as the qubits. Transitions between these states, and thus single qubit gates, are achieved by the application of radio frequency (RF) pulses. Two qubit gates require some sort of spin–spin interaction, which

2. **Error-robust quantum logic optimization using a cloud quantum computer interface**
Andre R. R. Carvalho, Harrison Ball, Michael J. Biercuk, Michael R. Hush, and Felix Thomsen
https://arxiv.org/pdf/2010.08057.pdf

   This paper provides an easy introduction about error robust pulse gates (but without going into high details about the different costume pulse schedules). It contains well defined experiments showing the benefits of pulse optimized gates.



Error-robust quantum logic optimization using a cloud quantum computer interface

Andre R. R. Carvalho, Harrison Ball, Michael J. Biercuk, Michael R. Hush, and Felix Thomsen
Q-CTRL, Sydney, NSW Australia & Los Angeles, CA USA
(Dated: October 19, 2020)

We describe an experimental effort designing and deploying error-robust single-qubit operations using a cloud-based quantum computer and analog-layer programming access. We design numerically-optimized pulses that implement target operations and exhibit robustness to various error processes including dephasing noise, instabilities in control amplitudes, and crosstalk. Pulse optimization is performed using a flexible optimization package incorporating a device model and physically-relevant constraints (e.g. bandwidth limits on the transmission lines of the dilution refrigerator housing IBM Quantum hardware). We present techniques for conversion and calibration of physical Hamiltonian definitions to pulse waveforms programmed via Qiskit Pulse and compare performance against hardware default DRAG pulses on a five-qubit device. Experimental measurements reveal default DRAG pulses exhibit coherent errors an order of magnitude larger than tabulated randomized-benchmarking measurements; solutions designed to be robust against these errors outperform hardware-default pulses for all qubits across multiple metrics. Experimental measurements demonstrate performance enhancements up to: $\sim 10\times$ single-qubit gate coherent-error reduction; $\sim 5\times$ average coherent-error reduction across a five qubit system; $\sim 10\times$ increase in calibration window to one week of valid pulse calibration; $\sim 12\times$ reduction gate-error variability across qubits and over time; and up to $\sim 9\times$ reduction in single-qubit gate error (including crosstalk) in the presence of fully parallelized operations. Randomized benchmarking reveals error rates for Clifford gates constructed from optimized pulses consistent with tabulated T_1 limits, and demonstrates a narrowing of the distribution of outcomes over randomizations associated with suppression of coherent-errors.

Keywords: quantum control, quantum cloud computing, optimized quantum gates

I. INTRODUCTION

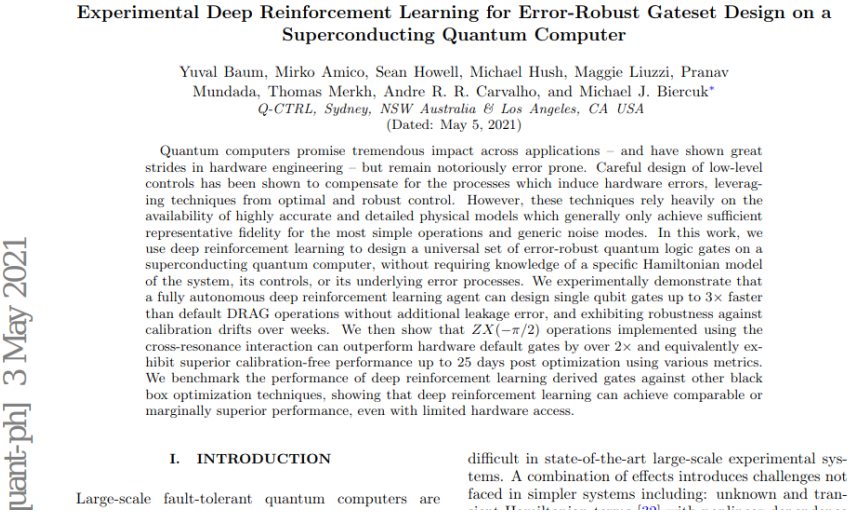
Quantum computers are growing in complexity, capability, and utility across sectors, realizing major scientific

ables the incorporation of low-level quantum control as a strategy to improve the performance of quantum computing hardware. Open-loop dynamic error suppression provides a validated approach to deterministically mit-

[quant-ph] 15 Oct 2020

3. **Experimental Deep Reinforcement Learning for Error-Robust Gateset Design on a Superconducting Quantum Computer**
Yuval Baum, Mirko Amico, Sean Howell, Michael Hush, Maggie Liuzzi, Pranav, Mundada, Thomas Merkh, Andre R. R. Carvalho, and Michael J. Biercuk
https://arxiv.org/pdf/2105.01079.pdf

   Good low level introduction to piecewise constant optimization for quantum gates. Here we have experimental results from PWC created gates on ibm quantum hardware. Authors additionally checked the time stability of the created gates.



Experimental Deep Reinforcement Learning for Error-Robust Gateset Design on a Superconducting Quantum Computer

Yuval Baum, Mirko Amico, Sean Howell, Michael Hush, Maggie Liuzzi, Pranav Mundada, Thomas Merkh, Andre R. R. Carvalho, and Michael J. Biercuk*
Q-CTRL, Sydney, NSW Australia & Los Angeles, CA USA
(Dated: May 5, 2021)

Quantum computers promise tremendous impact across applications – and have shown great strides in hardware engineering – but remain notoriously error prone. Careful design of low-level controls has been shown to compensate for the processes which induce hardware errors, leveraging techniques from optimal and robust control. However, these techniques rely heavily on the availability of highly accurate and detailed physical models which generally only achieve sufficient representative fidelity for the most simple operations and generic noise modes. In this work, we use deep reinforcement learning to design a universal set of error-robust quantum logic gates on a superconducting quantum computer, without requiring knowledge of a specific Hamiltonian model of the system, its controls, or its underlying error processes. We experimentally demonstrate that a fully autonomous deep reinforcement learning agent can design single qubit gates up to $3\times$ faster than default DRAG operations without additional leakage error, and exhibiting robustness against calibration drifts over weeks. We then show that $ZX(-\pi/2)$ operations implemented using the cross-resonance interaction can outperform hardware default gates by over $2\times$ and equivalently exhibit superior calibration-free performance up to 25 days post optimization using various metrics. We benchmark the performance of deep reinforcement learning derived gates against other black box optimization techniques, showing that deep reinforcement learning can achieve comparable or marginally superior performance, even with limited hardware access.

I. INTRODUCTION

Large-scale fault-tolerant quantum computers are

difficult in state-of-the-art large-scale experimental systems. A combination of effects introduces challenges not faced in simpler systems including: unknown and tran-

[quant-ph] 3 May 2021

Application to Ising simulation

Predefined pulse schedules are often longer than the Drag pulses for the X and SX gates. This can result in a longer overall circuit duration and the benefits need to be closely benchmarked. But this approach is valid for all quantum circuits to which we want to add custom pulses.
Ising models are often constructed using dozens of X and SX pulses. Calibration routine for custom pulse schedules will need more steps - firstly rough calibration to estimate the amplitude and duration of the pulse and secondly the fine tune calibration based on multiple X or SX gates in series.

Experiments

Qiskit fine tunning

Qiskit provides functions to perform amplitude and drag calibration automatically. We decided to calibrate only the used qubits (qubit 1, 3 and 5) using the following sequence:

1. Amplitude fine tuning for X gate with up to 14 gates in sequence (default qiskit settings).

2. Amplitude fine tuning for SX gate with up to 25 gates in sequence (default qiskit settings).
3. Drag calibration for X gate with up to 20 gates in sequence (default qiskit settings).
4. Drag calibration for SX gate with up to 20 gates in sequence (default qiskit settings).

5. Amplitude fine tuning for X gate with up to 100 gates in sequence.
6. Amplitude fine tuning for SX gate with up to 100 gates in sequence.
7. Drag calibration for X gate with up to 50 gates in sequence.
8. Drag calibration for SX gate with up to 50 gates in sequence.

```python
# Amplitude calibration functions
from qiskit_experiments.library.calibration.fine_amplitude import FineXAmplitudeCal,
FineSXAmplitudeCal

library = FixedFrequencyTransmon(default_values={"duration": duration})
dummy_cals = Calibrations.from_backend(backend, library)

update_cal_from_inst_map(qubits, ['sx','x'], dummy_cals, inst_map_cal)

jobs_data_amp_x = []
jobs_data_amp_sx = []


def FineAmplitudeCal_(qubit, x_repetitions=None, sx_repetitions=None):
    FineXAmplitudeCal_(qubit, repetitions=x_repetitions)
    FineSXAmplitudeCal_(qubit, repetitions=sx_repetitions)

def FineXAmplitudeCal_(qubit, repetitions=None):

    amp_x_cal = FineXAmplitudeCal(qubit, dummy_cals, backend=backend, schedule_name='x')


    print("FineXAmplitudeCal", qubit, repetitions)
    if repetitions is not None:
        amp_x_cal.set_experiment_options(repetitions=repetitions)

    job_data_x = amp_x_cal.run().block_for_results()


    jobs_data_amp_x.append(job_data_x)


def FineSXAmplitudeCal_(qubit, repetitions=None):
    amp_sx_cal = FineSXAmplitudeCal(qubit, dummy_cals, backend=backend,
schedule_name='sx')
    print("FineSXAmplitudeCal", qubit, repetitions)
    if repetitions is not None:
        amp_sx_cal.set_experiment_options(repetitions=repetitions)
    job_data_sx = amp_sx_cal.run().block_for_results()
    jobs_data_amp_sx.append(job_data_sx)

# Drag calibration functions
from qiskit_experiments.library import FineXDragCal, FineSXDragCal

jobs_data_drag_x = []
jobs_data_drag_sx = []

def FineDragCal_(qubit, repetitions=None):
    FineXDragCal_(qubit, repetitions)
    FineSXDragCal_(qubit, repetitions)

def FineXDragCal_(qubit, repetitions=None):
    drag_data = FineXDragCal(qubit, dummy_cals, backend=backend)
    print("FineXDragCal", qubit, repetitions)
    if repetitions is not None:
        drag_data.set_experiment_options(repetitions=repetitions)
    job_data = drag_data.run().block_for_results()
    jobs_data_drag_x.append(job_data)

def FineSXDragCal_(qubit, repetitions=None):
    drag_data = FineSXDragCal(qubit, dummy_cals, backend=backend)
    print("FineSXDragCal", qubit, repetitions)
    if repetitions is not None:
        drag_data.set_experiment_options(repetitions=repetitions)
    job_data = drag_data.run().block_for_results()
    jobs_data_drag_sx.append(job_data)

# execute calibration routines

for qubit in [1,3,5]:
    FineAmplitudeCal_(qubit)
```

Import the Amplitude calibration functions from qiskit libraries.

Create a calibration object (dummy_cals) with default backend settings.

Copy the previous calibration (inst_map_cal) to experiment calibration (dummy_cals).

Structures for storing the job results in case further analysis is needed.

Function to combine X and SX amplitude calibration.

Function to perform X amplitude calibration.

Create X calibration object for specified qubit.

If we want to overwrite the default calibration repetitions, we need to set the experiment options.

Execute the calibration routine and wait until the job is finished.

Store the experiment in case of further analysis

Function to perform SX amplitude calibration based on FineXAmplitudeCal_

Similar to Amplitude calibration we use the build in calibration routines from qiskit for Drag calibration - FineXDragCal & FineSXDragCal

Call our custom calibration functions.

For every qubit used in the Ising model Perform Amplitude calibration

```
    FineDragCal_(qubit)


for qubit in [1,3,5]:
    FineAmplitudeCal_(qubit, x_repetitions=list(range(0,100)),
                      sx_repetitions=[0, 1, 2]+list(range(3,100,2)))
    FineDragCal_(qubit, repetitions=list(range(0,50)))
```

and Drag calibration with default parameters.

For every qubit used in the Ising model Perform extended Amplitude and Drag calibration with higher repetitions for the calibration routines.

After the calibration process is finished we need to copy the calibration results into our global instruction_map (structure containing global gates schedules)

```
def update_drag_parameter(qubit, instruction, parameter, new_value, inst_map):


    old_sched = inst_map.pop(instruction, qubit)



    for instr_ in old_sched.instructions:
        if isinstance(instr_[1], pulse.instructions.play.Play):
            if isinstance(instr_[1].operands[0], pulse.Drag):



                duration = new_value if parameter == 'duration' else \
                           instr_[1].operands[0].duration
                amp = new_value if parameter == 'amp' else instr_[1].operands[0].amp
                sigma = new_value if parameter == 'sigma' else instr_[1].operands[0].sigma
                beta = new_value if parameter == 'beta' else instr_[1].operands[0].beta
                name = new_value if parameter == 'name' else instr_[1].operands[0].name

    new_sched = Schedule()
    new_sched += Play(Drag(duration, amp, sigma, beta, name), d(qubit))



    inst_map.add(instruction, qubit, new_sched)



def update_cal_from_inst_map(qubits, instructions, cal, inst_map):


    for qubit in qubits:
        for instruction in instructions:

            sched = inst_map.get(instruction, qubit)


            for instr_ in sched.instructions:
                if isinstance(instr_[1], pulse.instructions.play.Play):
                    if isinstance(instr_[1].operands[0], pulse.Drag):

                        duration = instr_[1].operands[0].duration
                        amp = instr_[1].operands[0].amp
                        sigma = instr_[1].operands[0].sigma
                        beta = instr_[1].operands[0].beta
                        name = instr_[1].operands[0].name

                        cal.add_parameter_value(amp, 'amp', qubit, instruction)
                        cal.add_parameter_value(duration, 'duration', qubit, instruction)
                        cal.add_parameter_value(sigma, 'σ', qubit, instruction)
                        cal.add_parameter_value(beta, 'β', qubit, instruction)

def update_inst_map_from_cal(qubits, instructions, cal, inst_map):

    for qubit in qubits:
        for instruction in instructions:

            update_drag_parameter(qubit, instruction, 'amp',
                    cal.get_parameter_value('amp', qubit, instruction), inst_map)
            update_drag_parameter(qubit, instruction, 'beta',
                     cal.get_parameter_value('β', qubit, instruction), inst_map)
            update_drag_parameter(qubit, instruction, 'sigma',
                     cal.get_parameter_value('σ', qubit, instruction), inst_map)
            update_drag_parameter(qubit, instruction, 'duration',
```

Utils functions to update instruction_schedule_maps and calibrations for Drag Pulses

Function which update the Drag pulse parameters in the instruction_map (inst_map)

Get the instruction to update from the inst_map. Using the .pop function we remove the specific instruction from our global inst_map.

Instructions are returned in a tuple (<time_value>, <Instruction>) and all Drag pulses are combined in Play instruction. So we look for Play instruction, with its pulse (operand) set to Drag.

We copy all Drag parameters to local objects with the exception of the parameter which we want to update - this parameter is taken from the function argument.

We create a new empty schedule.
To which we add a Play instruction with Drag pulse with our local parameters

We add our new instruction to the global inst_map.

Function to update calibration object based on instruction map

The function can be called with an array of qubits and instructions to update.

We make a schedule copy of the global instruction map.

We iterate through the instruction to find the Drag Pulse.

We copy the Drag Pulse parameters from the instruction map to local objects.

We update the calibration with the local Drag parameters.

Function to update instruction maps based on calibration parameters.

The function can be called with an array of qubits and instructions to update.

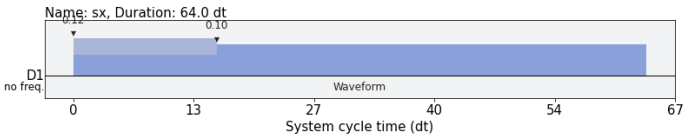
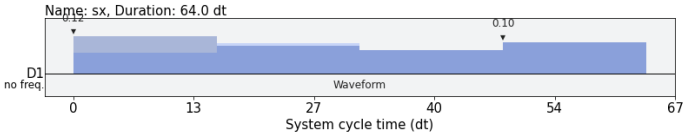
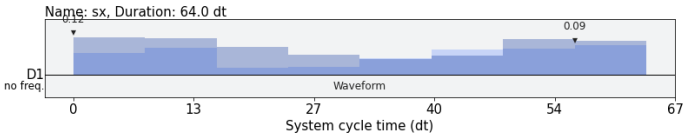
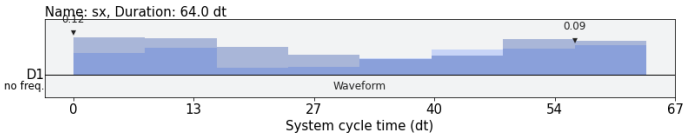
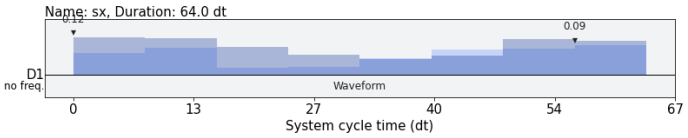
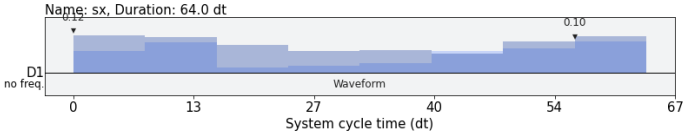
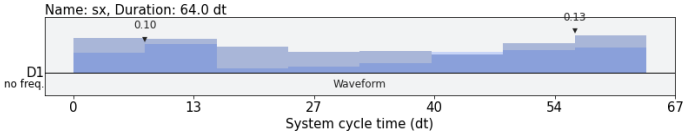
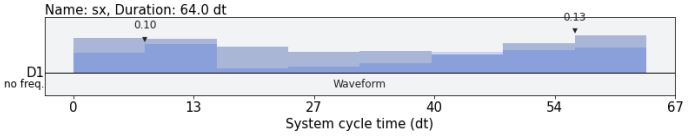
We use our previously defined update_drag_paramers util function. The actual calibration parameters can be easily obtained using the .get_parameter_value() function.

```
                    cal.get_parameter_value('duration', qubit, instruction), inst_map)

update_inst_map_from_cal(qubits, ['sx','x'], dummy_cals, inst_map_cal)
```

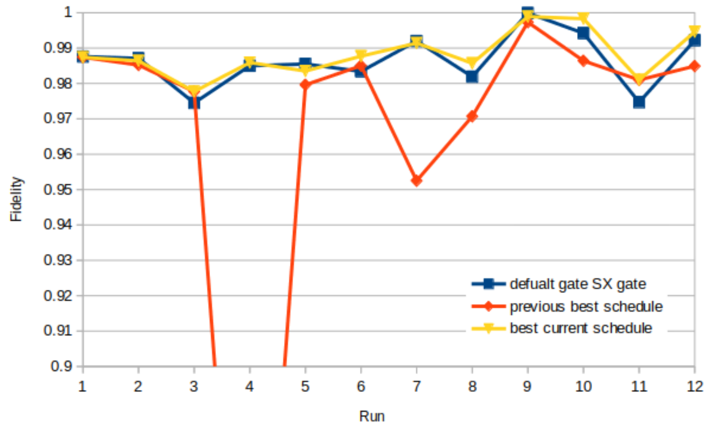
We copy the calibration results from dummy_cals (calibration on which we run our FineAmplitudeCalibration & FineDragCalibration functions) to our global inst_map_cal which stores all instruction schedules defining our quantum gates.

PWC

We create a simple test run using PWC optimization for defining SX gate. We decided to use only 64 segments and run up to 30 gates in sequence. State tomography of a circuit containing only SX gates in sequence was to score the performance of the pulse sequence.
This procedure required many jobs on the quantum computer.

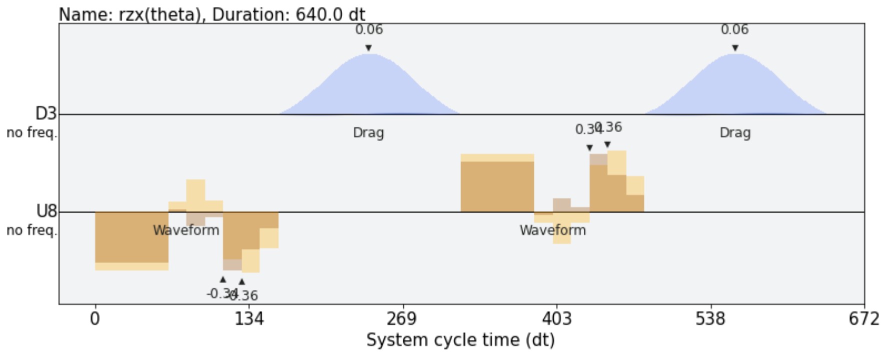
	Initial gate after first benchmark. In the initial phase we divide the pulse into 4 segments fidelity: 0.8482 default gate fidelity: 0.9865
	3 initial runs fidelity: 0.8831 default gate fidelity: 0.9823
We left the algorithm running for few hours and expanded the number of segments to 8	
	9 SX repetitions, fidelity reported with measurement filter gate fidelity: 0.9837 default gate fidelity: 0.9818 performance of previous best schedule on this repetition cycle: 0.9812
	11 SX repetitions, fidelity reported with measurement filter gate fidelity: 0.9942 default gate fidelity: 0.9937 performance of previous best schedule on this repetition cycle: 0.9792
	3 SX repetitions, fidelity reported with measurement filter gate fidelity: 0.9951 default gate fidelity: 0.9811 performance of previous best schedule on this repetition cycle: 0.9939
Run for more additional hours	
	17 SX repetitions, fidelity reported with measurement filter gate fidelity: 0.9983 default gate fidelity: 0.9942 performance of previous best schedule on this repetition cycle: 0.9864
	12 SX repetitions, fidelity reported with measurement filter gate fidelity: 0.981 default gate fidelity: 0.9747 performance of previous best schedule on this repetition cycle: 0.981
	10 SX repetitions, fidelity reported with measurement filter gate fidelity: 0.9946 default gate fidelity: 0.9922 performance of previous best schedule on this repetition cycle: 0.9849

The performance of the schedules at the end of the optimization.

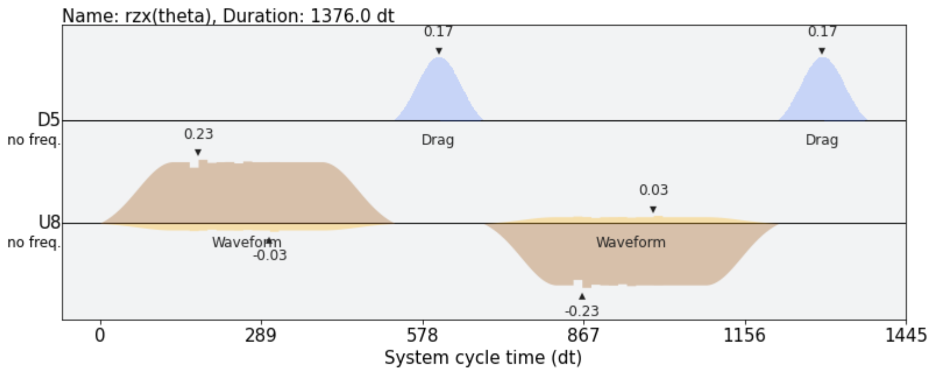


From our experiment we can clearly see that the performance of the currently benchmarked pulse very often is higher than the default qiskit gate, but this result is very fine tuned to the current number of gate repetition. The best schedule for specific number of gate repetition has poor performance when applied on a different gate set.

We ran a few iterations of the PWC algorithm to create an optimized RZX pattern, but the performance was pure.
The fidelity never exceeded 60% and the final waveform look as follows:



Additionally we run the PWC algorithm starting with the default RZX pulse and the resulting waveform was as follows:



Similar to full PWC optimization the performance of the new RZX pulse never exceeded the fidelity of the default pulse.

The code to apply PWC for RZX pulse is as follows:

```
from qiskit.quantum_info import state_fidelity, Statevector
from tqdm import tqdm_notebook as tqdm

qubits_ = [5,3]

def create_experiments(jobs, circuits, part_idx):
    waveforms = []
    experiments = \
        [{'qubits': qubits_, 'repetitions': 10, "comment": ""}] +\
        [{'qubits': qubits_, 'repetitions': 10, "comment": ""}] +\
        [{'qubits': qubits_, 'repetitions': 10, "comment": ""}] +\
        [{'qubits': qubits_, 'repetitions': 10, "comment": ""}] +\
        [{'qubits': qubits_, 'repetitions': 10, "comment": ""}] +\
        [{'qubits': qubits_, 'repetitions': 10, "comment": ""}] +\
        [{'qubits': qubits_, 'repetitions': 10, "comment": ""}] +\
        [{'qubits': qubits_, 'repetitions': 10, "comment": ""}] +\
        [{'qubits': qubits_, 'repetitions': 10, "comment": ""}] +\
        [{'qubits': qubits_, 'repetitions': 10, "comment": ""}]

    jobs = []
    circuits = []
```

This function randomly modifies a selected part of the RZX waveform for the benchmark.

```
    random.shuffle(experiments)
    for idx, exp in enumerate((experiments)):
        jobs.append(None)
        circuits.append(None)
        waveforms.append([])
        waveforms[-1] = list(np.copy(best_waveform))

        if idx != 0:
            part = waveforms[-1][part_idx*16:(part_idx+1)*16]
            real = np.random.normal(loc=1.0, scale=0.1)
            imag = np.random.normal(loc=1.0, scale=0.1)

            for idx_ in range(16):
                waveforms[-1][part_idx*16+idx_] = waveforms[-1][part_idx*16+idx_].real*real
+ 1j*waveforms[-1][part_idx*16+idx_].imag*imag

        exp['idx'] = idx
    return jobs, circuits, experiments, waveforms

def get_target_state():
    qc_ = QuantumCircuit(2)
    for _ in range(10):
        qc_.rzx(2*np.pi/6, 0,1)
    target_state_rzx = Statevector.from_instruction(qc_)
    return target_state_rzx

def evaluate_experiments(experiments, jobs, circuits, target_state, meas_fitter):
    fids = {}
    fids_mit = {}
    for idx, exp in enumerate(experiments):
        if str(exp) not in fids:
            fids[str(exp)] = []
            fids_mit[str(exp)] = []
        fid, fid_mit = state_tomo(jobs[idx].result(), circuits[idx], target_state,
meas_fitter)
        fids[str(exp)].append(fid)
        fids_mit[str(exp)].append(fid_mit)

    results = {}
    for exp in fids:
        results[str(round(sum(fids_mit[exp])/len(fids_mit[exp]),4))] = {'fids':
round(sum(fids[exp])/len(fids[exp]),4), 'exp': exp}

    best_result_idx = None
    for res in sorted(results, reverse=True):
        if best_result_idx is None:
            best_result_idx = json.loads(results[res]['exp'].replace("'",'"'))['idx']
    print("best_result_idx", best_result_idx)
    return results, best_result_idx

target_state = get_target_state()
meas_filter = get_meas_filter()

for _ in range(10):
    for idx__ in range(23,10,-1):
        print("create_experiments(...), part_idx:", idx__)
        jobs, circuits, experiments, waveforms = create_experiments(jobs, circuits,
part_idx=idx__)

        pbar = tqdm(total=len(experiments))
        for idx in range(len(experiments)):
            execute_experiment(experiments, idx)
            pbar.update(1)
        pbar.close()

        results, best_waveform_idx = evaluate_experiments(experiments, jobs, circuits,
target_state, meas_filter)
        best_waveform = waveforms[best_waveform_idx]
```

We don't modify the first experiment to have a baseline.

