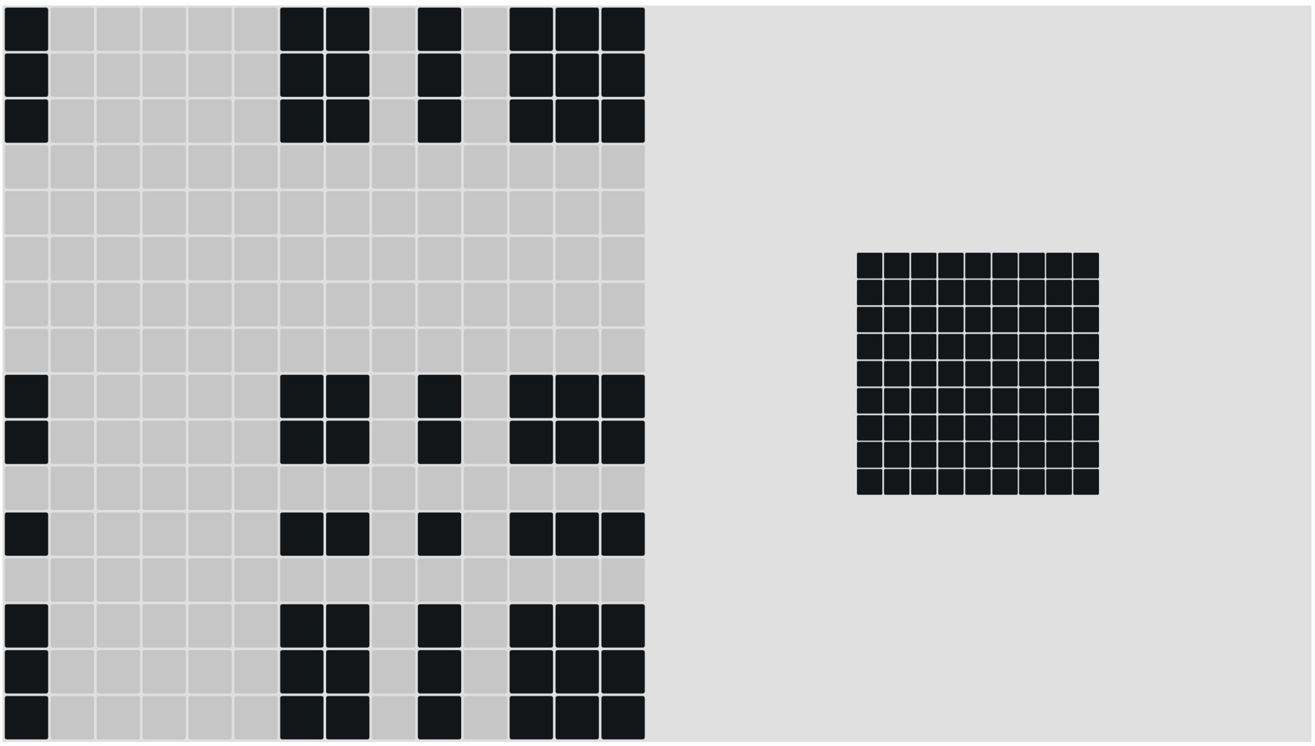
Exploring measurement error mitigation with the Mthree Qiskit extension

Mthree is a powerful tool from the Qiskit ecosystem that tackles measurement errors faster and more accurately than many other comparable methods.



Date

3 Apr 2024

Authors

Hwajung Kang

Topics

Qiskit · Error Correction & Mitigation

Share this blog

problems with quantum computation, we must deploy error handling methods that compensate for those errors. Eventually, researchers hope to achieve quantum error correction, a future capability that will allow quantum computers to correct errors as they appear during computations. Until practical error correction becomes a reality, we rely on simpler error handling methods like quantum error mitigation.