The waveform is divided into 16 pulses. And we iterate through the PWC experiment between the waveform pulses to change them randomly

Modify all 16 pulses by a random value for the real and imaginary parts of the waveform.

Target state function returns the expected RZX state to calculate the fidelity.

All the random circuits are evaluated and the index of the best performing is returned - it will be used as a basis for the next iteration.

We set up to run 10 iterations of the full RZX pulse.
The waveform is divided in 13 pulses - we don't change the ramp part of the RZX pulse, we only modify the flat top.

This result could be due to very poor optimization technique - using random pulse changes. Techniques based on gradients should perform better.

We did not use any PWC design pulse for the other experiments.

Q-CTRL Open Control

The Q-CTRL Open Control package provides predefined pulse schedules and can be installed via pip command:
`pip install qctrl-open-controls`
`pip install qctrl-visualizer`

```
import numpy as np
```

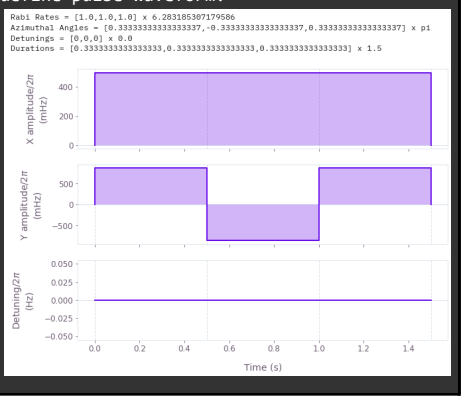
Import the needed libraries

```python
import matplotlib.pyplot as plt
from qctrlvisualizer import plot_controls
from qctrlopencontrols import (
    DrivenControl,
    new_bb1_control,
    new_primitive_control,
    new_sk1_control,
    new_corpse_control,
    new_scrofulous_control,
    new_gaussian_control,
    new_drag_control,
    new_corpse_in_sk1_control,
    new_corpse_in_bb1_control,
    new_wamf1_control,
    new_corpse_in_scrofulous_control
)

pulse_ = new_scrofulous_control(
    rabi_rotation=np.pi,
    azimuthal_angle=0,
    maximum_rabi_rate=2 * np.pi,
)

print(pulse_)
formatted_plot_data = pulse_.export(coordinates='cartesian',
                                    dimensionless_rabi_rate=False)
figure = plt.figure()
plot_controls(figure, formatted_plot_data)
```

q-ctrl package provides all pulses as objects.

Custom pulses require the rabi rotation values for the pulse - rabi rotation set to pi will result in the X gate, and rabi rotation set to pi/2 will define the SX gate.

Here we call a helper function to view the define pulse waveform:



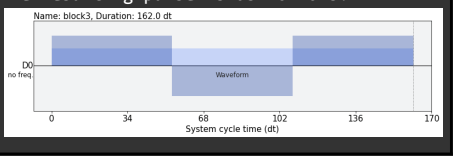
The library provides us with a time schedule which we need to convert to qiskit pulse waveform.

```python
def gen_waveform(x_duration=225, max_amplitude=0.2):
    time_max = pulse_.times[-1]
    x = []
    y = []
    for idx, time in enumerate(pulse_.times):
        time_ = pulse_.times[idx]
        if time_ == time_max:
            break
        time_end = pulse_.times[idx+1]
        time_tick = time_max / max_duration

        while time_ < time_end:
            time_ = time_ + time_tick
            x.append(time_)
            y.append(pulse_.amplitude_x[idx]*max_amplitude + \
                     1j*pulse_.amplitude_y[idx]*max_amplitude)
    return y

d0 = pulse.DriveChannel(0)
m0 = pulse.MeasureChannel(0)

y = gen_waveform(160, 0.01)

with pulse.build() as sched:
    pulse.play(y, d0)
    pulse.barrier(d0, m0)
sched.draw()
```

This function generates pulses for specified duration which defines our pulse schedule.

To use this new schedule we need to define a pulse schedule.

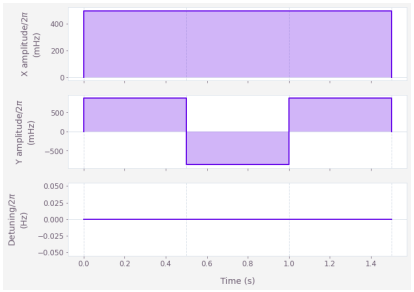
The resulting pulse is as follows:

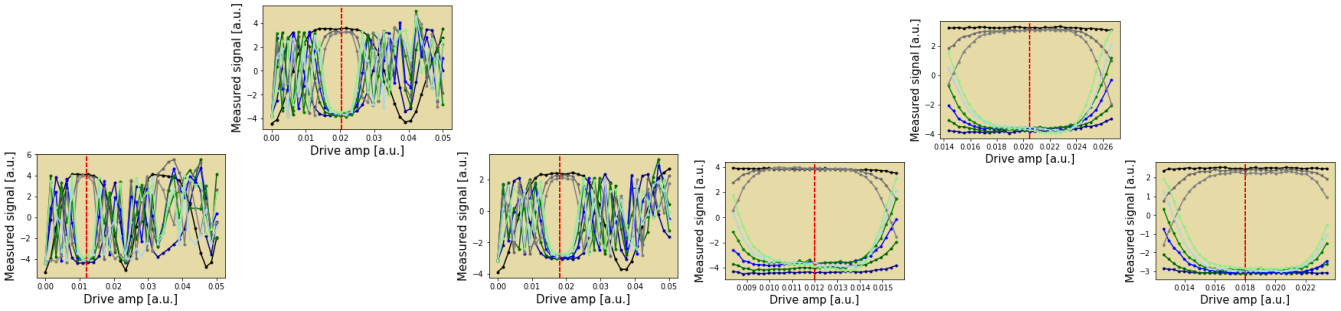


To calibrate the gates amplitude we run multiple rabi experiments with a rising number of gates in series (maximum 12). The benchmark was run on ibmq_casablanca computer.

Scrofulous X pulse with duration 336.

Scrofulous pulse is designed to mitigate amplitude errors, this results in non sinusoidal rabi plots.





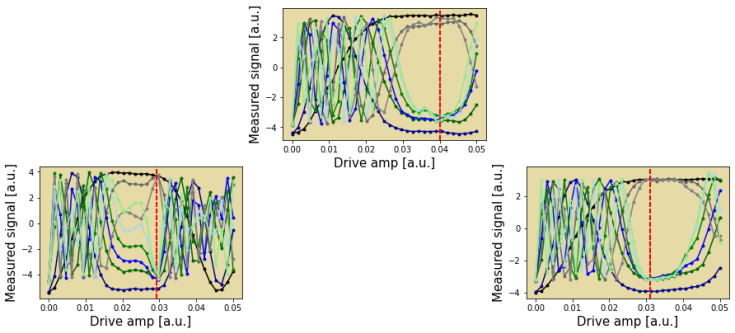
Charts on the left represent rough calibration and the one on the right are fine tuned amplitude views.
In the bellow charts only the first, third and fifth qubits are measured in the following pattern:

Qubit1

Qubit3 Qubit5

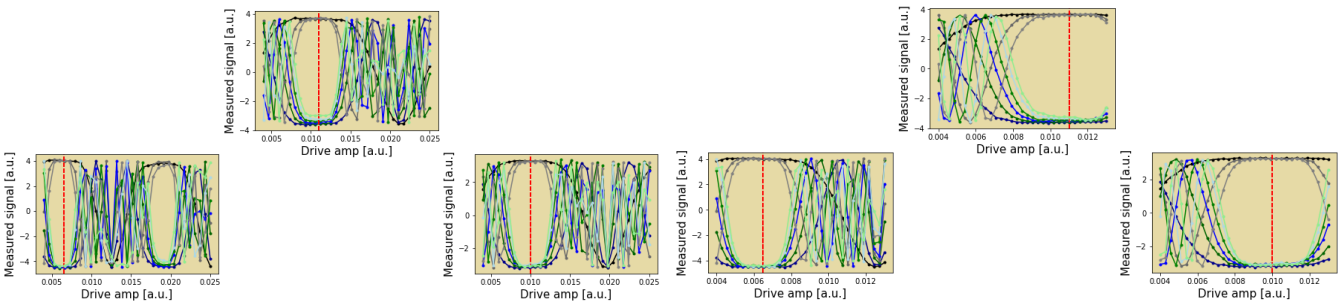
Scrofulous X pulse with duration 176.

Here we can see the problem with short pulse duration of the pulse schedules the performance is degraded.



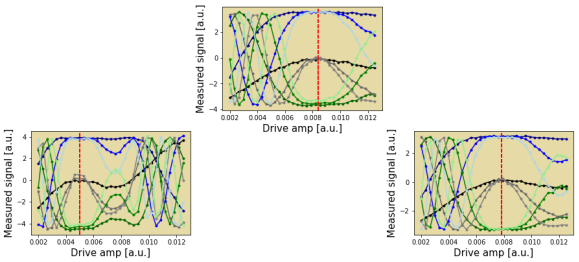
Scrofulous X pulse with duration 624

The performance is improved with the cost of a longer schedule.



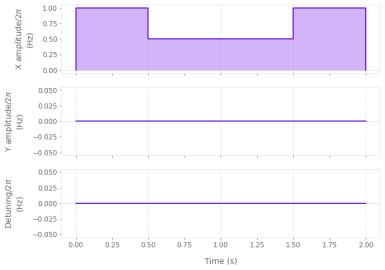
Scrofulous SX pulse with duration 608

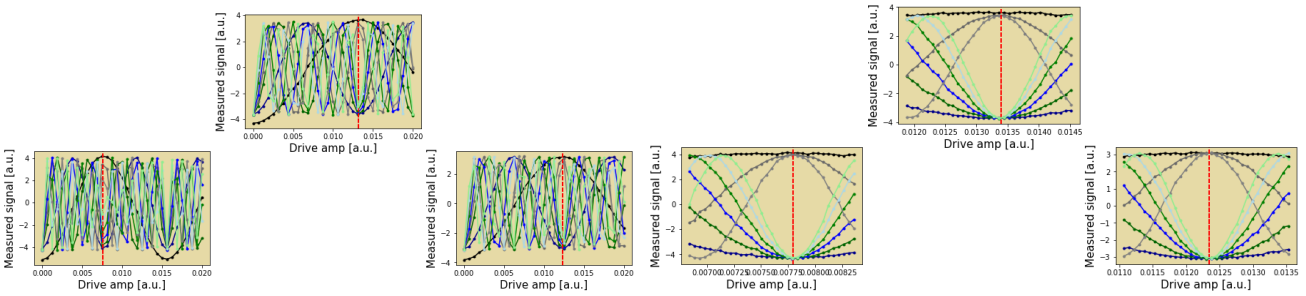
For the SX gate we can see different patterns from the standard rabi experiments. Here we can see the 3 levels - ground, excited and superposition of both states.



Wamf1 X pulse with duration 224

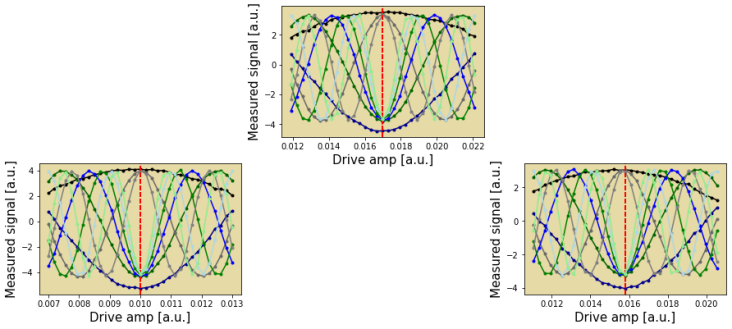
Wamf1 pulse is defined to mitigate the off-resonance errors and the rabi experiments do not show too much difference from the standard rabi experiment.





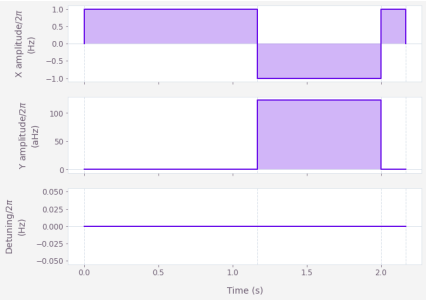
Wamf1 X pulse with duration 176

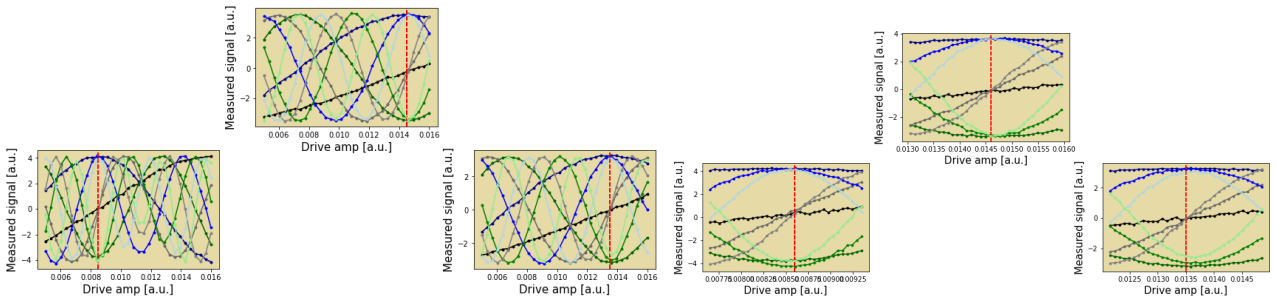
Default ibm X gates have length of 160, the Warm1 pulse with similar duration has low performance.



Corpse SX pulse with duration 608

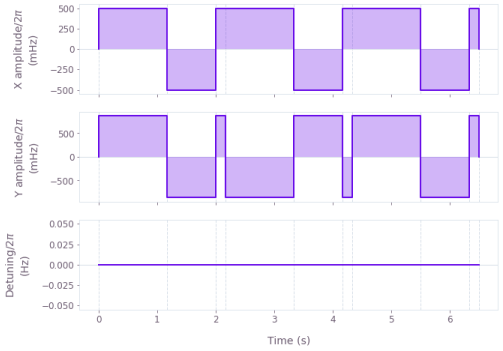
Similar to Wamf1 pulse Corpse pulse mitigates the off-resonance errors. But the overall pulse schedule is much longer.

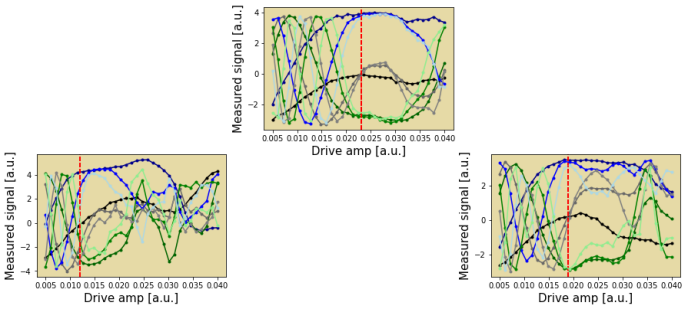




Corpse in Scrofulous SX gate with duration 1040

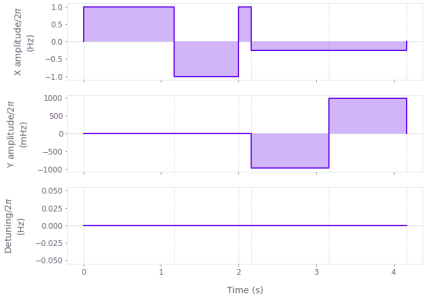
Corpse in scrofulous pattern combines the off-resonance and amplitude error mitigation. Combination of different pulses results in long schedules. Even with an overall duration of 1040 the pulse performance is pure.

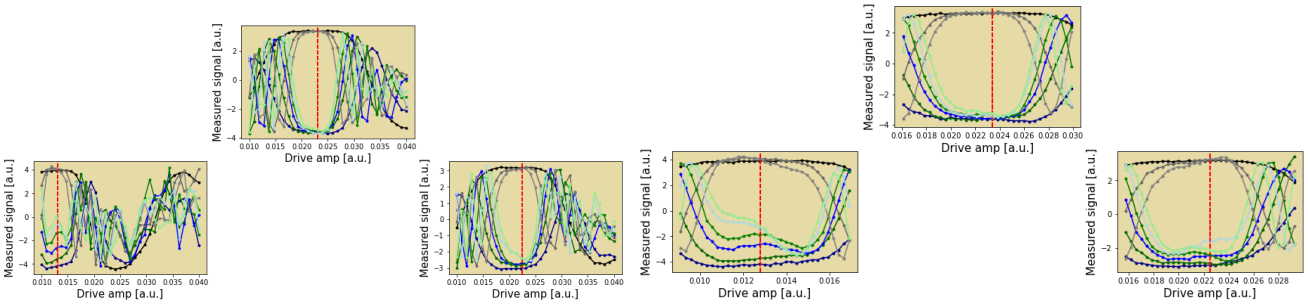




Corpse in SK1 X pulse with duration 816

This pulse presents better results with a shorter duration but it is still much longer than the default pulse.



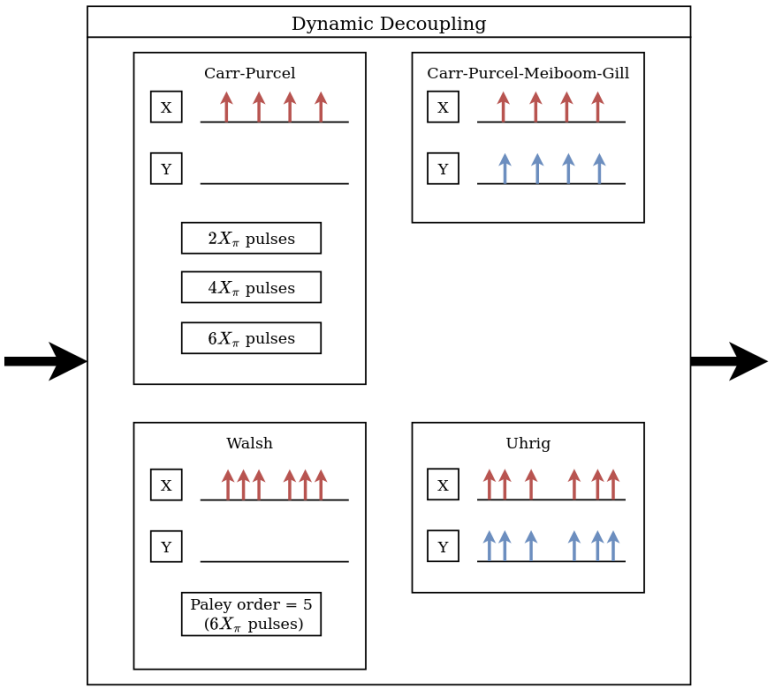


Application to other problems

Running additional calibration routines can improve the performance of all quantum circuits, but the default calibration on IBM quantum computers is good, especially up to a few hours after the automatic calibration is performed.

Long circuits could benefit from creating a custom gate with shorter duration than the default ones, but this would require many calibration steps and the benefits could be reduced by the higher frequency harmonics from the higher pulse amplitude. Additional Drag beta fine tuning can reduce the qubits shift to higher states.
Custom schedules based on PWC optimization requires benchmarking many quantum circuits which very often is not feasible. But the experiments (Baum et al.) are showing good results in keeping the high fidelity for many days after the initial calibration.
Off-resonance and pulse amplitude errors reduction patterns have longer durations than default IBM drag pulses. Longer circuits could have their performance degraded due to longer pulse duration which results in decoherence, shorter circuits are better suited for this optimization. When we have the description of the systematic error in our system predefined pulse schedules can be used with some benefits. Without the knowledge of the system error we would need to benchmark the behavior of different pulse schedules to find the optimal one for our system.

Dynamic decoupling

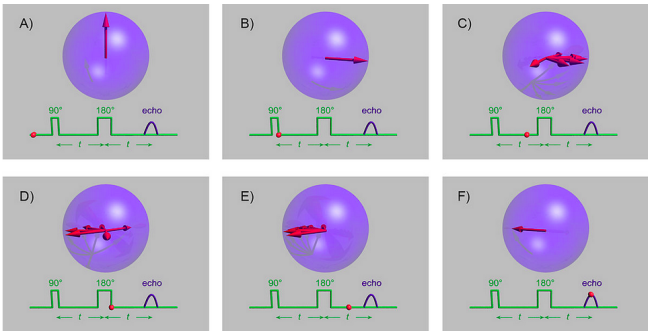


Generic introduction

All physical qubits in a quantum computer are susceptible to decoherence and one of the techniques to reduce the qubits environmental coupling is dynamic decoupling (DD). It is a series of periodic control pulses (most often in form of X_π or Y_π gates).

Dynamical decoupling (DD) is a form of quantum error suppression that modifies the system-environment interaction so that its overall effects are very nearly self-canceling, thereby decoupling the system evolution from that of the noise-inducing environment (Viola et al.).

Assuming that the Hamiltonian of the system is $H = H_B + H_{SB}$, where H_B includes all bathonly terms and H_{SB} includes all terms acting non-trivially on the system. The idea behind DD is to allow the system to evolve under the Hamiltonian dynamics and then applying a DD control pulse which refocuses the evolution towards the error-free ideal state (West et al.).



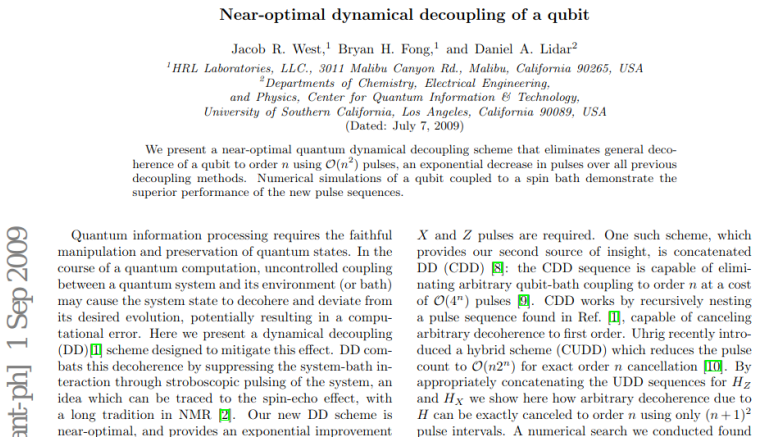
("Spin echo")

Every pulse introduces a single qubit error so the number of pulses in every sequence needs to be balanced in terms of gains of the DD schema and the noise for the pulses / gates.

Papers for further reading

1. **Near-optimal dynamical decoupling of a qubit**
   Jacob R. West,1 Bryan H. Fong,1 and Daniel A. Lidar
   https://arxiv.org/pdf/0908.4490.pdf

   Here we have a good comparison between dynamic decoupling schedules. The Quadratic schedule was described in this paper, which is proposed as an evolution of other commonly used DD sequences.



Near-optimal dynamical decoupling of a qubit

Jacob R. West,[1] Bryan H. Fong,[1] and Daniel A. Lidar[2]
[1]HRL Laboratories, LLC., 3011 Malibu Canyon Rd., Malibu, California 90265, USA
[2]Departments of Chemistry, Electrical Engineering, and Physics, Center for Quantum Information & Technology, University of Southern California, Los Angeles, California 90089, USA
(Dated: July 7, 2009)

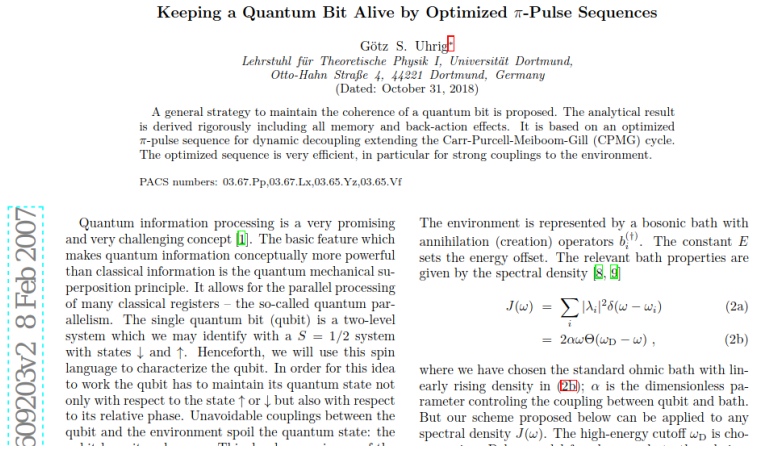
We present a near-optimal quantum dynamical decoupling scheme that eliminates general decoherence of a qubit to order n using $\mathcal{O}(n^2)$ pulses, an exponential decrease in pulses over all previous decoupling methods. Numerical simulations of a qubit coupled to a spin bath demonstrate the superior performance of the new pulse sequences.

Quantum information processing requires the faithful manipulation and preservation of quantum states. In the course of a quantum computation, uncontrolled coupling between a quantum system and its environment (or bath) may cause the system state to decohere and deviate from its desired evolution, potentially resulting in a computational error. Here we present a dynamical decoupling (DD)[1] scheme designed to mitigate this effect. DD combats this decoherence by suppressing the system-bath interaction through stroboscopic pulsing of the system, an idea which can be traced to the spin-echo effect, with a long tradition in NMR [2]. Our new DD scheme is near-optimal, and provides an exponential improvement

X and Z pulses are required. One such scheme, which provides our second source of insight, is concatenated DD (CDD) [8]: the CDD sequence is capable of eliminating arbitrary qubit-bath coupling to order n at a cost of $\mathcal{O}(4^n)$ pulses [9]. CDD works by recursively nesting a pulse sequence found in Ref. [1], capable of canceling arbitrary decoherence to first order. Uhrig recently introduced a hybrid scheme (CUDD) which reduces the pulse count to $\mathcal{O}(n2^n)$ for exact order n cancellation [10]. By appropriately concatenating the UDD sequences for H_Z and H_X we show here how arbitrary decoherence due to H can be exactly canceled to order n using only $(n+1)^2$ pulse intervals. A numerical search we conducted found

2. **Keeping a Quantum Bit Alive by Optimized π-Pulse Sequences**
   Gotz S. Uhrig
   https://arxiv.org/pdf/quant-ph/0609203.pdf

   Good theoretical analysis of dynamic decoupling. This paper introduces the now-called Uhrig pulse schedule.



Keeping a Quantum Bit Alive by Optimized π-Pulse Sequences

Götz S. Uhrig
Lehrstuhl für Theoretische Physik I, Universität Dortmund, Otto-Hahn Straße 4, 44221 Dortmund, Germany
(Dated: October 31, 2018)

A general strategy to maintain the coherence of a quantum bit is proposed. The analytical result is derived rigorously including all memory and back-action effects. It is based on an optimized π-pulse sequence for dynamic decoupling extending the Carr-Purcell-Meiboom-Gill (CPMG) cycle. The optimized sequence is very efficient, in particular for strong couplings to the environment.

PACS numbers: 03.67.Pp,03.67.Lx,03.65.Yz,03.65.Vf

Quantum information processing is a very promising and very challenging concept [1]. The basic feature which makes quantum information conceptually more powerful than classical information is the quantum mechanical superposition principle. It allows for the parallel processing of many classical registers – the so-called quantum parallelism. The single quantum bit (qubit) is a two-level system which we may identify with a $S = 1/2$ system with states $\downarrow$ and $\uparrow$. Henceforth, we will use this spin language to characterize the qubit. In order for this idea to work the qubit has to maintain its quantum state not only with respect to the state $\uparrow$ or $\downarrow$ but also with respect to its relative phase. Unavoidable couplings between the qubit and the environment spoil the quantum state: the

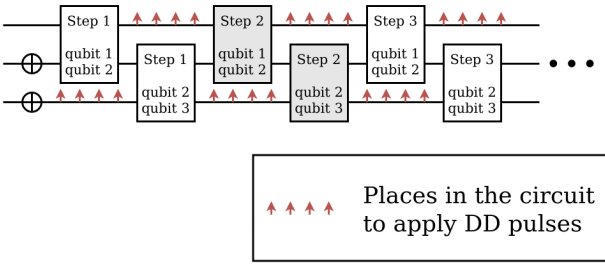
The environment is represented by a bosonic bath with annihilation (creation) operators $b_i^{(\dagger)}$. The constant E sets the energy offset. The relevant bath properties are given by the spectral density [8, 9]

$$J(\omega) = \sum_i |\lambda_i|^2 \delta(\omega - \omega_i) \quad (2a)$$
$$= 2\alpha\omega\Theta(\omega_D - \omega) , \quad (2b)$$

where we have chosen the standard ohmic bath with linearly rising density in (2b); α is the dimensionless parameter controling the coupling between qubit and bath. But our scheme proposed below can be applied to any spectral density $J(\omega)$. The high-energy cutoff ω_D is cho-

Application to Ising simulation

In the case of the 3 qubits Ising simulation we want to apply dynamic decoupling when there are long delays between operations on a specific qubit:.



The pulses are always applied in even patterns (in case of the X_π gates) so the final state of the qubit is not affected by the sequence (with the exception of the pulse noise) - the overall DD sequence represents an identity gate I.

Experiments

Code

We decided to implement our own dynamic decoupling function instead of using the default Qiskit function `DynamicDecoupling()`.
`add_dd()` function takes the quantum schedules as an input and adds dynamic decoupling pulses to places where there are long delays between next operations. This function allows us to benchmark different pulse patterns.
Usage:

```
schedule_qc = add_dd(schedule_qc)
```

Source code:

```
!pip install qctrl-open-controls
```

Install qctrl-open-control package to have access to predefine DD sequences.

```
from qctrlopencontrols import (new_carr_purcell_sequence,
                               new_uhrig_sequence,
                               new_walsh_sequence
                              )
def add_dd(schedules, pattern="Walsh_Paley5"):

    new_schedules = []

    for schedule in schedules:
        time_slots_for_dd = []

        instructions = schedule.instructions
        for idx, inst in enumerate(instructions):


            if isinstance(inst[1].channel, pulse.channels.DriveChannel):
                chn = inst[1].channels[0].index


                for idx_, inst_ in enumerate(instructions[idx+1:]):
                    if isinstance(inst_[1].channel, pulse.channels.DriveChannel):
                        chn_ = inst_[1].channels[0].index


                        if chn == chn_:



                            time_delta = inst_[0] - (inst[0] + inst[1].duration)


                            if time_delta > 2000:
                                time_slots_for_dd.append(
                                    {'start_time': inst[0] + inst[1].duration,
                                     'duration': time_delta, 'channel': chn})
                            break


        new_sched = Schedule.initialize_from(schedule)


        for time, inst in schedule.children:
            new_sched.insert(time, inst, inplace=True)



        for dd_slot in time_slots_for_dd:

            dd_ = inst_map_cal.get('x', dd_slot['channel'])

            dds = None
            if pattern == "Walsh_Paley5":
                dds = new_walsh_sequence(duration=dd_slot['duration'],
                                         paley_order = 5.,name='Walsh DDS')
            elif pattern == "CP_n2":
                dds = new_carr_purcell_sequence(duration=dd_slot['duration'],
                                        offset_count = 2.,name='Carr-Purcell DDS')
            elif pattern == "CP_n4":
                dds = new_carr_purcell_sequence(duration=dd_slot['duration'],
                                        offset_count = 4.,name='Carr-Purcell DDS')
            elif pattern == "CP_n6":
                dds = new_carr_purcell_sequence(duration=dd_slot['duration'],
                                       offset_count = 6.,name='Carr-Purcell DDS')
            elif pattern == "Uhrig_n4":
                dds = new_uhrig_sequence(duration=dd_slot['duration'],
                                         offset_count = 4.,name='Uhrig DDS')
            elif pattern == "Uhrig_n6":
                dds = new_uhrig_sequence(duration=dd_slot['duration'],
                                         offset_count = 6.,name='Uhrig DDS')

            for offset in dds.offsets:
                new_sched.insert(int(dd_slot['start_time']+offset-dd_.duration/2),
                                 dd_, inplace=True)
        new_schedules.append(new_sched)

    return new_schedules
```

```
Import DD sequences.



Core function takes schedules as an input and
returns the schedules with added DD sequences.

Create the new schedules array.

Iterate through the input schedules.


Iterate through the instruction in the schedule.


If the instruction is on the DriveChannel
store its DriveChannel index (qubit on which the
instruction acts).

Iterate through the next instruction to find the
next instruction acting on the same DriveChannel
index (qubit).


If the channels are equal that means we found the
instruction for qubit X and the next instruction
on the same qubit.

Now we can find the delay between instructions on
this qubit.

If the time is greater than our threshold (here
hardcoded to 2000) store the time information to
add DD sequences in the next steps

Break the loop looking for the second
instruction.

After the iteration through instruction is
finished we have the time_slots_for_dd array with
time slices on which we can apply DD.

Create our new schedule based on the initial one.
Insert all instructions for the initial schedule
to our new one to have the schedule duplicated.


For every time slot with a delay.

dd_ will be our Xpi pulse schedule from the
global instruction map.

Get the DD parameters from the
qctrl-open-controls package.


We are interested only in the pulse positions
(dds.offsets array).
For every offset time we append our Xpi pulse
schedule.

Return the updated schedule.
```

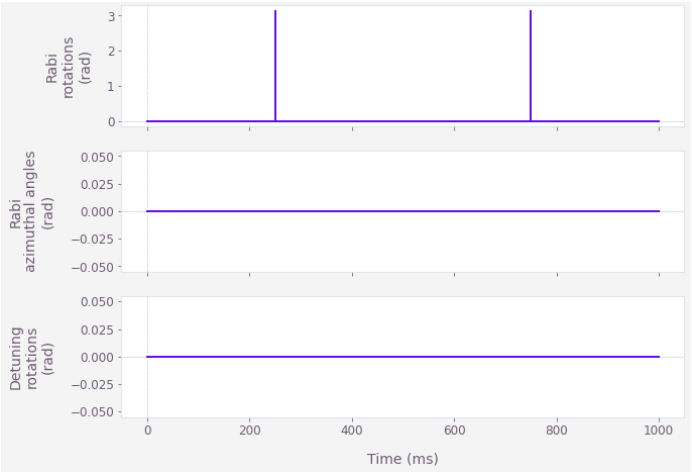
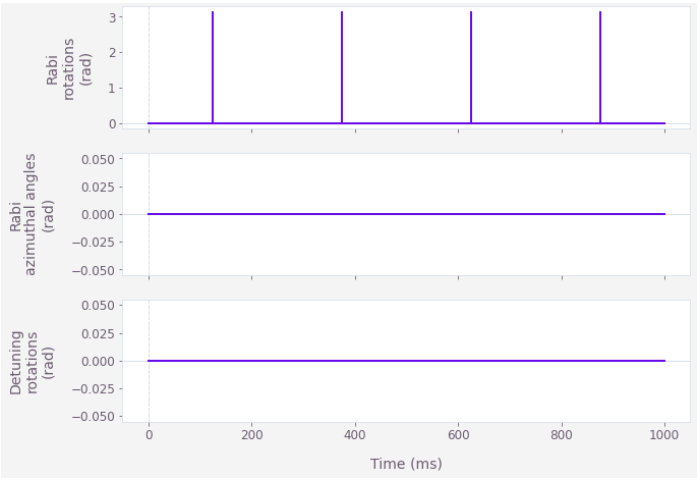
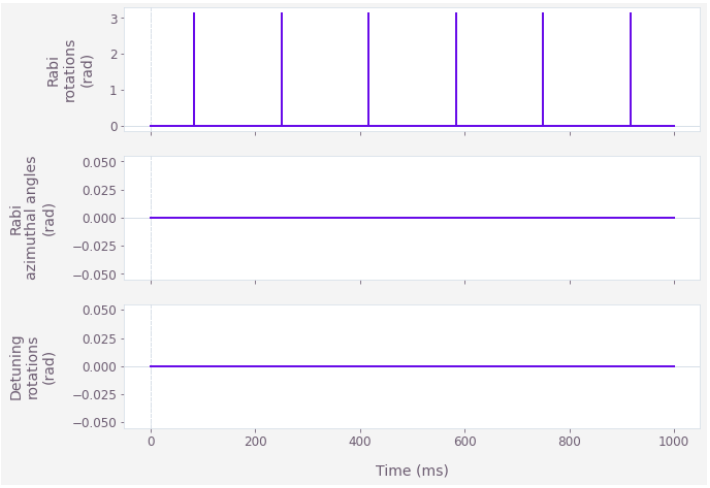
Carr-Purcel

Car-Purcel (Carr et al.) (CP) is one of the simplest pulse sequences using only X_π operations. The offsets of the pulses are calculated by formula:

$$t_i = \frac{\tau}{n}\,(i\ -\ \frac{1}{2}),$$

where t_i is the offset for i-th pulse, τ is the duration of the DD sequence and n is the number of pulses in the sequence.

This formula results in equally spread out X_π operations, ex:

Parameters	Graph	Pulse offsets
τ = 1s, n = 2		0.25, 0.75
τ = 1s, n = 4		0.125, 0.375, 0.625, 0.875
τ = 1s, n = 6		0.0833.., 0.25, 0.4166.., 0.5833.., 0.75, 0.9166..

Where (“DynamicDecouplingSequence — Open Controls Python package | Q-CTRL”):

- Rabi rotations - the rabi rotation ω_j at each time offset
- Rabi azimuthal angles - the azimuthal angle ϕ_j at each time offset
- Detuning rotations - the detuning rotation δ_j at each time offset

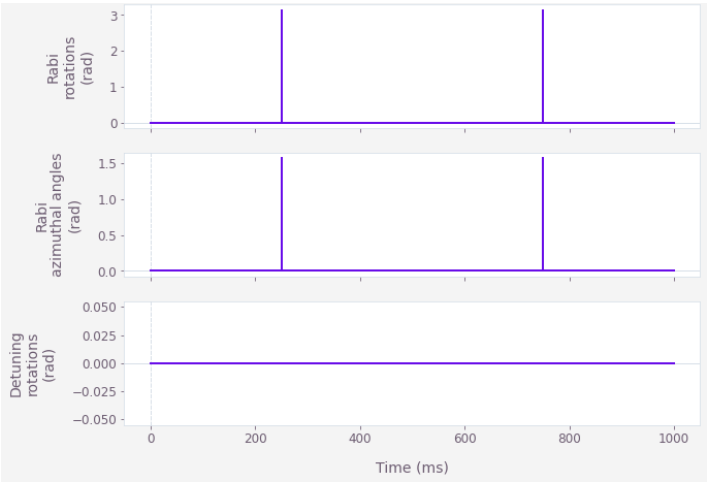
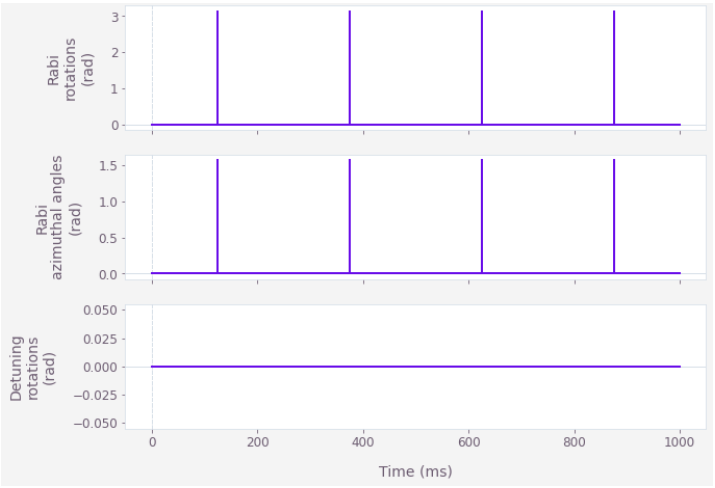
The DD operation at t_j is parametrized by equation:

$$U_j\ =\ \exp[-\frac{i}{2}\,(\omega_j \cos\phi_j \sigma_x + \omega_j \sin\phi_j \sigma_y + \delta_j \sigma_z)]$$

Carr-Purcell-Meiboom-Gill

The Carr-Purcell-Meiboom-Gill (CPMG) sequence uses the formula for pulse offsets as the Carr-Purcel sequence. But the improvements results from applying Y_{π} operations with the X_{π}rotations.

Ex:

Parameters	Graph	Pulse offsets
τ = 1s, n = 2		0.25, 0.75
τ = 1s, n = 4		0.125, 0.375, 0.625, 0.875

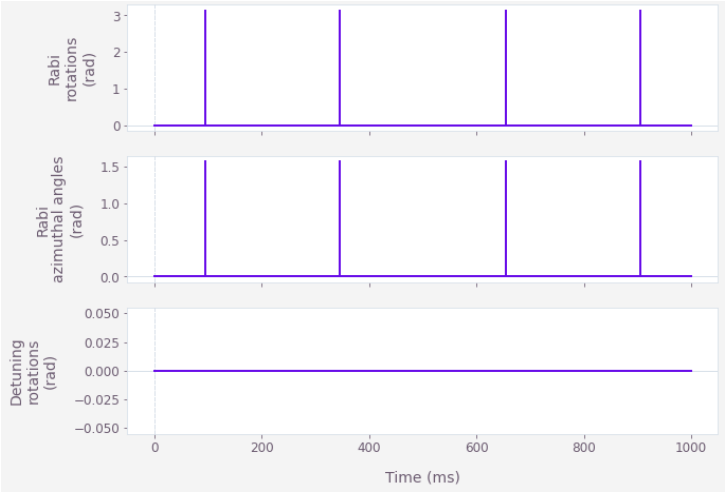
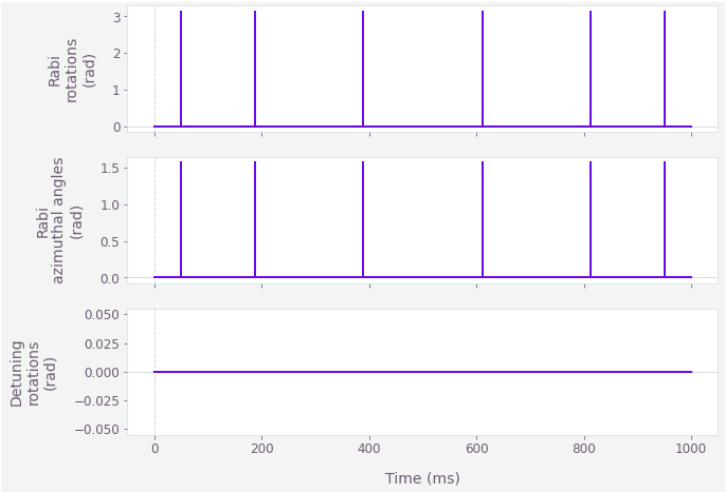
Uhrig

The paper presented by G. S. Uhrig (Uhrig) describes an optimized sequence of non-equidistant π-pulses, which results in better suppression of environmental coupling using a similar number of pulses as the CPMG scheme.
The offsets of the pulses are calculated by formula:

$$t_i = \tau \sin^2\left(\frac{i\pi}{2(n+1)}\right)$$

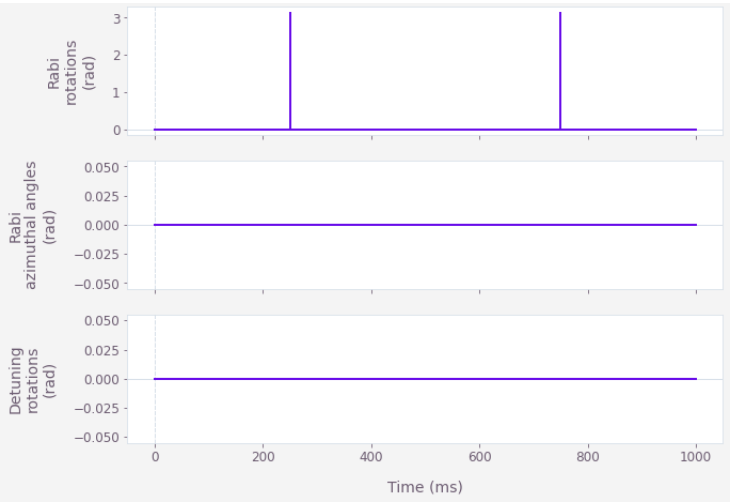
where t_i is the offset for i-th pulse, τ is the duration of the DD sequence and n is the number of pulses in the sequence.

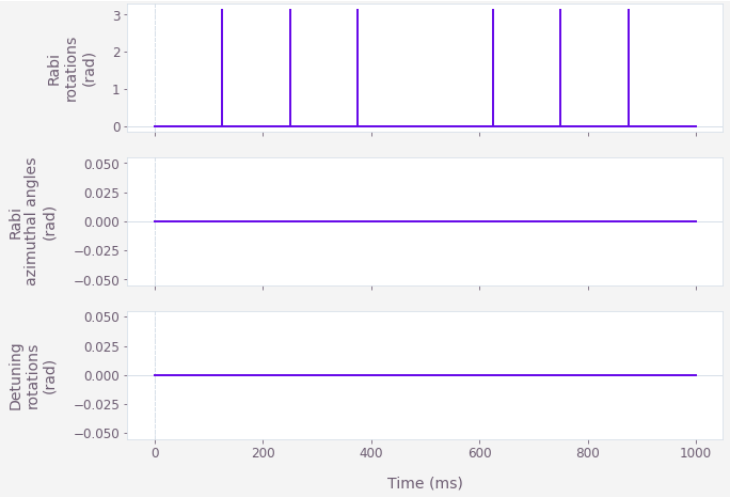
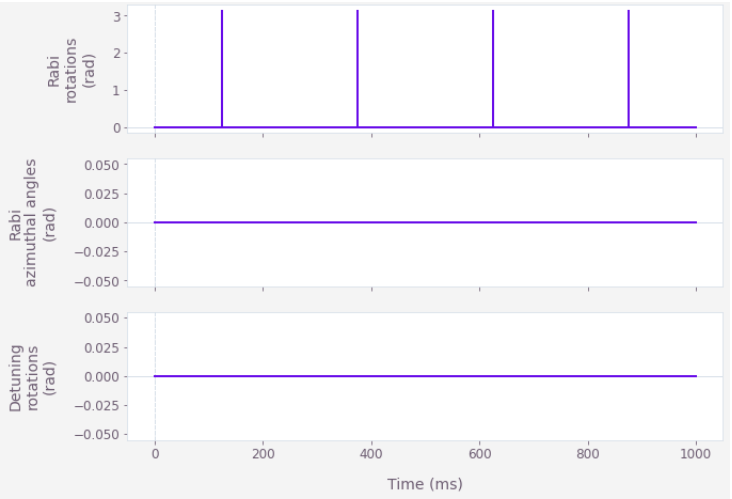
Parameters	Graph	Pulse offsets

τ = 1s, n = 4		0.0955, 0.3455, 0.6545, 0.9045
τ = 1s, n = 6		0.0495, 0.1883, 0.3887, 0.6113, 0.8117, 0.9505

Walsh

The pulse offsets in the Walsh sequence (Ball and Biercuk) are defined using the Walsh function (Rademacher). The sequence is parametrized by duration τ and Paley order k.
Ex.

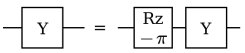
Parameters	**Graph**	**Pulse offsets**
τ = 1s, k = 3		0.25, 0.75 Same as Carr-Purcel: τ = 1s, n = 2

τ = 1s, k = 5		0.125, 0.25, 0.375, 0.625, 0.75, 0.875
τ = 1s, k = 6		0.125, 0.375, 0.625, 0.875 Same as Carr-Purcel: τ = 1s, n = 4

The only interesting sequence is the one with Paley Order k=5, other Paley orders result in an even number of pulses, the pulse pattern similar to Carr-Purcel or high number of pulses.

Results

IBM quantum systems do not have Y_{π} gates calibrated by the default, but are decomposition into RZ and X gates:

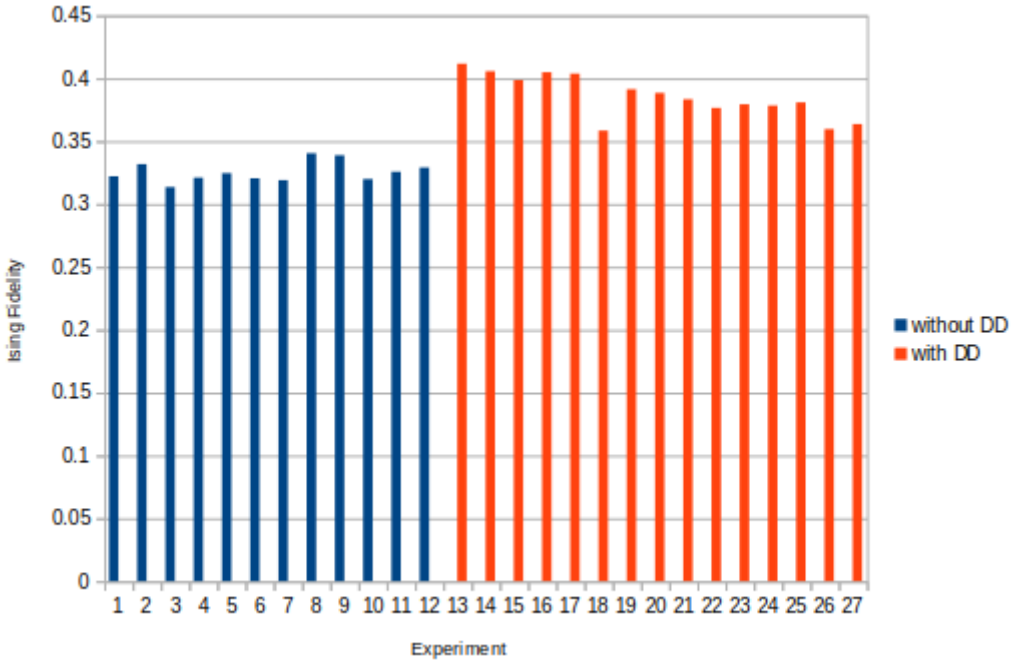


Due to this factor benchmarking was focused on the sequences using only single X_{π} gates for single pulse.

To benchmark the dynamic decoupling sequences we used the following settings:

- Trotter steps set to 7.
- Circuit is constructed using RZX operation.
- X and √X gates are calibrated using qiskit `Fine<X,SX>AmplitudeCal()` and `Fine<X,SX>DragCal()` functions.