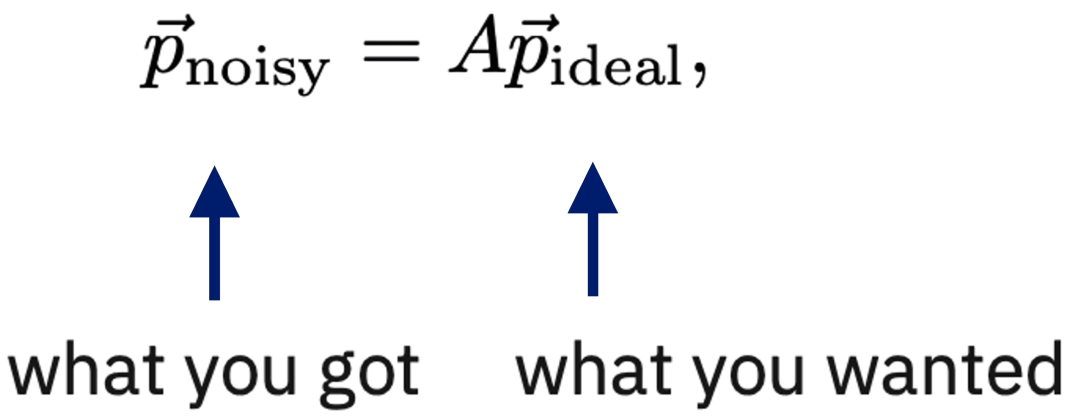
Quantum error mitigation refers to post-processing techniques that allow researchers to counteract the effects of noise once computations are complete. Until error correction becomes a reality, near-term quantum experiments will rely heavily on error mitigation. Error mitigation played a pivotal role in the 2023 IBM Quantum "utility" experiment, and there's also good reason to believe error mitigation will be important for reducing overhead requirements to enable quantum error correction and the truly large-scale quantum applications of the future.

But to realize the full potential of quantum error mitigation, we must continuously develop tools that allow researchers to apply the latest error mitigation techniques in their experiments. More specifically, we must deliver well-tested, performant software that helps to accelerate research in the field. By experimenting with these tools, quantum computational scientists will gain important insights that will help them use error mitigation more effectively.

In this blog post, we'll take an in-depth look at one such tool, currently available in the `qiskit-extension` module, Qiskit-Extensions/mthree. This software package, colloquially referred to as Mthree or M3, implements the **m**atrix-free **m**easurement **m**itigation (M3) method, which is designed for scalable measurement error mitigation on quantum computing platforms.

What are measurement errors?

M3 is a quantum error mitigation package that focuses specifically on what we call "measurement errors." While there are many different types of errors present in current and near-term quantum computers, measurement errors are among the most common. The measurements we perform to collect information from the qubits in our circuits have the largest error rates of any single instruction we can give a quantum computer. Therefore, measurement errors have a substantial impact on near-term, short-depth circuits. Fortunately, the effect of measurement errors is well described by a simple linear system as follows:



$$\vec{p}_{\text{noisy}} = A\vec{p}_{\text{ideal}},$$

where $\vec{p}_{noisy}$ and $\vec{p}_{ideal}$ represent the output probability vector of the measured bit strings with measurement noise and without measurement noise, respectively. Here, A is the $2^n \times 2^n$ “Assignment” matrix (where n is the number of qubits) that probabilistically maps a given ideal measurement outcome to the noisy outcome we actually observe.

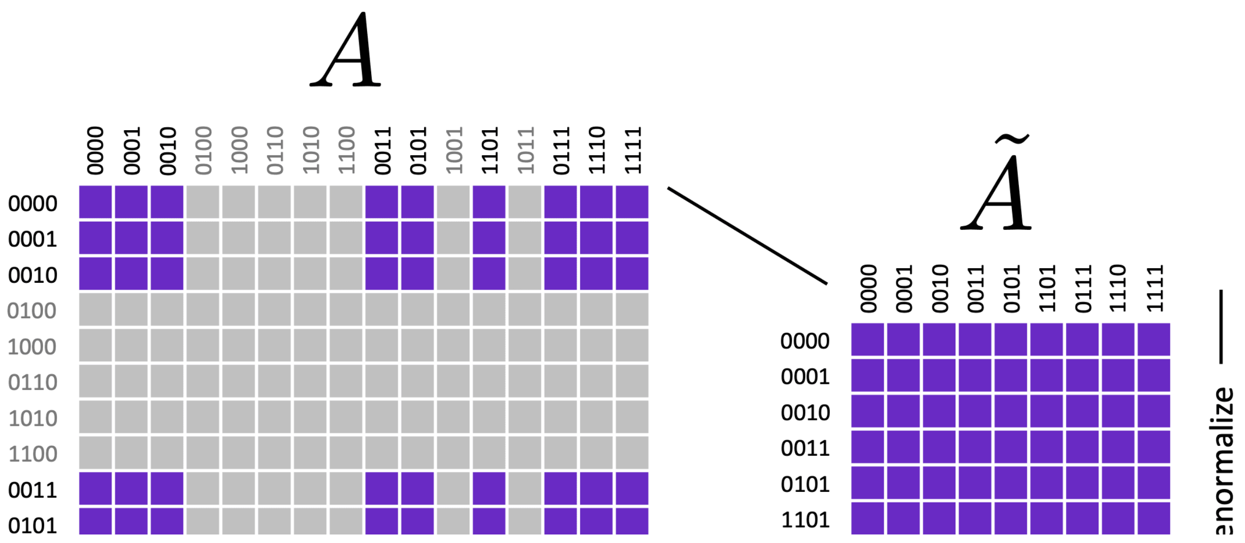
Naively, we can mitigate the measurement errors by multiplying the inverse of the assignment matrix, A^{-1}, with the output probability vector of the measured bit strings, $\vec{p}_{ideal}$. In this case, we can construct A by executing circuits for the 2^n basis states and measuring them in the computational basis. However, as you might imagine, this approach is not practical for large numbers of qubits, since the number of circuits to run and the dimensions of A and $\vec{p}$ grow exponentially with n.