From our experiments we can clearly see the advantage of dynamical decoupling patterns:



We wanted to test the 3 qubit Ising model fidelity using Carr-Purcel and Walsh sequences (with Paley order k = 5, other settings result in the CP pulses). Schedules based on Uhrig pattern with only X_π operations were additionally added. But we were not able to run enough experiments to have any conclusive reports.

Future research possibilities

We did not benchmark DD patterns using Y_π gates - Carr-Purcell-Meiboom-Gill and Uhrig sequence should result in reduction of the dephasing noise.

DD patterns are most often used with even number of X_π pulses so the resulting state is expected to be the same as without the compensation. The pulses still represent quantum gates which result in imprecise output state, due to the gate noise. More optimized X_π pulses could result in better state fidelity, optimal control could be applied in this case with some possible benefits.

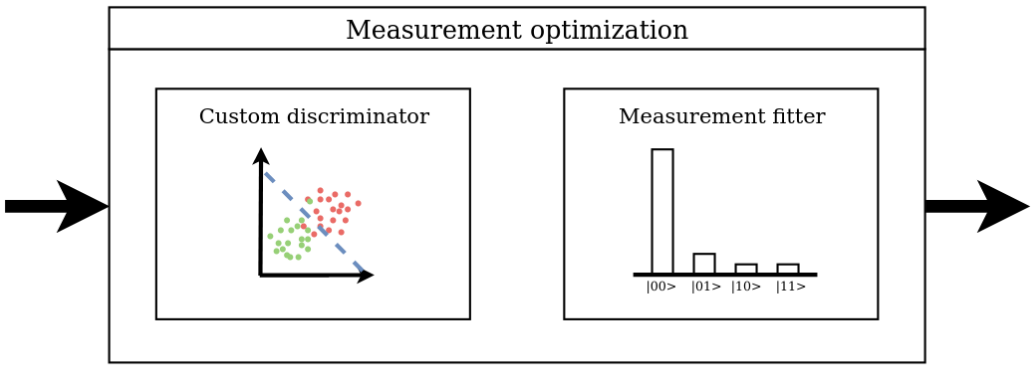
The pulse pattern has the goal to reduce the environmental coupling - better noise models of the specific system can help to create more robust DD pulses. This follows the idea from (West et al.), where Uhrig (Uhrig) sequence was expanded to a more generic noise model.

Quantum circuits without delays, especially with 2 qubits gates, are susceptible to crosstalk errors (Sarovar et al.) and further research into applying delays with dynamic decoupling pulses would be needed.

Application to other problems

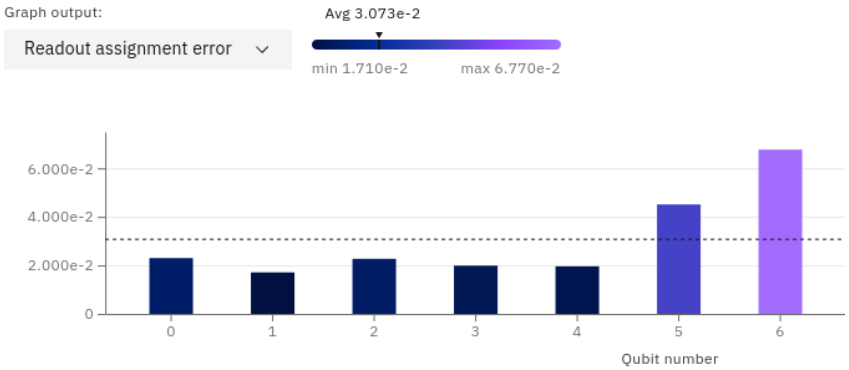
Dynamic decoupling is a very simple technique with good benefits, regardless of the quantum computer architecture [Trap ions (Arrazola et al.), photonic (Bardhan et al.)]. It should be used in all quantum circuits which include delays or long time between gates.

Measurement optimization

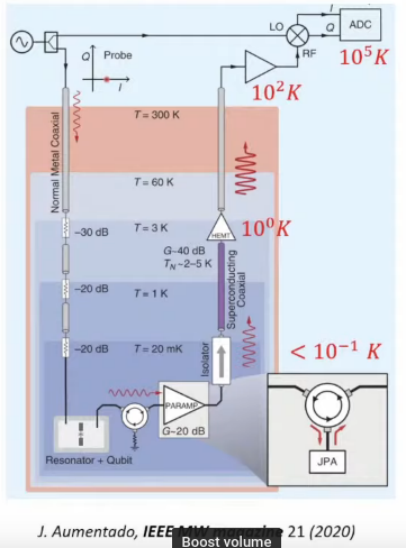


Generic introduction

Measurement of quantum state has one of the highest error rates in current quantum systems - in IBM systems the error can be around 2%.
Error rates on ibmq_jakarta on 10 march 2022:



Typical readout power is in the range of -130dBm (“Microwave Amplifiers for Quantum Information Processing | Seminar Series with Florent Q. Lecocq”), which is very close to room temperature equipment, so ideally we would need to integrate the signal over a long period of time, but this leads to decoherence.
This problem leads to a need for microwave amplifiers which can work close to the quantum circuit in the low temperature environment.
Standard measurement setups for quantum circuits is:



Many interconnected amplifiers and the thermal noise lead to measurement inaccuracies.

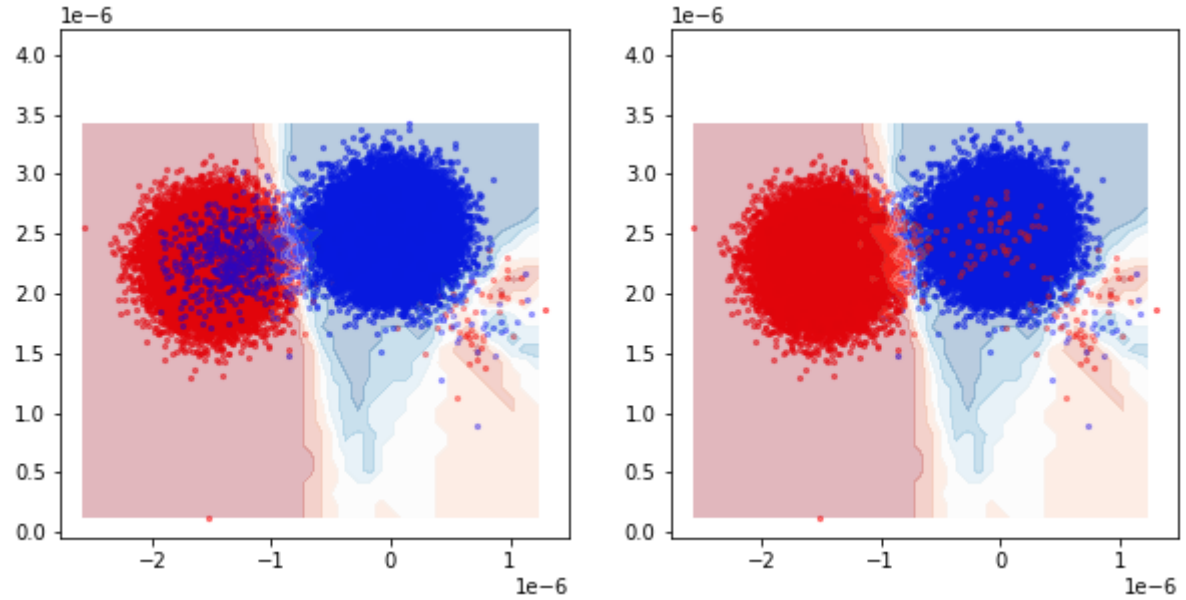
The real outcome of a superconducting qubits computer are I and Q values of a microwave signal transmitted or reflected by the readout resonators. Those values are then mapped into ground and excited states, this is done by “discriminator”. Discriminator is a generic object which can take broad input values and classify them into predefined finate states.
Most common algorithms used as classifier (“Classifier comparison — scikit-learn 1.0.2 documentation”):

- k-Nearest Neighbors (“k-nearest neighbors algorithm”) - the object is classified most common class of its k nearest neighbors. This is a very simple technique but showing good results for quantum discriminators.
- support-vector machine (SVM) (“Support-vector machine”) - we want to find the maximum-margin hyperplane that divides our input points into our classes. We can use linear functions to define the hyperplane or define non-linear kernels (ex. radial basis function kernel). The

complexity for SVM is in order of $O(n^2)$ (Chang and Lin). Quantum measurement often requires tens of thousands of shots to get an accurate readout, and creating a SVM classifier based on that number of samples is quite long.

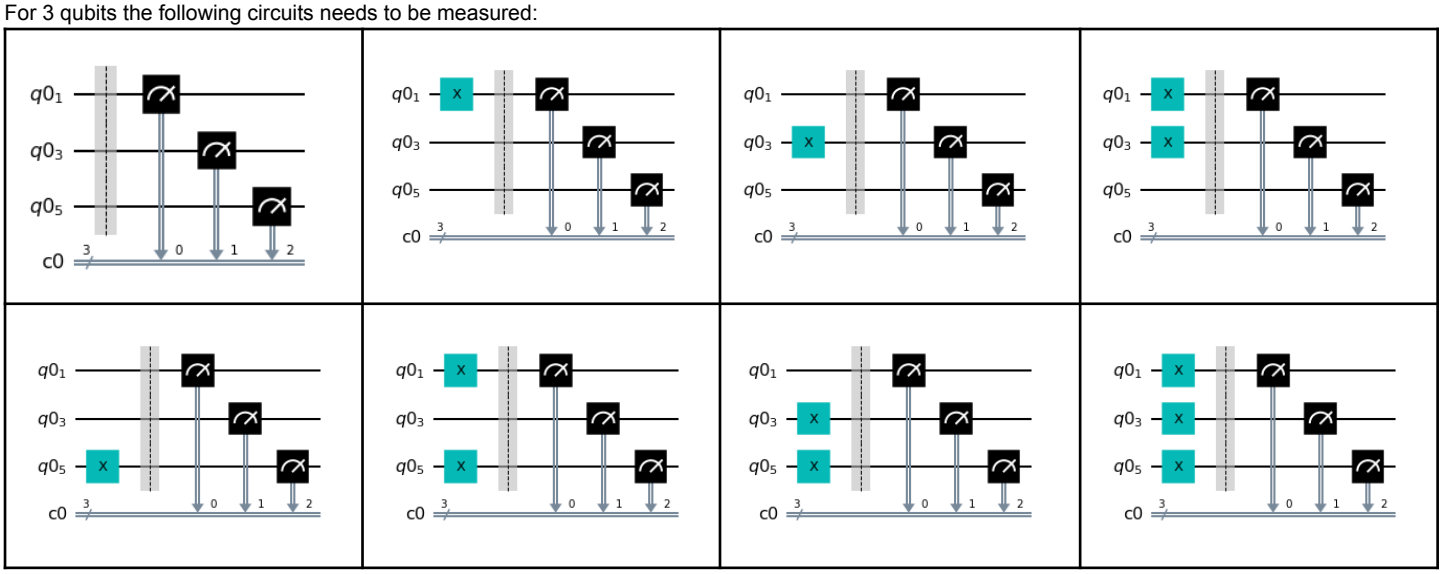
- Decision tree classifier (“Decision tree learning”) - a tree structure object is created which splits the data into sub-trees with the final leafs of the tree pointing to the predicted class. This technique is very sensitive to the training data, if we don't have a broad enough distribution the decision tree classifier will not perform well (small change in the training data can result in large change in the tree), this is quite limiting for usage in quantum systems where slow drifts are changing the system.
- Neural networks (“Neural network”) - a black box model which consists of many layers of matrix operation, which is iteratively improved via gradient descent methods. Neural networks classifiers are showing great progress in many fields, and could be used to create better quantum discriminators.
- Naive Bayes (“Naive Bayes classifier”) - probabilistic classifiers which assign probability of specific class based on the input.

Example of measurements of ground (red) and excited (blue) states:

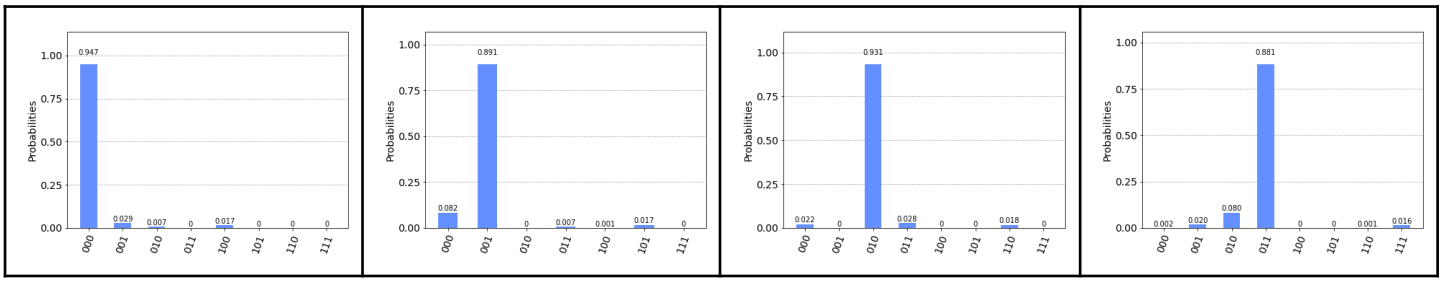


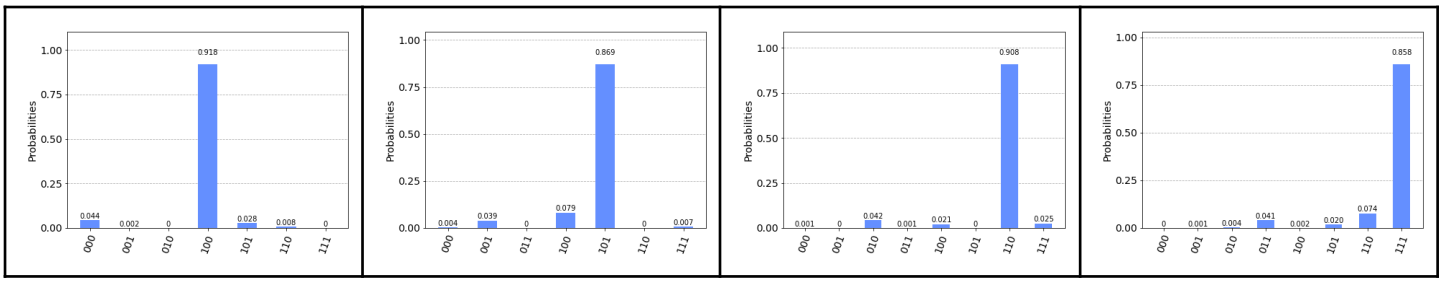
Both plots represent the same measurement but on the left side the excited state is plotted in front and on the right the ground state is plotted in the front. The background color coding is based on 10 nearest neighbors classifiers.

The quantum measurement systems are often biased to specific quantum states (Nation et al.). To understand the noise we need to perform full Hilbert space measurements.
For 3 qubits the following circuits needs to be measured:

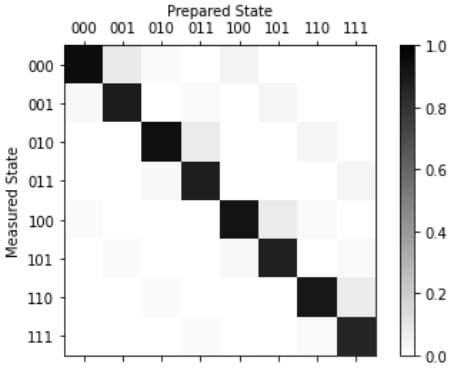


In the generic case we need 2^qubits circuits.
In the ideal case all the above circuits shall return pure states but the real measurement includes other states:
Ex.





Those measurements can be combined into a matrix:



The idea behind the measurement filter is to create a filter object which removes the bias from the final measurement.

Paper for further reading

1. **Improving readout performance in superconducting qubits using machine learning**
   https://docs.q-ctrl.com/boulder-opal/application-notes/improving-readout-performance-in-superconducting-qubits-using-machine-learning

   Good introduction to creating custom classifiers for quantum measurement. With code and experiments. But this code requires proprietary libraries.

Improving readout performance in superconducting qubits using machine learning

Using Q-CTRL discriminators and optimized measurement parameters to boost readout performance

Boulder Opal provides an efficient toolset to include all aspects of quantum computer operation, from control optimization through to readout.

In superconducting qubit experiments, the problem of correctly identifying the state of a qubit as $|0\rangle$ (ground) or $|1\rangle$ (excited) corresponds to the problem of discriminating between different values of th (I, Q) components of a microwave signal transmitted or reflected by the readout resonators. Incorrect assignment of qubit states to analog readout values limits the performance of quantum circuits and even runtime optimizations where so-called SPAM inhibits hardware performance improvement.

In this application note, we demonstrate how efficient machine learning tools can be used to improve discrimination performance in superconducting qubits. We will cover:

- Applying machine learning techniques to improve state discrimination in the (I, Q) plane
- Optimizing measurement pulses to enhance discrimination on IBM Quantum hardware
- Deploying maximum likelihood estimation for state assignment from a measurement record.

Ultimately we will demonstrate how to reduce SPAM errors from $\sim 14\%$ to $\sim 1\%$ on a real quantum computer by a combination of these methods, providing a framework for general enhancement of readout performance in any superconducting circuit.

This notebook contains preview features that are not currently available in Boulder Opal. Please contact us if you want a live demonstration for your hardware. Preview features are included through the package `qctrlcore`, which is not publicly available: cells with `qctrlcore` will not be executable.

Some cells in this notebook require an account with IBM-Q to execute correctly. If you want to run them, please go to the IBM-Q experience to set up an account.

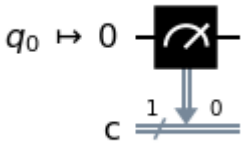
Application to Ising model

Custom discriminator is a universal technique which can be applied to Ising simulation and other problems.
Measurement filters can be easily applied with low cost for the Ising simulation especially for low qubit simulation. This is one of the simplest and most used techniques for error mitigation.

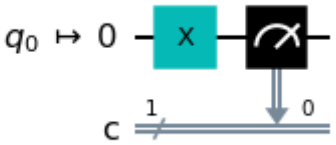
Experiments

Custom discriminator

Qiskit pulse control allows adding custom discriminators based on sklearn classifier objects.
To create a discriminator we need to gather the data for ground and excited states. For the ground state we need the simplest quantum circuit - just a measurement of a qubit which is reseted to the ground state:



To gather the data for the excited state additional X gate is added:



Additional sweep of the amplitude of the measurement channel was benchmarked.

Utils functions:

```
def measurement_circuit(qubits, final_state, delay=None):
    qc = QuantumCircuit(backend.configuration().n_qubits, len(qubits))

     if final_state == 'ext':
         qc.x(qubits)

    if delay is not None:
        qc.delay(delay, qubits)

    qc.measure(qubits, list(range(len(qubits))))


    return qc

def add_measurement_cal(schedule_qc, amp_mul=None, discriminators=None):




    if amp_mul is None:
        measurement_amp_mult = [1.0]*7
    else:
        measurement_amp_mult = [amp_mul]*7

    if discriminators is not None:
        measurement_amp_mult = []
        for discr in discriminators:
            measurement_amp_mult.append(discr['settings']['amp_mul'])

    schedule_qc_ = []
    for sched_ in schedule_qc:

        new_sched_ = Schedule.initialize_from(sched_)




        for time, inst in sched_.children:


            if isinstance(inst.channels[0], AcquireChannel):


                new_acquire_sched_ = Schedule()


                for time_, inst_ in inst.children[1][1].children:
                    if isinstance(inst_.channels[0], MeasureChannel)
                       and isinstance(inst_, pulse.instructions.play.Play):

                        new_acquire_sched_.insert(time_, Play(GaussianSquare(
                            duration=inst_.operands[0].duration,
                            amp=inst_.operands[0].amp * \
                                 measurement_amp_mult[inst_.channel.index],
                            sigma=inst_.operands[0].sigma,
```

```
Function to create the circuits for
ground and excited state.

The excited state is generated by
adding an X gate.

Delay could be added if needed.


Add default measurement schedule which
can be later updated with amplitude
change or with custom discriminator.

Return the quantum circuit


This function updates the measurement
parameters in the schedules. It can
change the amplitude of the measurement
pulse and add a custom discriminator.



Setup the default values.

If amplitude multiplier (amp_mul) was
given this value will be applied to all
qubit pulses.

If a discriminator object is provided
then the amplitude multiplier (amp_mul)
will be taken from the discriminator
settings.


Iterate through all input schedules.

New schedule will be created and
children will be copied from the
initial schedule. This includes the
schedule name which is needed for state
tomography.

Iterate through all children's
schedules.

Look for the AcquireChannel schedules -
this indicates the measurement pulses,
which we want to modify.

Create a new schedule
(new_acquire_sched) which will be our
amplitude modified pulse.


Look for the MeasureChannel and Play
instruction.


Now we insert a GaussianSquare pulse to
our new_acquire_sched using the default
parameters from the input schedule,
with the exception of the amplitude
which is multiplied by our custom
value.
```

```
                            width=inst_.operands[0].width,
                            name=inst_.operands[0].name),
                        MeasureChannel(inst_.channel.index),
                        name=inst_.name), inplace=True)
                else:
                    if isinstance(inst_, Acquire):


                        aq_idx = inst_.channels[0].index


                        if discriminators is not None:
                            if discriminators[aq_idx] is not None:

                                discriminator = discriminators[aq_idx]
                                                               ['discriminator']
                                discriminator.name = "quadratic_discriminator"



                                discriminator.params = {}

                                new_acquire_sched_.insert(time_, Acquire(
                                     inst_.operands[0],
                                     AcquireChannel(aq_idx),
                                   MemorySlot(inst_.operands[2].index),
                                discriminator=discriminator), inplace=True)
                            else:
                                new_acquire_sched_.insert(time_, inst_, inplace=True)
                        else:
                            new_acquire_sched_.insert(time_, inst_, inplace=True)
                    else:
                        new_acquire_sched_.insert(time_, inst_, inplace=True)
            new_sched_.insert(time, new_acquire_sched_, inplace=True)

        else:
            new_sched_.insert(time, inst, inplace=True)

    schedule_qc_.append(new_sched_)

return schedule_qc_
```

For the Acquire instruction we want to add a custom discriminator.

aq_idx is the qubit index on which we operate.

If discriminators for this qubit were provided.

Discriminator is taken from the function input.

We need to add to it a name - without this step we are getting an qiskit runtime error.

And we need to add parameters to avoid runtime errors.

Now we insert the new Acquire instruction with our discriminator object.

If discriminators were not provided we insert to our new acquire schedule the parameters from the parent schedule.

We insert the newly created acquire schedule (new_acquire_sched_) to our new modified schedule (new_sched_).

New schedules are combine into an array as the function input schedule

New schedules are returned.

Code to run the benchmark:

```
jobs_discriminator = []
circuits_discriminator = []
schedules_discriminator = []

qubits = [0,1,2,3,4,5,6]

experiments = [
    {'duration': 22400, 'amp_mul': 1.0, 'qubits': qubits},
    {'duration': 22400, 'amp_mul': 1.2, 'qubits': qubits},
    {'duration': 22400, 'amp_mul': 1.4, 'qubits': qubits},
    {'duration': 22400, 'amp_mul': 1.6, 'qubits': qubits},
    {'duration': 22400, 'amp_mul': 1.8, 'qubits': qubits},
    {'duration': 22400, 'amp_mul': 2.0, 'qubits': qubits},
    {'duration': 22400, 'amp_mul': 2.2, 'qubits': qubits},
    {'duration': 22400, 'amp_mul': 2.4, 'qubits': qubits},
    {'duration': 22400, 'amp_mul': 2.6, 'qubits': qubits},
    {'duration': 22400, 'amp_mul': 2.8, 'qubits': qubits},
]

for exp in experiments:

    qc_gnd = measurement_circuit(exp['qubits'], final_state='gnd', delay=0)
    qc_ext = measurement_circuit(exp['qubits'], final_state='ext', delay=0)

    schedule_gnd = build_schedule(qc_gnd, backend, method='alap')


    schedule_gnd = add_measurement_cal([schedule_gnd], amp_mul=exp['amp_mul'])[0]


    schedule_ext = build_schedule(qc_ext, backend, method='alap')
    schedule_ext = add_measurement_cal([schedule_ext], amp_mul=exp['amp_mul'])[0]

    job = execute([schedule_gnd, schedule_ext]*4, backend, meas_level=1,
                  meas_return='single', memory=True, shots=30000,
                  optimization_level=0)
```

Arrays to store the job, circuits and schedules used for the benchmark.

We can run the measurement benchmark for all qubits with one circuit.

Create the experiment settings.

For every experiment

Create the quantum circuits in ground (qc_gnd) and excited states (qc_ext)

Convert the quantum circuit into schedules

Add measurement pulses according to our experiments

Execute the job, returning the pure I, Q values from the readout (meas_levl=1, meas_return='single').

`    circuits_discriminator.append([qc_gnd, qc_ext]*4)`
`    schedules_discriminator.append([schedule_gnd, schedule_ext]*4)`
`    job_monitor(job)`
`    jobs_discriminator.append(job)`
	Store the job input and results into a global array.

Create Discriminators using k-Nearest Neighbors classifier:

`import sklearn`
`from sklearn.neighbors import KNeighborsClassifier`
`from qiskit.ignis.measurement import SklearnIQDiscriminator`
	Qiskit allows us to use sklearn classifiers.

`discriminators = []` `scale_factor = 1e-14`	discriminators array will contain the best discriminator for every qubit.

`for qubit_idx in qubits:`
`    best_score = 0`
`    best_settings = None`
`    discriminator = None`
	Iterate through all qubits

`    for job_idx, job in enumerate(jobs_discriminator):`	Iterate through all experiments which we run. This will loop through all amplitudes which we checked.

`        gnd_results = job.result().get_memory(0)[:, qubit_idx]*scale_factor`
`        exc_results = job.result().get_memory(1)[:, qubit_idx]*scale_factor`
`        for circ_idx in range(1,4):`
`            gnd_results = np.concatenate([gnd_results,`
`                job.result().get_memory(circ_idx*2 + 0)[:, qubit_idx]*scale_factor])`
`            exc_results = np.concatenate([exc_results,`
`                job.result().get_memory(circ_idx*2 + 1)[:, qubit_idx]*scale_factor])`
	Extract all the I,Q values for every execution for the ground and excited circuits.

`        gnd_results = np.array([[np.real(x), np.imag(x)] for x in gnd_results])`
`        exc_results = np.array([[np.real(x), np.imag(x)] for x in exc_results])`
 	
`        x = np.concatenate([gnd_results, exc_results])`
	Split the real and imaginary parts of the quantum readout. The gnd_results array will have the shape: (num_of_circuit_executions, 2), where [:, 0] is the real part and [:,1] is the imaginary part.

`        y = np.array([0]*len(gnd_results)+[1]*len(exc_results))`	y is our labels - 0 for the ground state and 1 for the excited states.
`        classifier = KNeighborsClassifier(100)`	Create k-nearest neighbor classifier with k=100.
`        classifier.fit(x,y)`	Fit the model to our data.
`        score = classifier.score(x, y)`	Check the classifier performance.

`        if score > best_score:`
`            best_score = score`
`            best_settings = experiments[job_idx]`
	Through all the jobs iteration we want to select the best classifier. Everytime our classifier is better than the previous we store the better one.

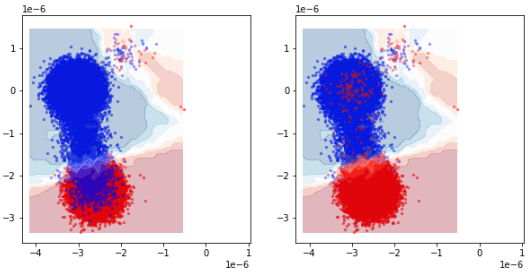
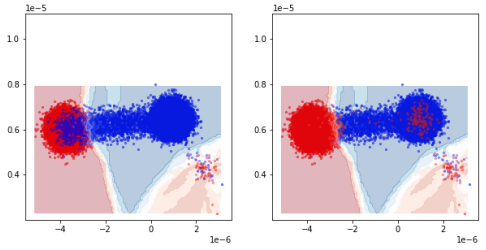
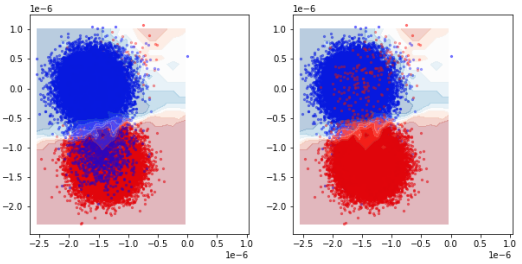
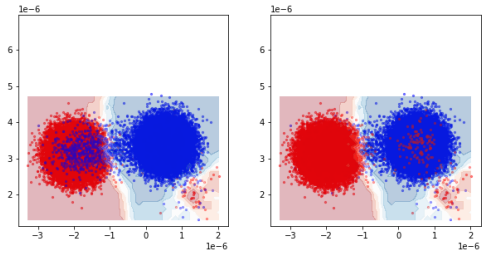
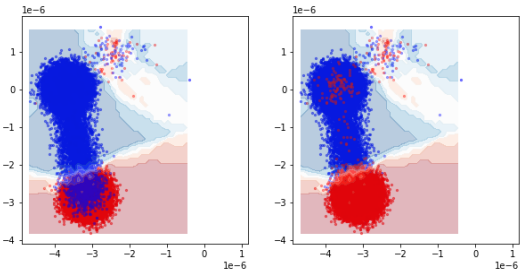
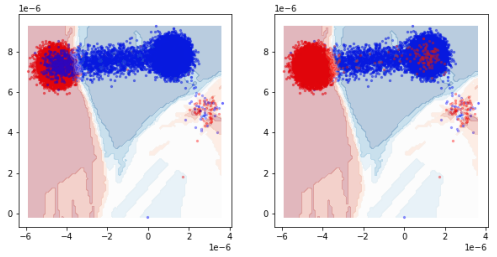
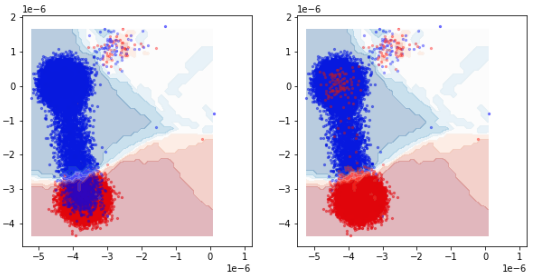
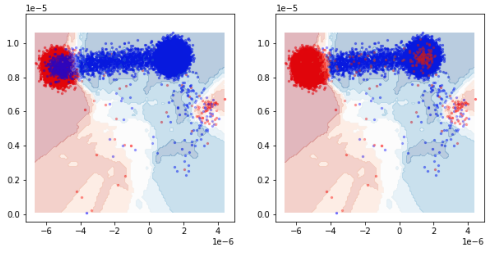
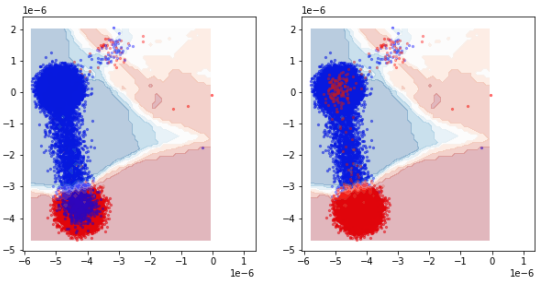
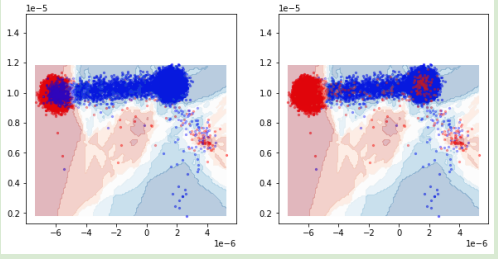
`            discriminator = SklearnIQDiscriminator(KNeighborsClassifier(100),`
`                                job.result(), [qubit_idx],`
`                                expected_states=[0,1], standardize=True,`
`                                schedules=schedules_discriminator[job_idx][:2])`
	Create the Sklearn based qiskit discriminator.

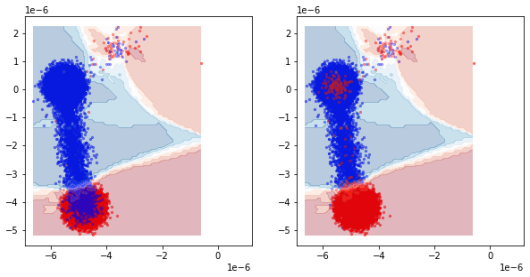
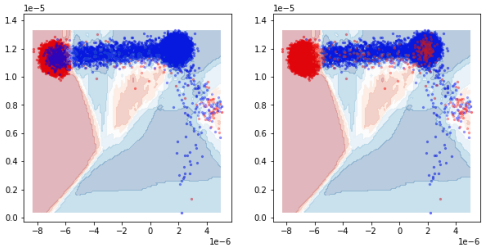
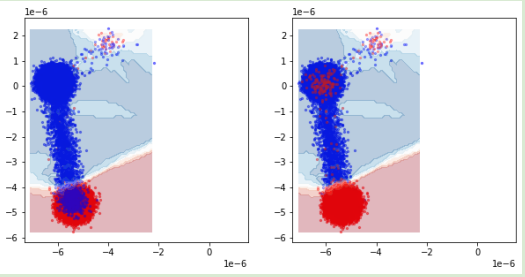
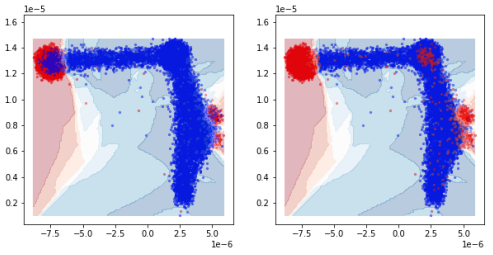
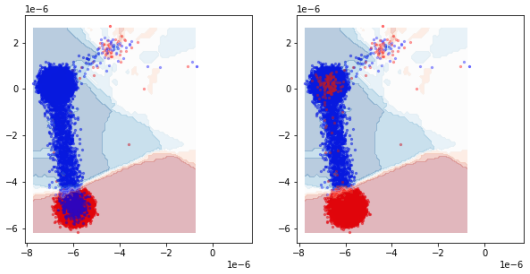
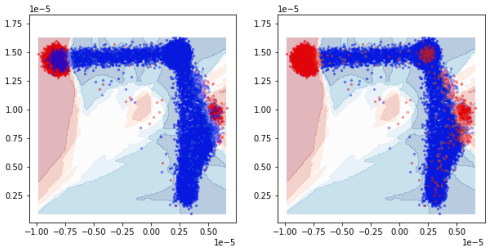
`    discriminators.append({'discriminator': discriminator,`
`                           'score': best_score,`
`                           'settings': best_settings})`
	After iterating through all the jobs - Store the discriminator into a global array with additional information about the settings which includes the best amplitude multiplier value.

`import pickle`
`timestr = time.strftime("%Y%m%d-%H%M%S")`
`if not os.path.exists("./discriminators"):`
`    os.makedirs("./discriminators")`
`pickle.dump(discriminators,`
`    open("./discriminators/discriminators_" + timestr + ".pickle", "wb"))`
	Store created discriminators locally, using pickle.

Higher measurement amplitude increases the distance between ground and excited states:

	Qubit 0	Qubit 5

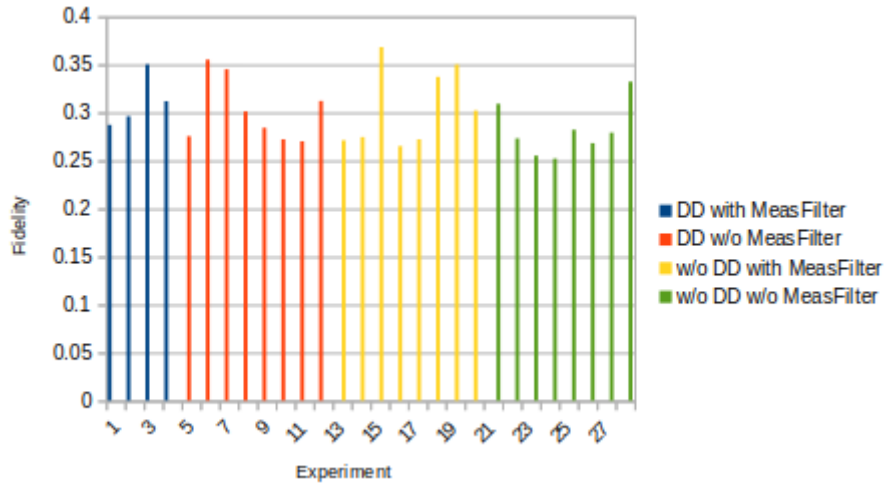
Default amplitude	 Both plots are based on the same run but with ground (blue) or excited (red) state in the front. Classifier performance: 0.9759	 Classifier performance: 0.9861
Duration = Default Pulse duration / 2, Amplitude = Default amplitude	 Classifier performance: 0.9747	 Classifier performance: 0.9868
Amplitude = Amplitude * 1.2 Classifier performance: 0.9749	 Classifier performance: 0.9806	 Classifier performance: 0.9870
Amplitude = Amplitude * 1.4		 Classifier performance: 0.9874
Amplitude = Amplitude * 1.6	 Classifier performance: 0.9826	 Classifier performance: 0.9882

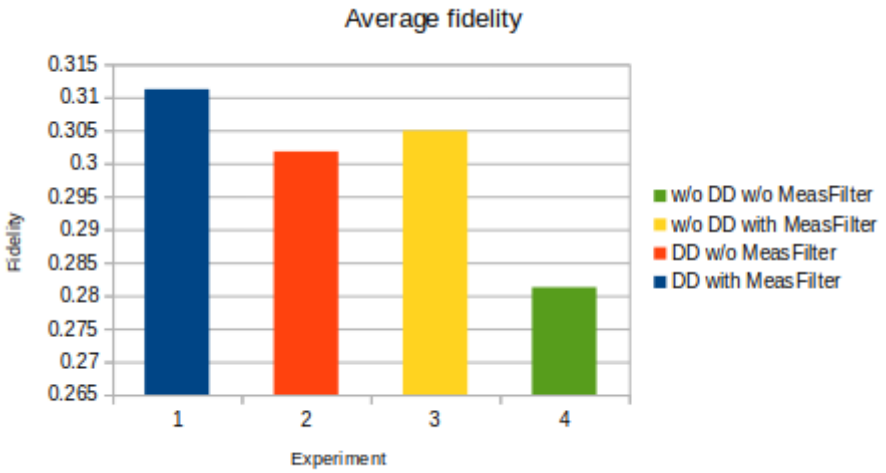
Amplitude = Amplitude * 1.8	 Classifier performance: 0.9848	 Classifier performance: 0.9879
Amplitude = Amplitude * 2.0	 Classifier performance: 0.9867	 Classifier performance: 0.9858
Amplitude = Amplitude * 2.2	 Classifier performance: 0.9852	 Classifier performance: 0.9846

Benchmark of a custom discriminator is showing higher performance than the default.

Default ibm discriminator	Custom discriminator **(improvements)**
Qubit 0 score: 57570.0 / 60000.0 = 0.9595 Qubit 1 score: 58104.0 / 60000.0 = 0.9684 Qubit 2 score: 58804.0 / 60000.0 = 0.9801 Qubit 3 score: 58558.0 / 60000.0 = 0.9760 Qubit 4 score: 58294.0 / 60000.0 = 0.9716 Qubit 5 score: 59243.0 / 60000.0 = 0.9874 Qubit 6 score: 58614.0 / 60000.0 = 0.9769	Qubit 0 score: 58581.0 / 60000.0 = 0.9764 (**+1.8%**) Qubit 1 score: 58936.0 / 60000.0 = 0.9823 (**+1.4%**) Qubit 2 score: 59412.0 / 60000.0 = 0.9902 (**+1%**) Qubit 3 score: 59298.0 / 60000.0 = 0.9883 (**+1.2%**) Qubit 4 score: 58496.0 / 60000.0 = 0.9749 (**+0.3%**) Qubit 5 score: 59386.0 / 60000.0 = 0.9898 (**+0.2%**) Qubit 6 score: 59246.0 / 60000.0 = 0.9874 (**+1.1%**)

To view the impact of custom discriminator we run RZX based circuit with 6 trotter steps, with and without dynamical decoupling Walsh pattern:





We can see an improvement of around 1-2%.

Measurements filter

Qiskit provides an easy way to perform measurement filtering.

```
from qiskit.ignis.mitigation.measurement import
                                (complete_meas_cal, CompleteMeasFitter)

meas_calibs, state_labels = complete_meas_cal(qubit_list=[1,3,5],
                                              qr=7, circlabel='mcal')
job_cal = execute(meas_calibs, backend=backend, shots=32000)

meas_fitter = CompleteMeasFitter(job_cal.result(), state_labels)
meas_filter = meas_fitter.filter

...

mitigated_results = meas_filter.apply(job.result())
```

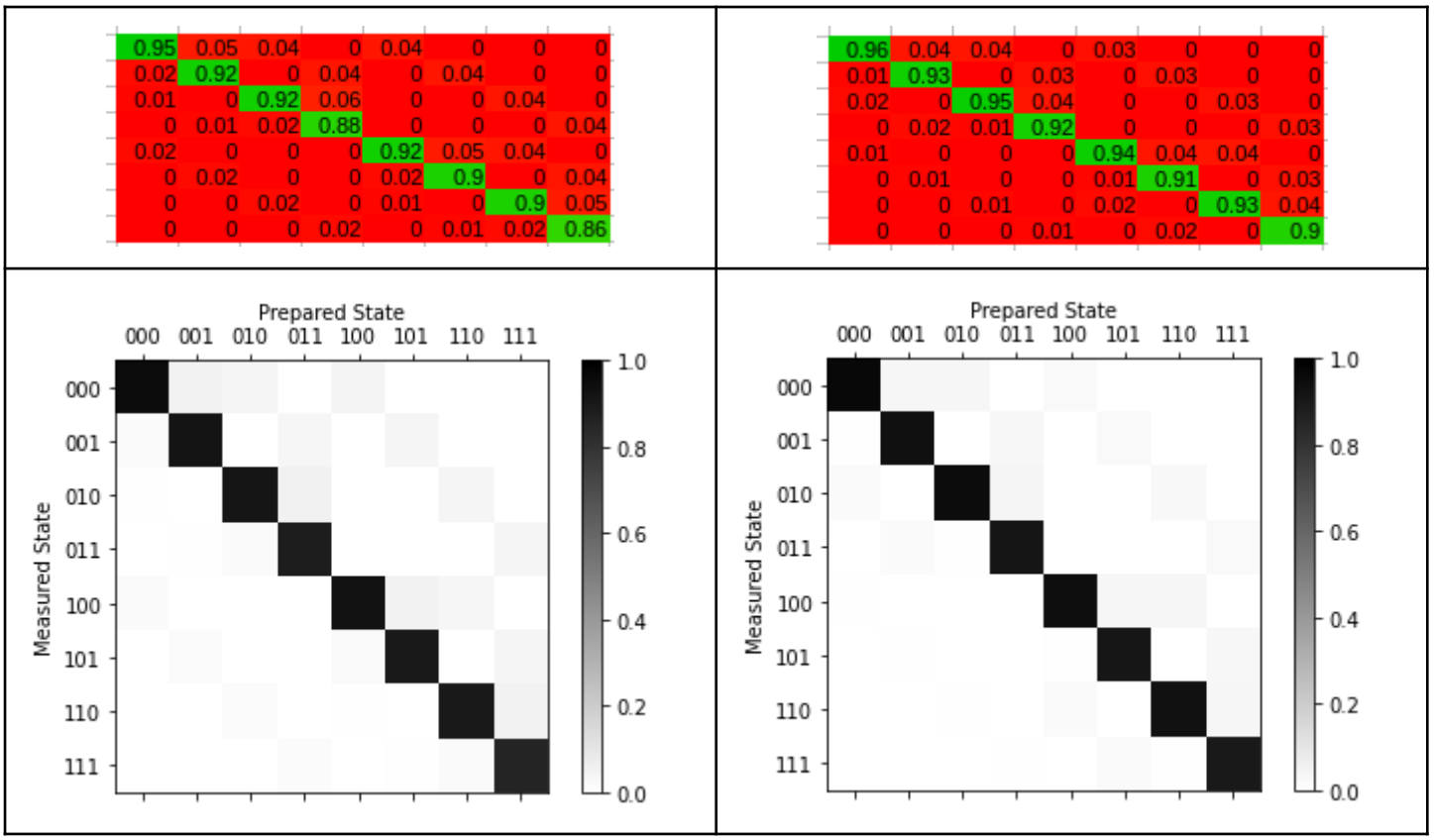
```
Import the standard qiskit functions


Create the measurement circuits for the full Hilbert
Space measurements.

Execute the circuits.

Create the fitter object which contains the error
mitigation parameters.


Execute other quantum circuits jobs

Apply the measurement filter on specific job results.
```

CompleteMeasFitter provides interfaces to display the fitter performance:

```
print("readout_fidelity:", meas_fitter.readout_fidelity())
print("cal_matrix:", meas_fitter.cal_matrix)
meas_fitter.plot_calibration()
meas_filter = meas_fitter.filter
```

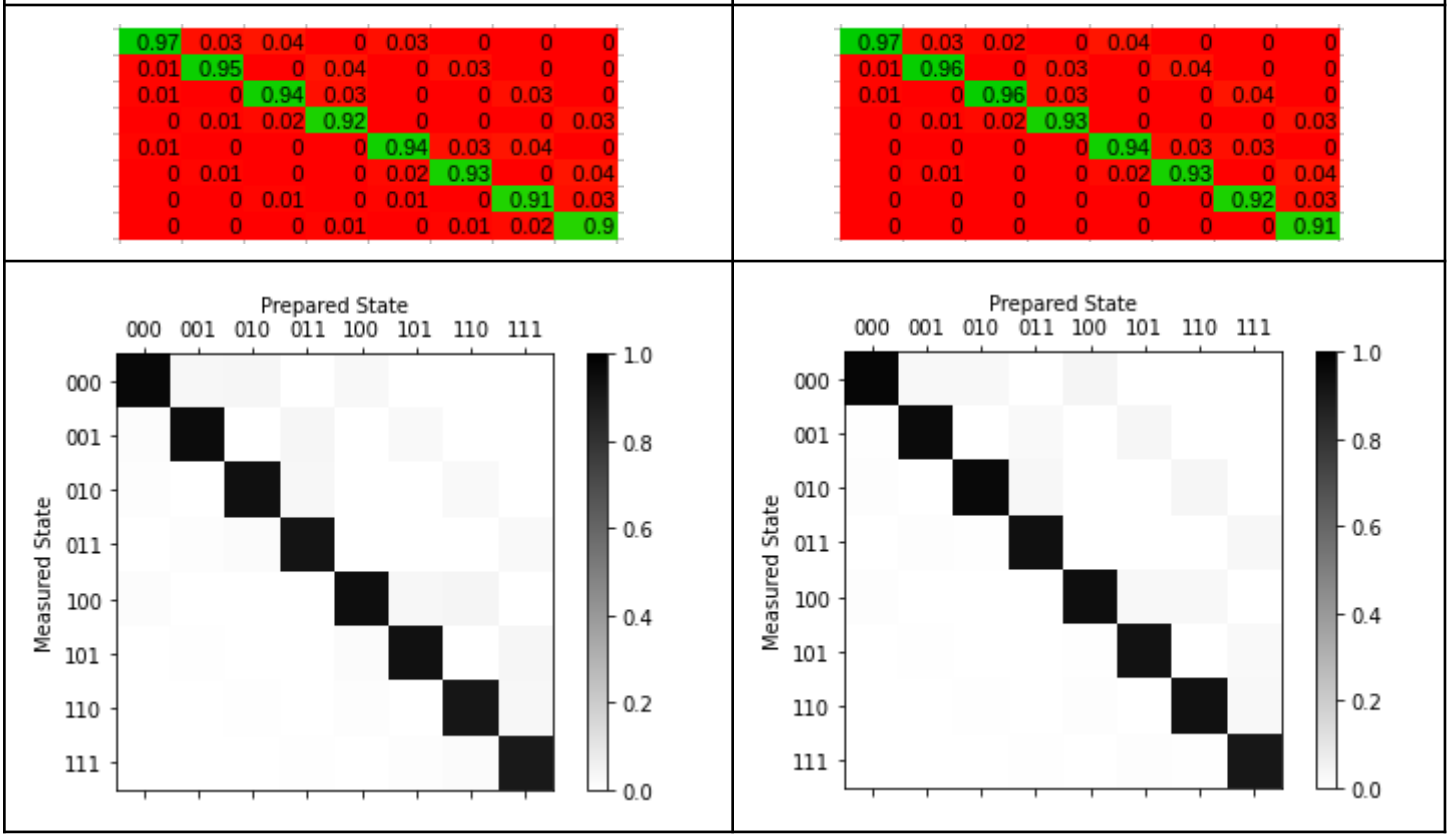
Results:

Default ibmq_jakarta performance	Custom k-Nearest Neighbor classifier
readout_fidelity: 0.9056	readout_fidelity: 0.9307



We additionally check the stability of the custom discriminator and measurement filter:

Custom k-Nearest Neighbor classifier - 1 day after calibration	Custom k-Nearest Neighbor classifier - 5 days after calibration
readout_fidelity: 0.9318	readout_fidelity: 0.9412



Further research possibilities

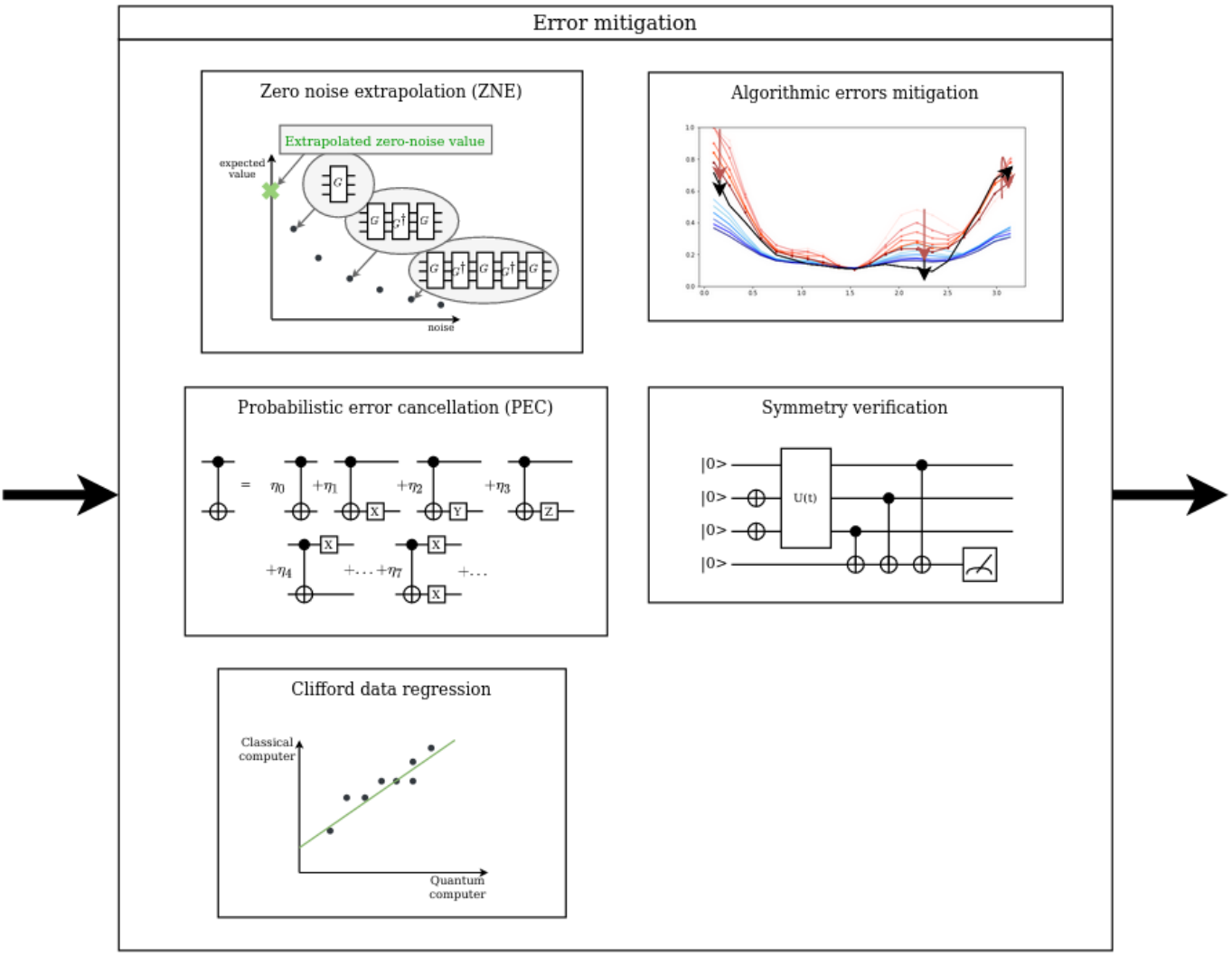
Impact of the measurement pulse amplitude modification shall be closer examined. Duration of the measurement pulse could be calibrated to further improve the discriminator performance.
Measurement samples gathered for long time periods could be used to create more robust discriminators by taking into consideration slow fluctuation of the quantum circuits between calibrations. Performance of custom discriminators for a longer time needs to be benchmarked.

Application to other problems

Custom discriminators based on machine learning could improve the readout performance (“Improving readout performance in superconducting qubits using machine learning | Application notes | Boulder Opal”). Gathering data for custom discriminators requires few quantum computer calls, so the cost of this technique is quite low. We were not able to find many research papers done for this technique so its performance needs to be closely examined for specific problems.

Number of circuits for the measurement filter grows exponentially with the qubit count ($2^{num\ of\ qubits}$), some solutions for this problem were proposed (Nation et al.). This technique provides a good fidelity improvement and low cost (for low qubit count) and could be applied easily to quantum circuits.

Error mitigation



Generic introduction

Error-mitigation techniques can enable access to accurate estimates of physical observables that are otherwise biased by noise (Mari et al.) or limitation of the circuits. Many techniques were design to mitigate different error and the most common are:

- Extrapolation methods: Zero Noise Estimation (ZNE) (Giurgica-Tiron et al.),
- Quasi-probability methods: Probabilistic Error Correction (PEC) (van den Berg et al.),
- Learning based error mitigation: Clifford Data Regression (Lowe et al.),
- Algorithm error mitigation
- Quantum subspace expansion,
- Quantum Error Correction (Otten and Gray),
- Symmetry verification.

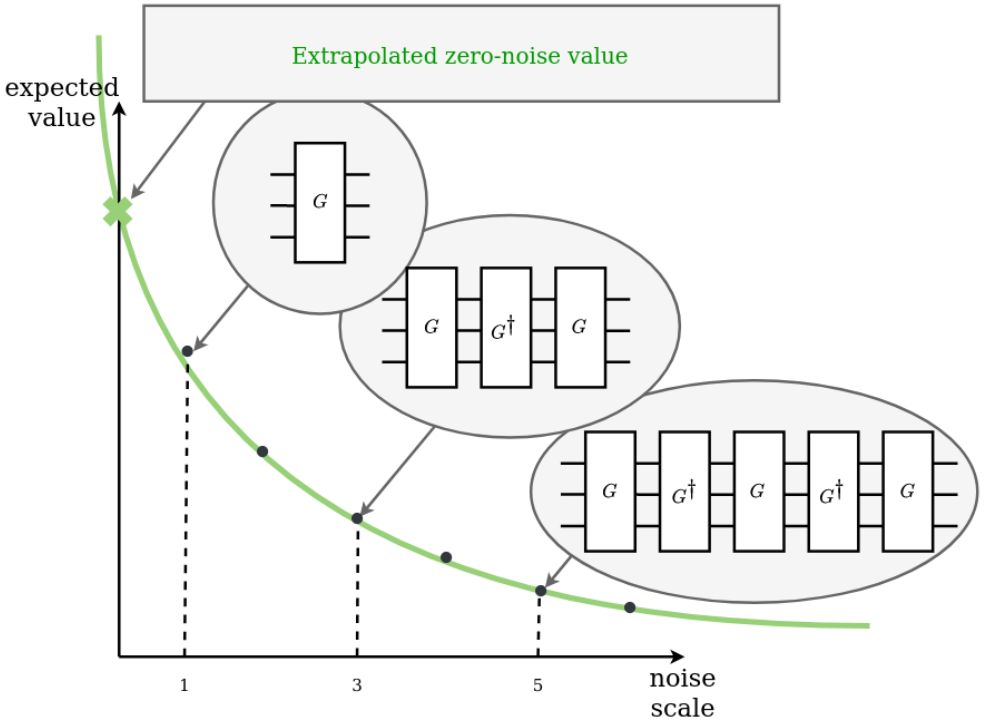
In this chapter we will briefly touch on the techniques we used and tried for the Ising simulation and skipped the unused techniques. Please check the recommended papers to get more information on error mitigation.

ZNE

The idea behind Zero Noise Estimation is that all our circuits have an inherent noise λ and it is possible to increase this noise by to $\lambda' = c\lambda$ where $c > 1$. By exploring the same circuit but with different noise levels it is possible to extrapolate the $\lambda = 0$ results.
The two most common techniques to scale the noise are:

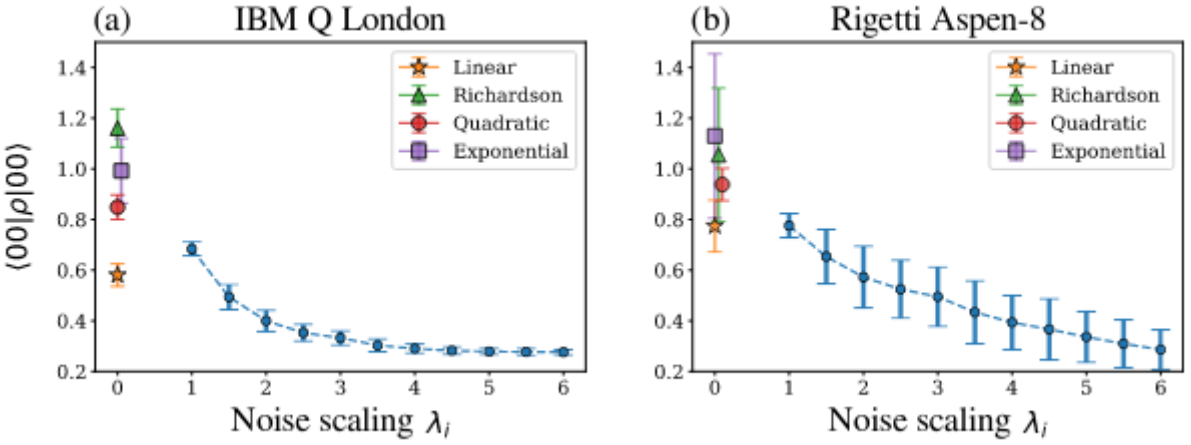
- Increasing the pulse duration for the gates which increase the overall time execution.
- Unitary folding - Replacing the subcircuit G with new circuits: $G\,G^{\dagger}G$.
- Replacing CNOT gates with 3xCNOT (Lowe et al.) (Endo et al.) (Unitary folding only for CNOT gates).

The unitary folding technique is one of the simplest and easiest to implement. The default circuits noise scale is 1, by adding unitary folding for the whole circuit we increase the noise scale to 3, adding additional unitary folding steps increases the noise scale to 5:



CNOT gate noise scaling is good only for circuits relaying on CNOT gates. Noise for 2 qubits gates is much higher than single qubit gates but some circuits require many single qubits operation which with CNOT scaling will have the noise profile biased.

The outcome measurements are often noise, simple extrapolation functions are performing better (Giurgica-Tiron et al.). Higher order polynomial functions often overfit to the data and simple linear and exponential functions perform well.
Different quantum computer architecture will require different extrapolation functions depending on their noise profile (LaRose et al.).



Mitigation of algorithm errors:

Trotter steps extrapolation

Usage of trotterization induces additional digitalization error in:

$$U(t) = (\prod_{l} e^{-iH_l t/n})^n + O(\frac{t^2}{n})$$

Let's set the $\epsilon_A = 1/n$ as the algorithmic error. We can increase the accuracy of the approximation by decreasing ϵ_A or equivalently having more Trotter steps. The number of Trotter steps needs to be properly chosen so the trade-off between algorithmic error and physical error is limited. In practice we need to apply quantum error mitigation techniques before mitigating the algorithmic error.
We can use the extrapolation methods to estimate the zero algorithmic error by choosing $n \to \infty$ (Endo et al.).

Trotter Hamiltonians permutation

Another technique used to reduce the algorithm errors is the randomization of the trotter operators (Childs et al.) (Jin and Li). First order approximation of Taylor expansion for the simple case of a hamiltonian: $H = H_1 + H_2$ is:

$$S_1(\lambda) = \exp(\lambda H_1)\exp(\lambda H_2) = I + \lambda(H_1 + H_2) + \frac{\lambda^2}{2}(H_1^2 + 2H_1H_2 + H_2^2) + O(\lambda^3),$$

where the ideal evolution is described by the equation:

$$V(\lambda) = \exp((H_1+H_2)\lambda) = I+\lambda(H_1+H_2)+\frac{\lambda^2}{2}(H_1^2+H_1H_2+H_2H_1+H_2^2)+O(\lambda^3).$$

The error between the ideal equation and the approximated is due to commuting relationship between H_1 and H_2.

$$\|V(\lambda) - S_1(\lambda)\| \leq \|[H_1, H_2]\| \frac{|\lambda|^2}{2} + O((\Lambda|\lambda|)^3),$$

For the example in which we revert the order of H_1 and H_2 the approximation equation changes to:

$$S_1^{\mathrm{rev}}(\lambda) := \exp(\lambda H_2)\exp(\lambda H_1).$$

Taking the mean value of $S_1(\lambda)$ and $S_1^{rev}(\lambda)$ will results in the perfect simulation (at least to the first order):

$$\left\| V(\lambda) - \frac{1}{2}(S_1(\lambda) + S_1^{\mathrm{rev}}(\lambda)) \right\| = O((\Lambda|\lambda|)^3).$$

By randomly choosing different permutations of the trotter steps hamiltonians we can reduce the noise due to commuting nature of the hamiltonians.

Papers for further reading

1. **Hybrid Quantum-Classical Algorithms and Quantum Error Mitigation**
   Suguru Endo, Zhenyu Cai , Simon C. Benjamin , and Xiao Yuan
   https://journals.jps.jp/doi/pdf/10.7566/JPSJ.90.032001

   We highly recommend a closer read of this publication, this is one of the best papers explaining most common error mitigation techniques and the effect of combining different techniques. The first chapters cover the Variational quantum optimization but the error mitigation content is written generically and can be easily understood with VQE knowledge.

   Hybrid Quantum-Classical Algorithms and Quantum Error Mitigation

   Suguru Endo[1*], Zhenyu Cai[2], Simon C. Benjamin[2], and Xiao Yuan[3,4†]

   [1]*NTT Secure Platform Laboratories, NTT Corporation, Musashino, Tokyo 180-8585, Japan*
   [2]*Department of Materials, University of Oxford, Parks Road, Oxford OX1 3PH, United Kingdom*
   [3]*Center on Frontiers of Computing Studies, Department of Computer Science, Peking University, Beijing 100871, China*
   [4]*Stanford Institute for Theoretical Physics, Stanford University, Stanford, CA 94305, U.S.A.*

   (Received July 30, 2020; accepted November 2, 2020; published online February 1, 2021)

   Quantum computers can exploit a Hilbert space whose dimension increases exponentially with the number of qubits. In experiment, quantum supremacy has recently been achieved by the Google team by using a noisy intermediate-scale quantum (NISQ) device with over 50 qubits. However, the question of what can be implemented on NISQ devices is still not fully explored, and discovering useful tasks for such devices is a topic of considerable interest. Hybrid quantum-classical algorithms are regarded as well-suited for execution on NISQ devices by combining quantum computers with classical computers, and are expected to be the first useful applications for quantum computing. Meanwhile, mitigation of errors on quantum processors is also crucial to obtain reliable results. In this article, we review the basic results for hybrid quantum-classical algorithms and quantum error mitigation techniques. Since quantum computing with NISQ devices is an actively developing field, we expect this review to be a useful basis for future studies.

   CONTENTS

2. **Extending quantum probabilistic error cancellation by noise scaling**
   Andrea Mari Nathan Shammah and William J. Zeng
   https://arxiv.org/pdf/2108.02237.pdf

   To further understand Extrapolation methods combined with probability methods this paper is a recommended read. It proposes a new algorithm combining zero noise extrapolation and probabilistic error cancellation called NEPEC, but additionally provides a very good explanation of other algorithms.

Extending quantum probabilistic error cancellation by noise scaling

Andrea Mari,[1] Nathan Shammah,[1] and William J. Zeng[1,2]
[1] *Unitary Fund*
[2] *Goldman, Sachs & Co, New York, NY*

We propose a general framework for quantum error mitigation that combines and generalizes two techniques: probabilistic error cancellation (PEC) and zero-noise extrapolation (ZNE). Similarly to PEC, the proposed method represents ideal operations as linear combinations of noisy operations that are implementable on hardware. However, instead of assuming a fixed level of hardware noise, we extend the set of implementable operations by noise scaling. By construction, this method encompasses both PEC and ZNE as particular cases and allows us to investigate a larger set of hybrid techniques. For example, gate extrapolation can be used to implement PEC without requiring knowledge of the device's noise model, e.g., avoiding gate set tomography. Alternatively, probabilistic error *reduction* can be used to estimate expectation values at intermediate *virtual* noise strengths (below the hardware level), obtaining partially mitigated results at a lower sampling cost. Moreover, multiple results obtained with different noise reduction factors can be further post-processed with ZNE to better approximate the zero-noise limit.

CONTENTS

search problem for near-term quantum computing. Standard quantum error correction codes [1–5] are theoretically known, but can require significant resources (e.g. qubits, gates) that are unavailable on near-term quantum devices [6]. On the one hand, the most direct way to reduce physical errors is to improve the existing hardware, e.g., by realizing more stable qubits and less noisy operations. On the other hand, given the existing noisy hardware, large improvements can be achieved at the "software level" by using several techniques which have been recently called error mitigation methods [7–9] and which are the focus of this work.

One of such methods is zero-noise extrapolation (ZNE) [7, 10, 11], where a quantum observable is measured at different noise levels (by artificially increasing the hardware noise) and extrapolated to the zero-noise limit. Another promising method is probabilistic error cancellation (PEC) [7, 8, 12], where ideal circuits are approximated

3. Unifying and benchmarking state-of-the-art quantum error mitigation techniques

Daniel Bultrini, Max Hunter Gordon, Piotr Czarnik, Andrew Arrasmith, Patrick J. Coles, and Lukasz Cincio
https://arxiv.org/pdf/2107.13470.pdf

Error mitigation techniques are based on the trade-off number of the circuits vs performance. The paper presents a benchmark of most common techniques relying on extrapolation, probability and learning based mitigation. From this paper we can see the trade between number of circuits run and the accuracy of models, allowing us to select the best technique based on hardware availability.

Unifying and benchmarking state-of-the-art quantum error mitigation techniques

Daniel Bultrini,[1,2,*] Max Hunter Gordon,[3,*] Piotr Czarnik,[1]
Andrew Arrasmith,[1,4] Patrick J. Coles,[1,4] and Lukasz Cincio[1,4]
[1] *Theoretical Division, Los Alamos National Laboratory, Los Alamos, NM 87545, USA*
[2] *Theoretische Chemie, Physikalisch-Chemisches Institut, Universität Heidelberg, INF 229, D-69120 Heidelberg, Germany*
[3] *Instituto de Física Teórica, UAM/CSIC, Universidad Autónoma de Madrid, Madrid, Spain*
[4] *Quantum Science Center, Oak Ridge, TN 37931, USA*

Error mitigation is an essential component of achieving practical quantum advantage in the near term, and a number of different approaches have been proposed. In this work, we recognize that many state-of-the-art error mitigation methods share a common feature: they are data-driven, employing classical data obtained from runs of different quantum circuits. For example, Zero-noise extrapolation (ZNE) uses variable noise data and Clifford-data regression (CDR) uses data from near-Clifford circuits. We show that Virtual Distillation (VD) can be viewed in a similar manner by considering classical data produced from different numbers of state preparations. Observing this fact allows us to unify these three methods under a general data-driven error mitigation framework that we call UNIfied Technique for Error mitigation with Data (UNITED). In certain situations, we find that our UNITED method can outperform the individual methods (i.e., the whole is better than the individual parts). Specifically, we employ a realistic noise model obtained from a trapped ion quantum computer to benchmark UNITED, as well as state-of-the-art methods, for problems with various numbers of qubits, circuit depths and total numbers of shots. We find that different techniques are optimal for different shot budgets. Namely, ZNE is the best performer for small shot budgets (10^5), while Clifford-based approaches are optimal for larger shot budgets ($10^6 - 10^8$), and for our largest considered shot budget (10^{10}), UNITED gives the most accurate correction. Hence, our work represents a benchmarking of current error mitigation methods, and provides a guide for the regimes when certain methods are most useful.

I. INTRODUCTION

As new generations of quantum computers become larger and less noisy, the practical application of quantum algorithms is just around the corner. It is likely that any quantum algorithm that can achieve an advantage over its classical counterpart will have to contend with

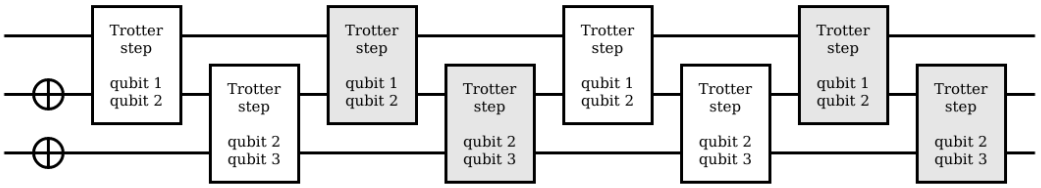
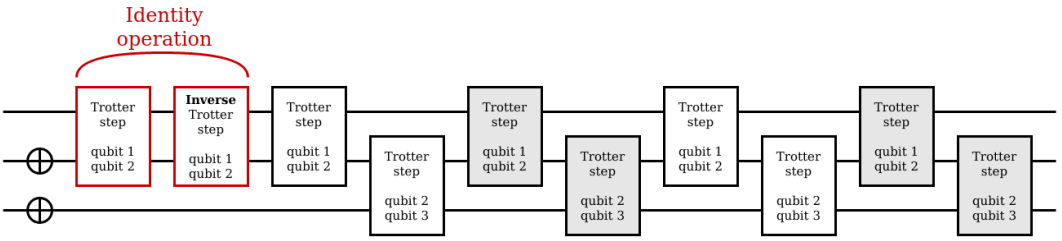
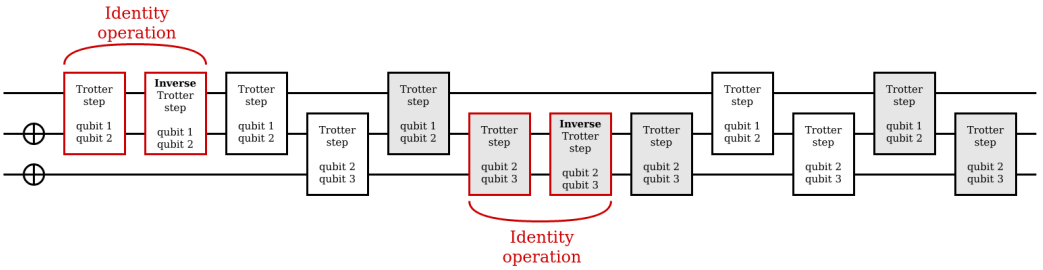
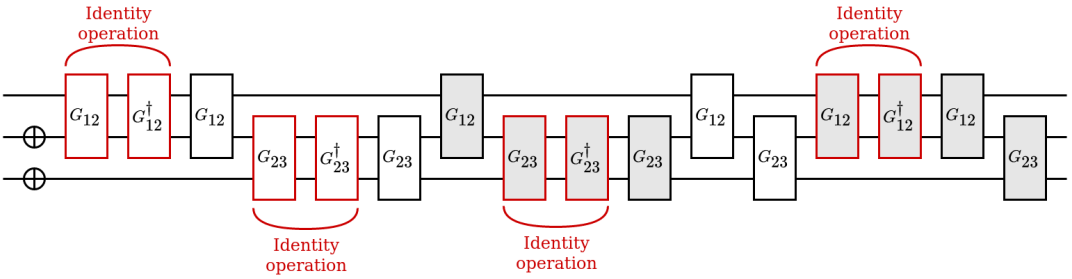
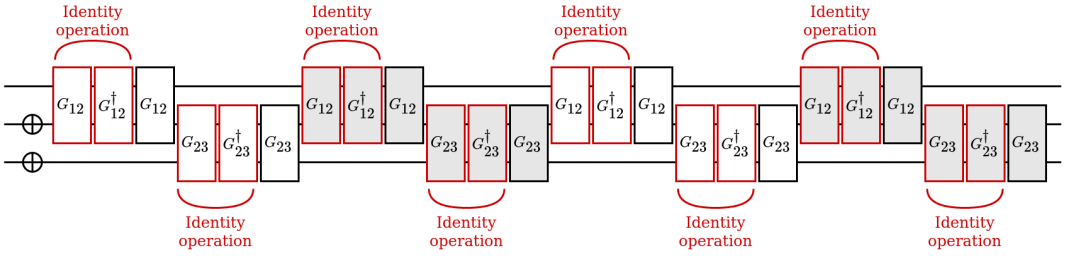
for circuits involving many gates, the lowest error points available are often too noisy for such fits to be helpful.

Alternatively, classically simulable circuits can be used to inform the experimenter about noise in a quantum device [6–10]. The Clifford data regression (CDR) method [6] makes use of the Gottesman-Knill theorem [11] which guarantees that quantum circuits con-

Application to Ising simulation.

Zero-noise estimation

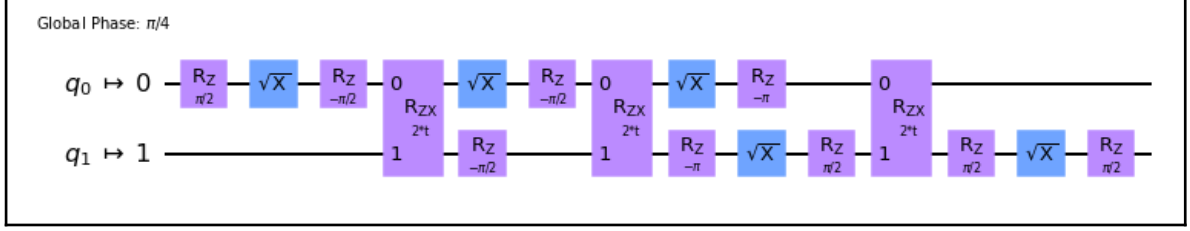
We did not decide to investigate the pulse extension due to the need for calibration.
Every trotterization step can be considered a seperate circuit which we can fold. This allows us to generate circuits noise scale in between 1 and 3 giving more interpolation steps.

Default circuit Noise scale = 1	
Noise scale = 1.25 $\lambda = \frac{10\ trotter\ steps}{8\ default\ trotter\ steps}$	
Noise scale = 1.5 $\lambda = \frac{12\ trotter\ steps}{8\ default\ trotter\ steps}$	
Noise scale = 2.0 $\lambda = \frac{16\ trotter\ steps}{8\ default\ trotter\ steps}$	
Noise scale = 3.0 $\lambda = \frac{24\ trotter\ steps}{8\ default\ trotter\ steps}$	

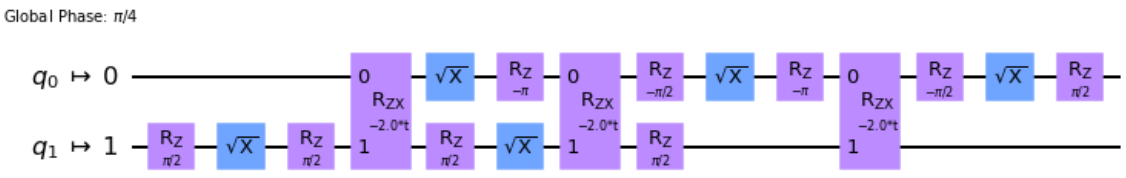
We need to construct the inverse circuit for our ising step. This can be done with the `.inverse()` function in qiskit.
Ex:

```
qc = transpile(Ising2Q_qc.inverse(), backend, optimization_level=2,
               basis_gates=['x', 'rx', 'rz', 'sx', 'cx', 'h'])
qc.draw(idle_wires=False)
```

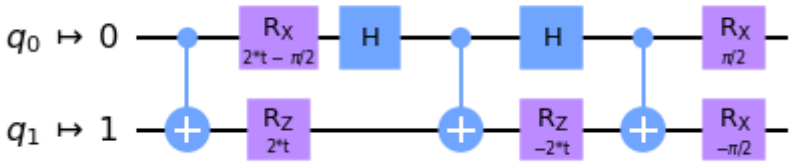
Trotter step for Rzx circuit



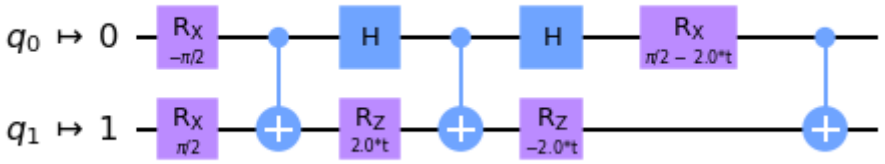
Inverse of Trotter step for Rzx circuit



Trotter step for CNOT circuit

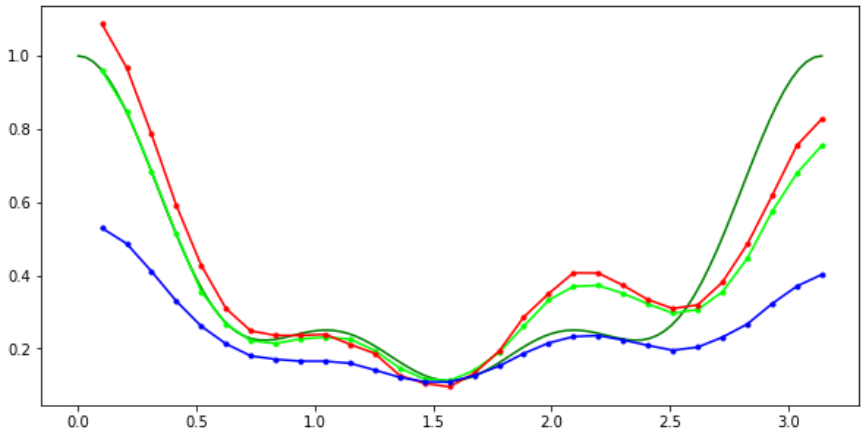


Inverse of Trotter step for CNOT circuit



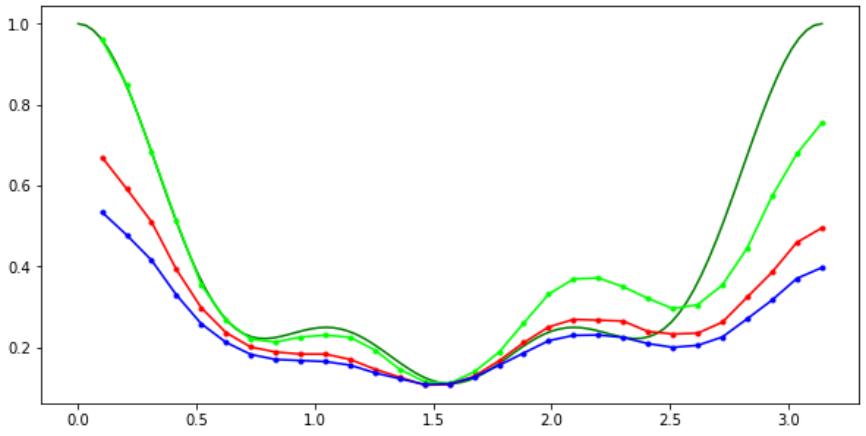
We benchmarked the impact of different extrapolation methods for the full ising simulation. This was run on a simulator and we observed the |*110*> state expected value.

Extrapolation function $y = A\,e^{-x} + B$

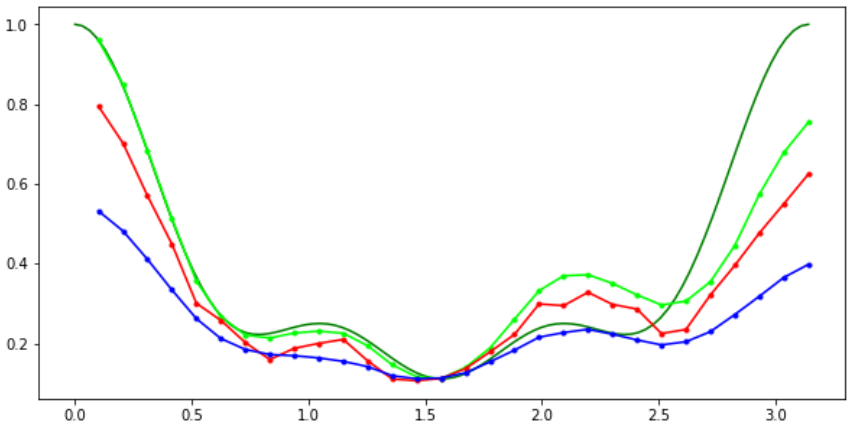


Noiseless Ising simulation
Noiseless Ising simulation with 7 Trotterization steps
ZNE mitigated 7 Trotterization step simulation
7 Trotterization steps simulation without error mitigation

Extrapolation function $y = Ax + B$



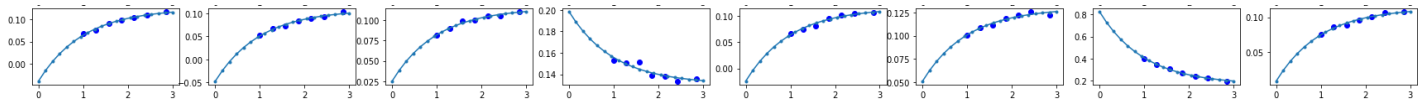
Extrapolation function $y = Ax^2 + Bx + C$



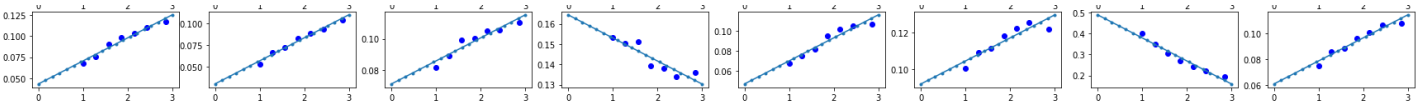
As we can see the best fit is reached using exponential function extrapolation, sometimes overestimating the final state.

Examples of the fitting function performance:

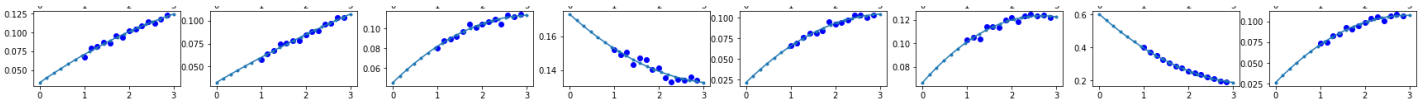
$y = A\,e^{-x} + B$:



$y = Ax + B$



$y = Ax^2 + Bx + C$

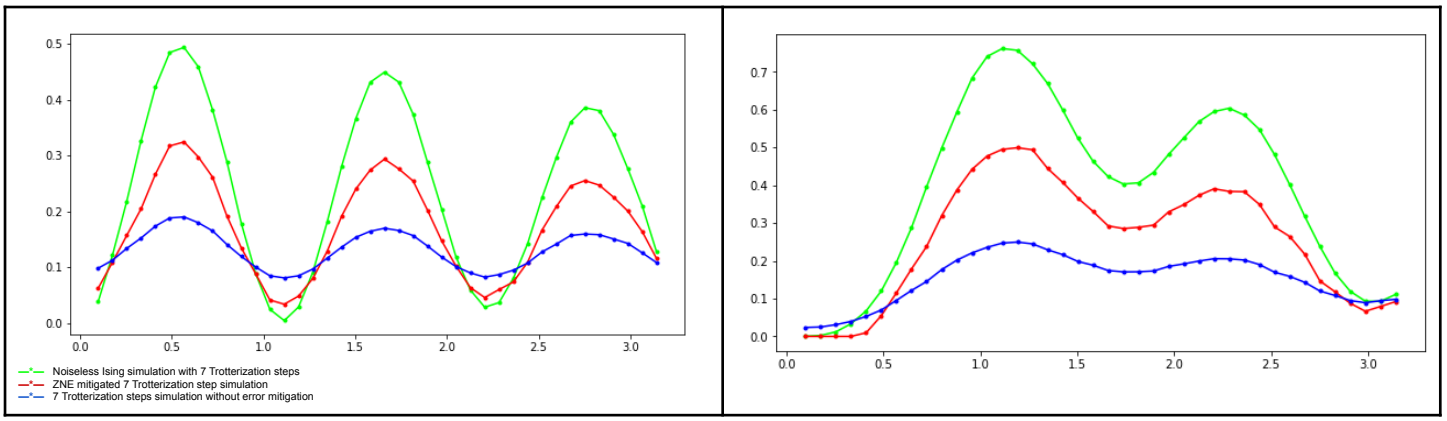


|000> |001> |010> |011> |100> |101> |110> |111>

We did not use function $y = Ae^{-Bx} + C$ due to problems with non converging to a solution using scipy `curve_fit()` function.

Extrapolation often results in state counts below 0, so we decided to saturate the minimal value at 0.
We run different starting setups for the Ising model to see if this decision impacts the performance, but we did not observe any errors in the simulation outcome. Adding “0” minimal thresholds improves the performance of the extrapolation and does not affect the shape of the estimated simulation.

Initial state |101> and measured expectation values of |110>	Initial state |011> and measured expectation values of |110>



Probabilistic Error Cancellation

We were not able to benchmark the PEC due to low availability of the ibmq_jakarta and the number of circuits required to mitigate the errors. But this technique could be easily applied to further reduce the error rates of Ising simulation.

Combining PEC and Extrapolation techniques improves the overall noise rejection but requires many additional circuits (Bultrini et al.):

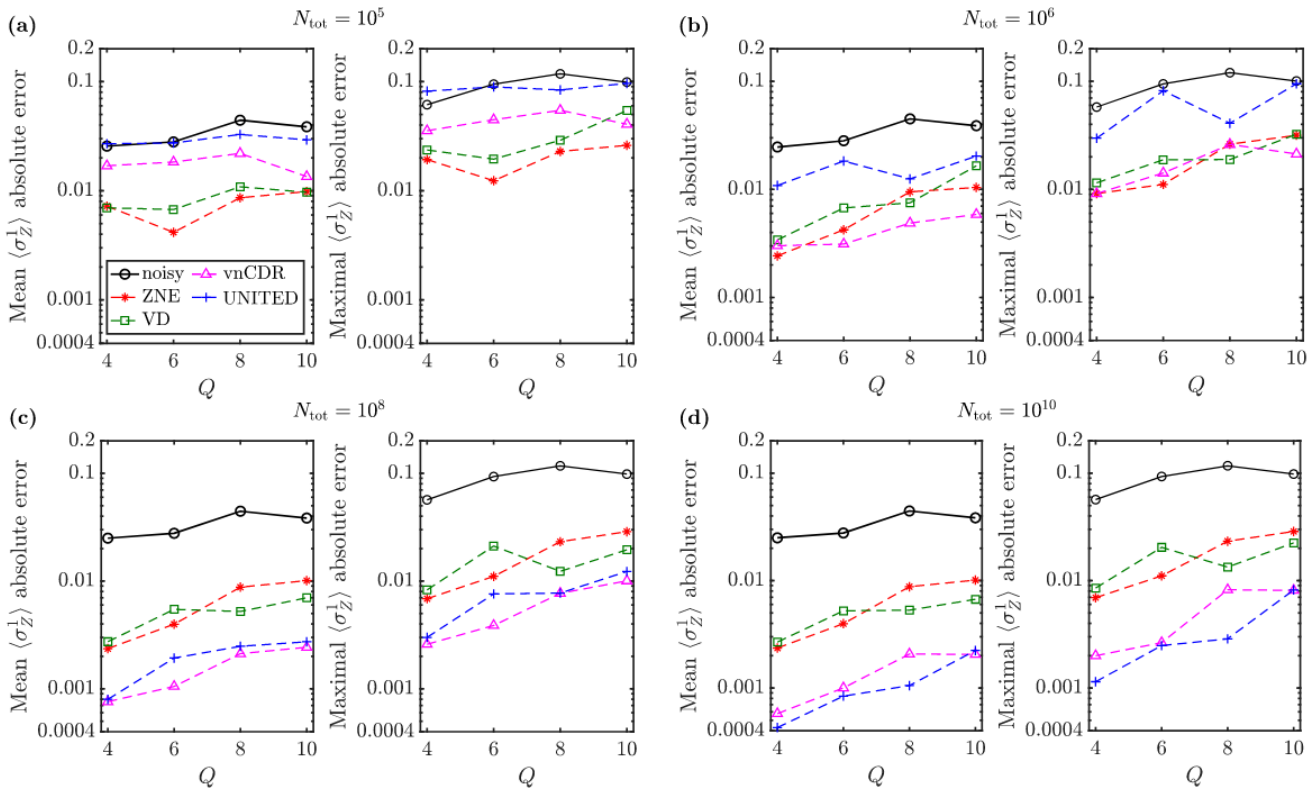


FIG. 2. **Comparing error mitigation methods for $L = Q$.** Mean and maximal absolute errors of the expectation value of σ_Z measured at the first qubit of 30 instances of random quantum circuits for noisy estimates and estimates obtained with error mitigation methods. In (a), the results for total number of shots per mitigated (noisy) expectation value $N_{\rm tot} = 10^5$. In (b), $N_{\rm tot} = 10^6$. In (c), $N_{\rm tot} = 10^8$. In (d), $N_{\rm tot} = 10^{10}$. For $N_{\rm tot} = 10^5$, we find that ZNE gives the best results. For $N_{\rm tot} = 10^6$, 10^8 vnCDR gives the best results while for $N_{\rm tot} = 10^{10}$ UNITED gives the best results.

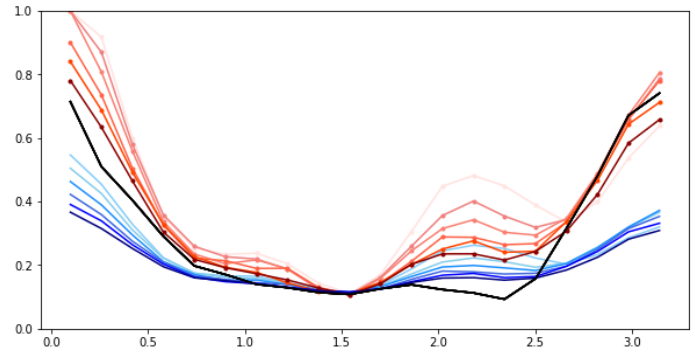
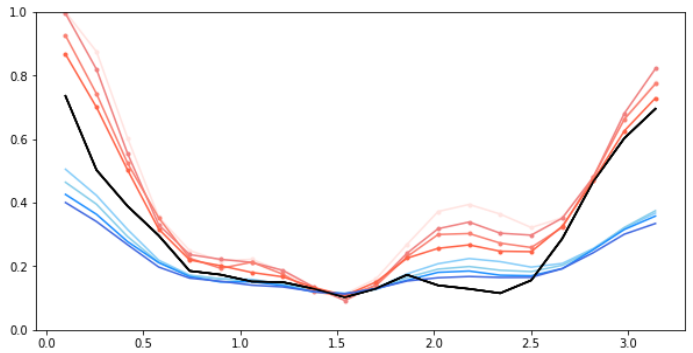
For the ising simulation we went with the minimal number of circuits required and only applied ZNE. Further benchmarking could be done in the future. But as seen in the graphic above 10^8 circuits are required to see benefits from more advanced techniques.

Clifford data regression

Ising simulation for 3 qubits can be easily simulated on a classical computer so we decided not to investigate Clifford data regression techniques because we could easily achieve perfect simulation.

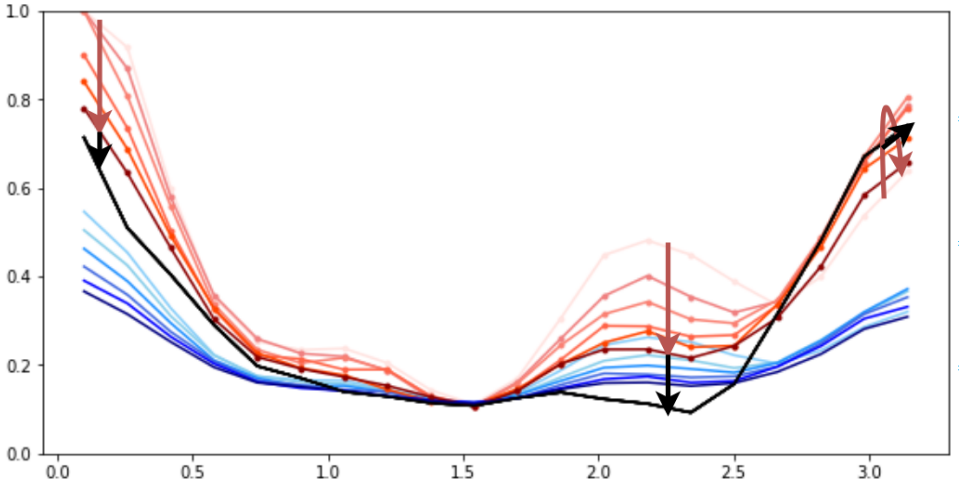
Algorithm errors mitigation.

This technique is suitable for ising simulation based on trotterization. Algorithm mitigation techniques require low noise in the circuit so for Ising simulation it shall be applied after error mitigation techniques for example ZNE.

6 - 12 Trotter steps	7 - 11 Trotter steps
	

Algorithm mitigation results (based on measurement with ZNE)
ZNE mitigated (lighter color fewer Trotter steps, darker color more steps)
Simulation with measurement filter (lighter color fewer Trotter steps, darker color more steps)

We can see that the algorithm mitigation overestimates the impact of the digitalization error driving the estimated value below the noiseless values:



The effect of the simulation noise is noticeable in the high trotter steps simulation. The proper choice of the benchmarked trotter steps can reduce the overestimation. We feel adjusting trotter steps will be too specific to our problem and it will not generalize easily to other ising simulation models, so we did not apply this technique for the final simulation.

Due to high optimization applied to the trotter step (we optimized the $XX(\delta)$, $YY(\delta)$ and $ZZ(\delta)$ gates layout), we are not able to easily perform the Hamiltonian permutation. We simulated the impact of different permutations of the 6 CNOT circuits but the results did not show any noticeable improvement.

Experiments

Setup

Error mitigation techniques are used as the last step in the quantum circuit optimization pipeline. ZNE works the best when the noise in the circuit is low enough and additional noise can be added before we are at the noise level of the quantum system.
We apply the following techniques to reduce the circuit noise:

- Trotter steps set from 7 to 12.
- Circuit is constructed using RZX operation.
- X and √X gates are calibrated using qiskit Fine<X,SX>AmplitudeCal() and Fine<X,SX>DragCal() functions, with default qiskit settings and extended number of gates in series.
- Dynamic decoupling sequence was added.
- Custom discriminator based on k-Nearest Neighbors classifier with $k = 100$.
- Measurement filter was used.

ZNE

To generate noise scaling we created a custom function `generate_zne_pattern()` to generate the information which Trotterization steps will be extended. We decided to follow the technique from (Giurgica-Tiron et al.) to use random gate folding.

```
def generate_zne_pattern(trotter_steps, noise):
    if noise > 3.01:
        raise NotImplementedError("noise higher than 3.01 is not yet implemented")
```

Right now only noise scaling up to 3 is only supported due to already high noise in the Ising simulation circuit. We are able to generate many intermediate steps from 1 to 3 noise scales. This is sufficient to have enough samples for the extrapolation.

```
    circuits_in_11_patern = 2 # zne without noise has 2 circuits for every
                              # trotter steps (G01, G12)
    circuits_in_12_patern = 4 # zne with noise only applied to one ising circuit
                              # has 4 circuits (G01, Gd01, G01,G12)
    circuits_in_22_patern = 6 # zne with noise on both ising circuits has 6 circuits
                              #  (G01, Gd01, G01, G12, Gd12, G12)
```

Number of circuits in one trotter steps if we don't apply any noise (pattern 11, noise for single qubit pair pattern 12 and noise for both qubit pairs - pattern 22)

```
    for num_of_22_patterns in range(trotter_steps, -1, -1):
        zne_score = num_of_22_patterns*circuits_in_22_patern + \
                    (trotter_steps-num_of_22_patterns)*circuits_in_11_patern
        if noise > zne_score / (trotter_steps*circuits_in_11_patern):
            break
```

Look for the minimal number of 22 patterns which are needed to generate the expected noise scaling.

```
    for num_of_12_patterns in range(trotter_steps, -1, -1):
        zne_score = num_of_12_patterns*circuits_in_12_patern + \
                    (trotter_steps-num_of_12_patterns-num_of_22_patterns)*\
                      circuits_in_11_patern + \
                    num_of_22_patterns*circuits_in_22_patern
        if noise > zne_score / (trotter_steps*circuits_in_11_patern):
            break
```

After calculating the number of 22 patterns, check how many 12 patterns can we apply.

```
    zne_pattern = [[1,1]]*trotter_steps
    steps = list(range(0, trotter_steps))
```

zne_pattern array will have the final ZNE sequence.

```
    for _ in range(0, num_of_22_patterns):
        rnd_step = random.randint(0, len(steps)-1)
        zne_pattern[steps[rnd_step]] = [2,2]
        steps.pop(rnd_step)
```

Assign 22 and 12 patterns to random trotter steps.

```
    for _ in range(0, num_of_12_patterns):
        rnd_step = random.randint(0, len(steps)-1)
        if random.random() > 0.5:
            zne_pattern[steps[rnd_step]] = [1,2]
        else:
            zne_pattern[steps[rnd_step]] = [2,1]
            steps.pop(rnd_step)
```

```
    return zne_pattern, num_of_12_patterns, num_of_22_patterns
```

Return the ZNE pattern.

```
zne_pattern_, _12, _22 = generate_zne_pattern(trotter_steps=7, noise=1.80)
```

For this example we will get pattern:
[[1, 1], [2, 1], [2, 2], [2, 2], [1, 1], [1, 1], [1, 1]]
That is ZNE pattern for q1q2 pair at trotter step 2, ZNE pattern for both qubits in Trotter steps 3 & 4.

The ZNE pattern can be supplied as an argument for the `ising_sim()`:

```
def ising_sim(trotter_steps, target_time, optimization_level, circ_type,
                      calibrated=False, zne_pattern=[]):
    ...

    Ising2Q_qr = QuantumRegister(2)
    Ising2Q_qc = QuantumCircuit(Ising2Q_qr, name='Ising_2Qubits')

    if circ_type == 'cnot':
        ...
    if circ_type == 'rzx':
        ...
    Ising2Q = Ising2Q_qc.to_instruction()
```

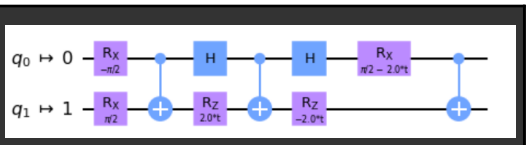
```
    if len(zne_pattern) != 0:
        Ising2Q_inv_qr = QuantumRegister(2)
        Ising2Q_inv_qc = QuantumCircuit(Ising2Q_qr, name='Ising_inv_2Qubits')
```

If the ZNE pattern was provided, create the Inverse circuits.

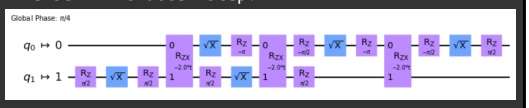
```
        if circ_type == 'cnot':
            t = Parameter('t')
            Ising2Q_inv_qc.rx(-np.pi/2,0)
```

Inverse CNOT trotter step:

```
            Ising2Q_inv_qc.rx(np.pi/2,1)
            Ising2Q_inv_qc.cnot(0,1)
            Ising2Q_inv_qc.h(0)
            Ising2Q_inv_qc.rz(2 * t, 1)
            Ising2Q_inv_qc.cnot(0,1)
            Ising2Q_inv_qc.h(0)
            Ising2Q_inv_qc.rz(-2 * t, 1)
            Ising2Q_inv_qc.rx(-2 * t + np.pi/2, 0)
            Ising2Q_inv_qc.cnot(0,1)

        if circ_type == 'rzx':
            rzx_inv_gate = Gate('rzx_inv_', 2, [])

            Ising2Q_inv_qc.h(1)
            Ising2Q_inv_qc.append(rzx_inv_gate, [0, 1])
            Ising2Q_inv_qc.sx(0)
            Ising2Q_inv_qc.rz(-np.pi, 0)
            Ising2Q_inv_qc.rz(np.pi/2, 1)
            Ising2Q_inv_qc.sx(1)
            Ising2Q_inv_qc.append(rzx_inv_gate, [0, 1])
            Ising2Q_inv_qc.rz(-np.pi/2, 0)
            Ising2Q_inv_qc.sx(0)
            Ising2Q_inv_qc.rz(-np.pi, 0)
            Ising2Q_inv_qc.rz(np.pi/2, 1)
            Ising2Q_inv_qc.append(rzx_inv_gate, [0, 1])
            Ising2Q_inv_qc.rz(-np.pi/2, 0)
            Ising2Q_inv_qc.sx(0)
            Ising2Q_inv_qc.rz(np.pi/2, 0)

        Ising2Q_inv = Ising2Q_inv_qc.to_instruction()

    qc = QuantumCircuit(7)
    qc.x([q1,q2])  # DO NOT MODIFY (|q_5,q_3,q_1> = |110>)
    for step in range(trotter_steps):
        qc.append(Ising2Q, [q0, q1])
        if len(zne_pattern) > step:
            for _ in range(2, int(zne_pattern[step][0])+1):
                qc.append(Ising2Q_inv, [q0, q1])
                qc.append(Ising2Q, [q0, q1])


        qc.append(Ising2Q, [q1, q2])
        if len(zne_pattern) > step:
            for _ in range(2, int(zne_pattern[step][1])+1):
                qc.append(Ising2Q_inv, [q1, q2])
                qc.append(Ising2Q, [q1, q2])

    ...
```



Inverse Rzx trotter step:



Create the full ising circuit.
Apply X gate to get the |110> state.

For every trotter step.
Add the 2 Qubit Ising sim for the first qubits.

If the ZNE pattern for this trotter step is set.
Apply the inverse and standard 2 Qubit Ising circuit to increase the noise. The number of repetition of the inverse and standard circuits are defined in the zne_pattern array.

Follow the same pattern for scale noise other qubits Ising circuit.

To execute the Ising simulation with different values of scale noise we use the following code:

```
circuits = []
schedules = []
jobs = []

target_time = np.pi
calibrated_ = True
meas_calibrated_ = True
dd_pattern_ = "Walsh_Paley5"

experiments = \
         [{'steps': 7, "circ_type": "rzx", "calibrated": calibrated_,
           "calibrated_meas": meas_calibrated_, "dd": dd_pattern_,
           "zne_pattern": generate_zne_pattern(trotter_steps=7, noise=1.00)[0],
           "comment": ""}] +\
         [{'steps': 7, "circ_type": "rzx", "calibrated": calibrated_,
           "calibrated_meas": meas_calibrated_, "dd": dd_pattern_,
           "zne_pattern": generate_zne_pattern(trotter_steps=7, noise=1.15)[0],
           "comment": ""}] +\
...
         [{'steps': 7, "circ_type": "rzx", "calibrated": calibrated_,
           "calibrated_meas": meas_calibrated_, "dd": dd_pattern_,
           "zne_pattern": generate_zne_pattern(trotter_steps=7, noise=1.86)[0],
           "comment": ""}] +\
...
         [{'steps': 7, "circ_type": "rzx", "calibrated": calibrated_,
           "calibrated_meas": meas_calibrated_, "dd": dd_pattern_,
           "zne_pattern": generate_zne_pattern(trotter_steps=7, noise=2.43)[0],
           "comment": ""}] +\
...
```

Global job inputs and results for further analysis.

Defines for the setup.

Experiments with different noise scale circuits created using generate_zne_pattern().

```
random.shuffle(experiments)
for _ in range(len(experiments)):
    jobs.append(None)
    circuits.append(None)
    schedules.append(None)

def execute_circuit(idx, target_time):
    exp = experiments[idx]
    qc, inst_map_ = ising_sim(trotter_steps=exp['steps'], target_time=target_time,
                              optimization_level=0, circ_type=exp["circ_type"],
                              calibrated=exp['calibrated'],
                              zne_pattern=exp["zne_pattern"])
    circuits[idx] = qc

    schedule_qc = schedule(qc, backend, inst_map=inst_map_, method='alap')
    if exp['calibrated_meas']:
        schedule_qc = add_measurement_cal(schedule_qc,
                                          discriminators=discriminators)
    if exp['dd'] is not None:
        schedule_qc = add_dd(schedule_qc, pattern=exp['dd'])

    job = execute(schedule_qc, backend, meas_level=2, shots=32000,
                  optimization_level=0)
    job.wait_for_final_state()
    job_monitor(job)
    jobs[idx] = job

for idx in range(0, len(experiments)):
    execute_circuit(idx, target_time)
```

Randomize the experiments so the execution order does not influence the results.

Execute circuit.

Generate the ising circuit based on the input parameters.

Build pulse schedules.
If calibration measurement is done apply the custom measurement pulse and discriminator.

If a dynamic decoupling pattern is needed add it using our custom function.

Execute and wait for the result.

The jobs object from every experiment will be stored in a global jobs array.

Run the executor for all experiments.

The ZNE data is extrapolated and the fidelity is calculated using the following code:

```
from scipy.optimize import curve_fit


measurement_filter = True

zne_fit_func = lambda x, A, B: A*np.exp(-x)+B


num_tomo_circ = 3**3 # state tomography requires 3^qubits circuits


zne_result = copy.deepcopy(jobs.result())
zne_result_dict = zne_result.to_dict()
zne_result_dict['results'] = zne_result.to_dict()['results'][0:num_tomo_circ]
zne_result = zne_result.from_dict(zne_result_dict)

circuit_without_noise_idx = 0



unmittigated_result = copy.deepcopy(jobs.result())
unmittigated_result_dict = unmittigated_result.to_dict()
unmittigated_result_dict['results'] = \
             unmittigated_result.to_dict()['results'][0:num_tomo_circ]
unmittigated_result = unmittigated_result.from_dict(unmittigated_result_dict)

if measurement_filter:
    mitigated_results = meas_filter.apply(jobs.result())

    meas_filter_result = copy.deepcopy(mitigated_results)
    meas_filter_result_dict = meas_filter_result.to_dict()
    meas_filter_result_dict['results'] = \
            meas_filter_result.to_dict()['results'][0:num_tomo_circ]
    meas_filter_result = meas_filter_result.from_dict(meas_filter_result_dict)

for tomo_circ_idx in range(0, num_tomo_circ):
    exp_results = {}
    for qubits_state in jobs.result().get_counts()[-1]:
        x = []
        y = []

        exp_results[qubits_state] = 0
        shots = jobs.result().results[-1].shots # assuming that all circuits were run
with the same shots number

        for idx, exp in enumerate(experiments):
            zne_pattern = list(np.array(exp["zne_pattern"]).flat)
```

curve_fit returns the best parameters values for the extrapolation function based on the input points (ZNE samples).

Do we want to apply measurement filter?

Exponential function will be used for the extrapolation.

Number of tomography circuits 3^(number of measured qubits).

Create a copy of the Result object used as an input for StateTomographyFitter().

Index of the experiment in which circuit without noise was run (this will be used to see the performance between final result with and without ZNE).

Create a copy of the Results object with the direct quantum measurements.

Create a copy of the Results with measurement filter applied.

Iterate through all tomography measurements.

Iterate through all qubit states. We will update those state results with ZNE.

Iterate through all experiments - that is circuits with different noise scales.

```
            zne_coef = [(x-1)*2+1 for x in zne_pattern]
            zne_noise = sum(zne_coef) / len(zne_coef)

            if zne_noise == 1.0:
                circuit_without_noise_idx = idx

            expected_state = 0
            try:
                if measurement_filter:
                    expected_state = mitigated_results.get_counts()[idx*num_tomo_circ
+ tomo_circ_idx][qubits_state] / shots
                else:
                    expected_state = jobs.result().get_counts()[idx*num_tomo_circ +
tomo_circ_idx][qubits_state] / shots
            except (AttributeError, KeyError):
                pass
            x.append(zne_noise)
            y.append(expected_state)

        fitparams, _ = curve_fit(zne_fit_func, x, y, maxfev=100000)

        fit_data = zne_fit_func(np.linspace(0, 3, 20), *fitparams)

        zne_counts = int(fit_data[0]*shots)

        exp_results[qubits_state] = zne_counts if zne_counts >= 0 else 0


    for key in sorted(exp_results):
        result_dict = zne_result.results[tomo_circ_idx].to_dict()



        result_dict['data']['counts']['0x'+str(int(key,2))] = exp_results[key]

        zne_result.results[tomo_circ_idx] = \
                  zne_result.results[tomo_circ_idx].from_dict(result_dict)


print('zne_result:', zne_result.get_counts()[-1])
print('w/o zne results:', unmittigated_result.get_counts()[-1])

fid = state_tomo(zne_result, circuits[0:num_tomo_circ])
print("fidelity with zne:", fid)
fid = state_tomo(unmittigated_result, circuits[0:num_tomo_circ])
print("fidelity w/o zne:", fid)
if measurement_filter:
    fid = state_tomo(meas_filter_result, circuits[0:num_tomo_circ])
    print("fidelity w/o zne with meas filter:", fid)
```

Calculate the noise scale (zne_noise) based on the number of Ising circuits repetitions.

If the noise scale is 1.0 we store the index of the experiment.

Use the mitigated or direct results.

expected_state is the number of measurements in a specific state.

Store the scale noise as “x” and store the expected state for specific qubit state as “y”.

Find the optimal extrapolation function parameters for the scale noise and qubit state.

Extrapolate the ZNE results into other scale noise, the extrapolated result at x=0 will be our zero noise scale result.

Store the zero noise scale result into exp_results object. If the extrapolation result is below 0 set it to 0.

Based on the extrapolated results update a new Result object (zne_result).
We store the job results into a dictionary.

We update the dictionary counts data without our updated estimation.

We apply the new results by using the .from_dict() function to populate the Result object.

Display the results with ZNE and without.

Calculate fidelity by passing zne_result object and run standard state tomography with the original circuit to have a comparison.

We reported the following results of the 3 qubit Ising simulation with ZNE:

- Qubits Calibration & measurement classifier training was performed before the experiments were run,
- All experiments ZNE circuits were run on the same day 20 March 2022, so the results can be compared:

8 trotter steps X & SX gates additional calibration Custom measurement classifier Carr-Purcel DD pattern with $n = 2$ 6 different noise scaling circuits for ZNE IBM job id: 623698f98293e938331e495d	
Fidelity with ZNE and measurement filter	0.5911023447447307
Fidelity w/o ZNE and measurement filter	0.36902706943802505
Fidelity w/o ZNE and w/o measurement filter	0.34979264037318536

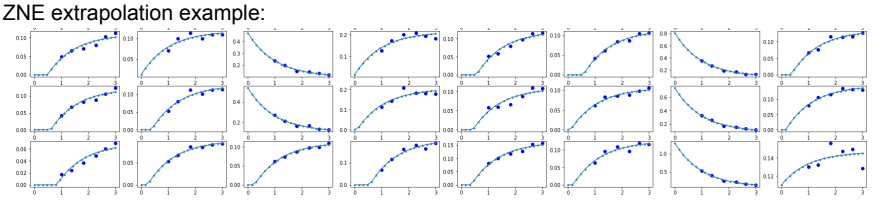
9 trotter steps X & SX gates additional calibration Custom measurement classifier Carr-Purcel DD pattern with $n = 4$ 6 different noise scaling circuits for ZNE IBM job id: 623723048293e9ba701e4b24	
Fidelity with ZNE and measurement filter	0.7096436197928739
Fidelity w/o ZNE and measurement filter	0.43605507857416864

Fidelity w/o ZNE and w/o measurement filter	0.4119806030238874

9 trotter steps X & SX gates additional calibration Custom measurement classifier Carr-Purcel DD pattern with $n = 2$ 6 different noise scaling circuits for ZNE IBM job id: 62372a0819e6895e55c7fb8c	
Fidelity with ZNE and measurement filter	0.7366945273703155
Fidelity w/o ZNE and measurement filter	0.46170425236804835
Fidelity w/o ZNE and w/o measurement filter	0.4359226315894702

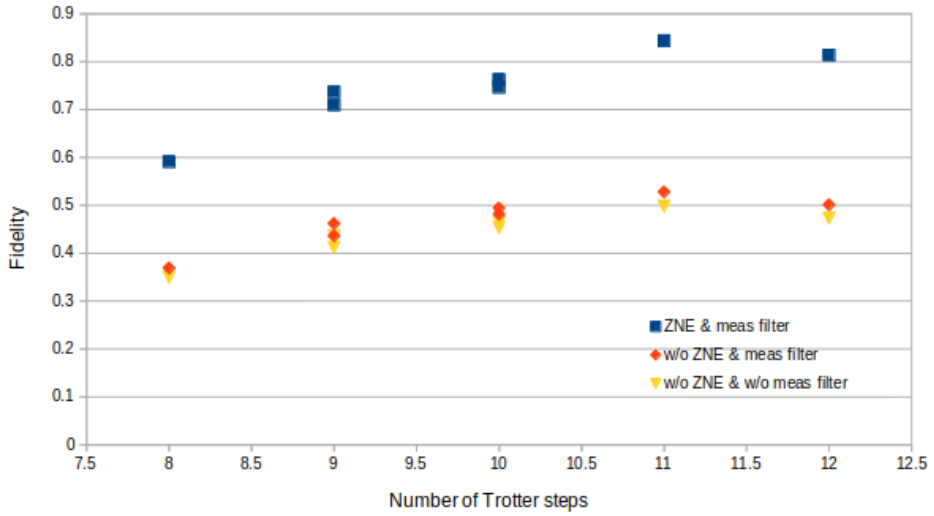
10 trotter steps X & SX gates additional calibration Custom measurement classifier Walsh DD pattern with $k = 5$ 6 different noise scaling circuits for ZNE IBM job id: 6236a195ecc4138876b703c4	
Fidelity with ZNE and measurement filter	0.746374921398376
Fidelity w/o ZNE and measurement filter	0.4813589703355625
Fidelity w/o ZNE and w/o measurement filter	0.4536954734764774

10 trotter steps X & SX gates additional calibration Custom measurement classifier Carr-Purcel DD pattern with $n = 4$ IBM job id: 62373d4209995c28eb491671	
Fidelity with ZNE and measurement filter	0.7627213077698899
Fidelity w/o ZNE and measurement filter	0.4941170415935472
Fidelity w/o ZNE and w/o measurement filter	0.46507034800999175

11 trotter steps X & SX gates additional calibration Custom measurement classifier Carr-Purcel DD pattern with $n = 4$ 6 different noise scaling circuits for ZNE IBM job id: 623754a3d97bff37d8693572	
Fidelity with ZNE and measurement filter	0.8436757037388082
Fidelity w/o ZNE and measurement filter	0.5278307709439489
Fidelity w/o ZNE and w/o measurement filter	0.49812467158765333
ZNE extrapolation example: 	

12 trotter steps X & SX gates additional calibration Custom measurement classifier Carr-Purcel DD pattern with $n = 4$ 6 different noise scaling circuits for ZNE IBM job id: 62376a4ae32b423728ecc711	

Fidelity with ZNE and measurement filter	0.8136720085593458
Fidelity w/o ZNE and measurement filter	0.5008132776248124
Fidelity w/o ZNE and w/o measurement filter	0.4733116515700832



We managed to execute a few following circuits with a measurement classifier train 6 days ago (the same as in the results above). We use the default qiskit calibration because the ibm device was heavily utilized so we did not have the time to perform the calibration:

11 trotter steps Custom measurement classifier Walsh DD pattern with $k = 5$ 10 different noise scaling circuits for ZNE IBM job id: 623eceebd97bff328f694f00	
Fidelity with ZNE and measurement filter	0.8045209893329075
Fidelity w/o ZNE and measurement filter	0.5122969971541214
Fidelity w/o ZNE and w/o measurement filter	0.47969823184302096

11 trotter steps Custom measurement classifier Carr-Purcel pattern with $n = 4$ 10 different noise scaling circuits for ZNE IBM job id: 623ec0c974de0e833f85b645	
Fidelity with ZNE and measurement filter	0.810684730182189
Fidelity w/o ZNE and measurement filter	0.5680693964869504
Fidelity w/o ZNE and w/o measurement filter	0.48257606845954387

11 trotter steps Custom measurement classifier Carr-Purcel DD pattern with $n = 4$ 10 different noise scaling circuits for ZNE IBM job id: 623ec33ba2f72d3234dabd1a	
Fidelity with ZNE and measurement filter	0.8259556033433962
Fidelity w/o ZNE and measurement filter	0.5399594111699143
Fidelity w/o ZNE and w/o measurement filter	0.5052097986995577

Those results prove the 11 Trotter steps provide good overall simulation fidelity.

The ibmq_jakarta seems not to be fully stable with long trotter simulation (11 steps+) and we received few time Internal Error [9348] (Internal Error [9348]).

Application to other problems

ZNE technique is very generic and does not need information about the noise model in the quantum computer. But this technique is very sensitive to extrapolation methods, to reduce this problemy many circuits should be run to reduce the noise in single measurements.
For zero noise estimation noise needs to be increased so only short models (with initial low noise) can benefit from this technique.
Increasing the noise by scaling the circuit pulses requires additional calibration - we would need to create the standard gate - X, SX, CNOT gates with extended pulses (the relation between simple pulse amplitude and duration is approximately linear - so only fine tuning is needed).

The error mitigation techniques require many circuits to be executed to get a better overview of the noise in the system. Some techniques require gate tomography which increases the number of circuits.

Algorithm mitigation techniques are very specific to overall quantum circuits and they are not universal, but the idea can be explored with other algorithms than digital simulation.

Symmetry verification heavily relies on some knowledge of the process / quantum circuit. This technique required additional ancillas qubits which are not always available. Trap ion quantum computers with some designs can allow easier preparation of the superposition for the ancilla qubit. For superconductive quantum computers preparing the ancilla qubits can require many 2 qubits operators (SWAPs and CNOTs) operations which will induce further noise in the circuit.

Works Cited

Arrazola, I., et al. “Pulsed Dynamical Decoupling for Fast and Robust Two-Qubit Gates on Trapped Ions.” https://arxiv.org/pdf/1706.02877.pdf.

Ball, H., and M. J. Biercuk. “Walsh-synthesized noise-filtering quantum logic.” https://arxiv.org/pdf/1410.1624.pdf.

Ball, Harrison, and Michael J. Biercuk. “Walsh-synthesized noise filters for quantum logic.” *EPJ Quantum Technology*, 2015.

Bardhan, Bhaskar R., et al. “Dynamical Decoupling in Optical Fibers:.” https://arxiv.org/pdf/1105.4164.pdf.

Baum, Yuval, et al. “Experimental Deep Reinforcement Learning for Error-Robust Gateset Design on a Superconducting Quantum Computer.” https://arxiv.org/pdf/2105.01079.pdf.

Bultrini, Daniel, et al. “Unifying and benchmarking state-of-the-art quantum error mitigation techniques.” https://arxiv.org/pdf/2107.13470.pdf.

“Calibrating Qubits with Qiskit Pulse.” *Qiskit*, https://qiskit.org/textbook/ch-quantum-hardware/calibrating-qubits-pulse.html. Accessed 31 March 2022.

“Calibrating single-qubit gates on ibmq_armonk — Qiskit Experiments 0.2.0 documentation.” *Qiskit*, 14 December 2021, https://qiskit.org/documentation/experiments/tutorials/calibrating_armonk.html. Accessed 31 March 2022.

Carr, et al. “Effects of diffusion on free precession in nuclear magnetic resonance experiments.” *Physical review*, vol. 94, 1954, p. 630.

Chang, Chih-Chung, and Chih-Jen Lin. “LIBSVM: A Library for Support Vector Machines.” https://www.csie.ntu.edu.tw/~cjlin/papers/libsvm.pdf.

Childs, Andrew M., et al. “Faster quantum simulation by randomization.” https://arxiv.org/pdf/1805.08385.pdf.

Childs, Andrew M., et al. “A Theory of Trotter Error.” https://arxiv.org/abs/1912.08854.

“Classifier comparison — scikit-learn 1.0.2 documentation.” *Scikit-learn*, https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html. Accessed 31 March 2022.

Cummins, Holly K., et al. “Tackling Systematic Errors in Quantum Logic Gates with Composite Rotations.” https://arxiv.org/pdf/quant-ph/0208092.pdf.

“Decision tree learning.” *Wikipedia*, https://en.wikipedia.org/wiki/Decision_tree_learning. Accessed 31 March 2022.

de Fouquieres, P., et al. “Second order gradient ascent pulse engineering.” https://arxiv.org/pdf/1102.4096.pdf.

Doria, P., et al. “Optimal Control Technique for Many-Body Quantum Dynamics.” *Physical review letters*, vol. 106, 2011, p. 190501, doi:10.1103/PhysRevLett.106.190501.

“Drag — Qiskit 0.34.2 documentation.” *Qiskit*, https://qiskit.org/documentation/stubs/qiskit.pulse.library.Drag.html. Accessed 31 March 2022.

“DynamicDecouplingSequence — Open Controls Python package | Q-CTRL.” *Q-CTRL Documentation*, https://docs.q-ctrl.com/open-controls/references/qctrl-open-controls/qctrlopencontrols/DynamicDecouplingSequence.html#qctrlopencontrols.DynamicDecouplingSequence. Accessed 31 March 2022.

Earnest, Nathan, et al. “Pulse-efficient circuit transpilation for quantum applications on cross-resonance-based hardware.” https://arxiv.org/pdf/2105.01063.pdf.

Earnest-Noble, Nathan. “Pulse Efficient Quantum Circuits.” https://learn.qiskit.org/content/summer-school/2021/resources/lecture-notes/Lecture9.2.pdf.

“EchoedCrossResonanceHamiltonian — Qiskit Experiments 0.2.0 documentation.” *Qiskit*, 14 December 2021, https://qiskit.org/documentation/experiments/stubs/qiskit_experiments.library.characterization.EchoedCrossResonanceHamiltonian.html. Accessed 31 March 2022.

Endo, et al. “Hybrid quantum-classical algorithms and quantum error mitigation.” *Journal of the Physical Society of Japan*, vol. 90, 2021, p. 032001.

Gambetta, J. M., et al. “Analytic control methods for high fidelity unitary operations in a weakly nonlinear oscillator.” https://arxiv.org/pdf/1011.1949.pdf.

Giurgica-Tiron, Tudor, et al. “Digital zero noise extrapolation for quantum error mitigation.” https://arxiv.org/pdf/2005.10921.pdf.

Goerz, Michael. “Optimizing Robust Quantum Gates in Open Quantum Systems.” 2015, https://kobra.bibliothek.uni-kassel.de/handle/urn:nbn:de:hebis:34-2015052748381.

Ichikawa, Tsubasa, et al. “Designing Robust Unitary Gates: Application to Concatenated Composite Pulse.” https://arxiv.org/pdf/1105.0744.pdf.

“Improving readout performance in superconducting qubits using machine learning | Application notes | Boulder Opal.” *Q-CTRL Documentation*, https://docs.q-ctrl.com/boulder-opal/application-notes/improving-readout-performance-in-superconducting-qubits-using-machine-learning. Accessed 31 March 2022.

Jin, Shi, and Xiantao Li. “A partially random trotter algorithm for quantum hamiltonian.” https://arxiv.org/pdf/2109.07987.pdf.

“k-nearest neighbors algorithm.” *Wikipedia*, https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm. Accessed 31 March 2022.

LaRose, et al. “Mitiq: A software package for error mitigation on noisy quantum computers.” https://arxiv.org/pdf/2009.04417.pdf.

Lloyd, Seth. “Universal quantum simulators.” *Science*, vol. 273, 1996, pp. 1073-1078.

Lowe, Angus, et al. “Unified approach to data-driven quantum error mitigation.” https://arxiv.org/pdf/2011.01157.pdf.

Malekakhlagh, Moein, and Easwar Magesan. “Mitigating off-resonant error in the cross-resonance gate.” https://arxiv.org/pdf/2108.03223.pdf.

Mari, Andrea, et al. “Extending quantum probabilistic error cancellation by noise scaling.” https://arxiv.org/abs/2108.02237.

Menke, Tim. “Realizing a Calibration Program for Superconducting Qubits.” https://qudev.phys.ethz.ch/static/content/science/Documents/semester/Tim_Menke_Semester_Thesis_130829.pdf.

“Microwave Amplifiers for Quantum Information Processing | Seminar Series with Florent Q. Lecocq.” *YouTube*, 28 January 2022, https://www.youtube.com/watch?v=sVRWtSv0boc. Accessed 31 March 2022.

“Naive Bayes classifier.” *Wikipedia*, https://en.wikipedia.org/wiki/Naive_Bayes_classifier. Accessed 31 March 2022.

Nation, Paul D., et al. “Scalable mitigation of measurement errors on quantum computers.” https://arxiv.org/pdf/2108.12518.pdf.

“Neural network.” *Wikipedia*, https://en.wikipedia.org/wiki/Neural_network. Accessed 31 March 2022.

Otten, Matthew, and Stephen Gray. “Accounting for Errors in Quantum Algorithms via Individual Error Reduction.” https://arxiv.org/pdf/1804.06969.pdf.

Proctor, Timothy, et al. “Detecting and tracking drift in quantum information processors.” *Nature Communications*, vol. 11, 2020, p. 5396, https://doi.org/10.1038/s41467-020-19074-4.

“qiskit-community/open-science-prize-2021.” *GitHub*, https://github.com/qiskit-community/open-science-prize-2021. Accessed 31 March 2022.

Rademacher, H. *Math. Ann*, vol. 87, 1922, pp. 112-138.

Sarovar, Mohan, et al. “Detecting crosstalk errors in quantum information processors.” https://arxiv.org/pdf/1908.09855.pdf.

Satoh, et al. “Pulse-engineered Controlled-V gate and its applications on superconducting quantum device.” https://arxiv.org/abs/2102.06117.

“Spin echo.” *Wikipedia*, https://en.wikipedia.org/wiki/Spin_echo. Accessed 31 March 2022.

“Support-vector machine.” *Wikipedia*, https://en.wikipedia.org/wiki/Support-vector_machine. Accessed 31 March 2022.

“System properties.” *IBM Quantum*, https://quantum-computing.ibm.com/lab/docs/iql/manage/systems/properties. Accessed 31 March 2022.

Tacchino, Francesco, et al. *Quantum computers as universal quantum simulators: state-of-art and perspectives*, https://arxiv.org/pdf/1907.03505.pdf.

Theis, L. S., et al. “Counteracting systems of diabaticities using DRAG controls: The status after 10 years.” https://arxiv.org/pdf/1809.04919.pdf.

Tucci, Robert R. “An Introduction to Cartan’s KAK Decomposition for QC Programmers.” https://arxiv.org/pdf/quant-ph/0507171.pdf.

Tycko, R. “Broadband population inversion.” *Physical Review Letters*, vol. 51, 1983, p. 775.

Uhrig, Gotz S. “Keeping a quantum bit alive by optimized π-pulse sequences.” *Physical Review Letters*, no. 98, 2007, p. 100504.

van den Berg, Ewout, et al. “Probabilistic error cancellation with sparse Pauli-Lindblad models on noisy quantum processors.” https://arxiv.org/pdf/2201.09866.pdf.

Vidal, et al. “Universal quantum circuit for two-qubit transformations with three controlled-NOT gates.” *Physical Review A*, vol. 69, 2004, p. 010301.

Viola, L., et al. *Phys. Rev. Lett.*, vol. 82, p. 24177.

West, Jacob R., et al. “High fidelity quantum gates via dynamical decoupling.” https://arxiv.org/pdf/1012.3433.pdf.

Wu, Re-Bing, et al. “A Data-Driven Gradient Algorithm for High-Precision Quantum Control.” https://arxiv.org/pdf/1712.01780.pdf.

Xu, Yilun, et al. “Automatic Qubit Characterization and Gate Optimization with QubiC.” https://arxiv.org/pdf/2104.10866.pdf.