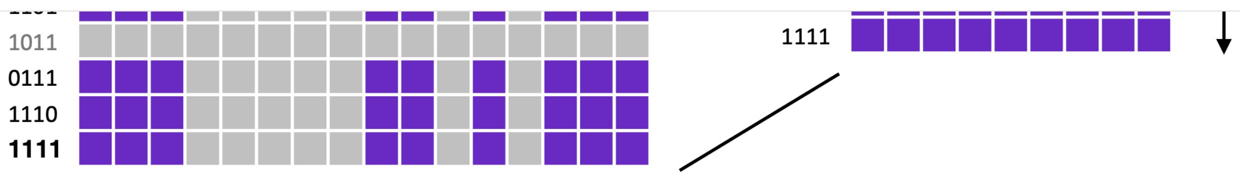
One idea for alleviating these resource requirements is to approximate A by performing a tensor product of the 2×2 assignment matrix of each qubit, assuming there is no correlated error. In this case, we need $2 \times n$ basis state circuits, where we prepare each circuit pair in the basis states of an individual qubit, 0 and 1, and then measure them. Although we can use tensor product approximation to reduce the required number of calibration circuits to be linear in the number of measured qubits, the dimension of the assignment matrix remains exponential, $2^n \times 2^n$. This is one of the main issues that most measurement error mitigation methods face.

How M3 tackles quantum measurement errors

M3 can help us sort out this problem. If measurement errors are small — on the order of a percent or so — and circuits are sampled sufficiently, M3 can take advantage of the fact that the noisy outcome bit strings actually contain all the ideal bit strings as a subset; the signal is hidden in the noise.

Instead of working with the full $2^n \times 2^n$ A matrix, we can perform error mitigation in a reduced $k \times k$ subspace defined by the k unique bit-strings in the noisy output. With k being at most equal to the number of samples taken (number of shots), k can be much smaller than 2^n, especially for a large n.





To understand this idea in detail, let’s consider a simple two-qubit example.

Suppose that, out of four possible bit strings, we obtain three bit strings — 0 (00), 2 (10), and 3 (11) with some probabilities denoted by ${p_{ij}}^{noisy}$, as shown in the figure below. With enough shots and an error rate of only a few percent, we can ignore the terms in the shaded yellow region. This reduces the A matrix dimension down to 3×3 from $2^2 \times 2^2$. This truncation can be huge when we’re working with a large number of qubits, and therefore reduces the matrix size of A dramatically.

$$\vec{p}_{ideal} = [p_{00}^{ideal}, p_{01}^{ideal}, p_{10}^{ideal}, p_{11}^{ideal}] \qquad \vec{p}_{noisy} = [p_{00}^{noisy}, p_{01}^{noisy}, p_{10}^{noisy}, p_{11}^{noisy}]$$

A^{q0} : Assignment Matrix for q^0 $\qquad$ A^{q1} : Assignment Matrix for q^1

$$p_{00}^{noisy} = A_{00}^{q1}A_{00}^{q0}p_{00}^{ideal} + A_{00}^{q1}A_{01}^{q0}p_{01}^{ideal} + A_{01}^{q1}A_{00}^{q0}p_{10}^{ideal} + A_{01}^{q1}A_{01}^{q0}p_{11}^{ideal}$$

$$p_{01}^{noisy} = A_{00}^{q1}A_{10}^{q0}p_{00}^{ideal} + A_{00}^{q1}A_{11}^{q0}p_{01}^{ideal} + A_{01}^{q1}A_{10}^{q0}p_{10}^{ideal} + A_{01}^{q1}A_{11}^{q0}p_{11}^{ideal}$$

$$p_{10}^{noisy} = A_{10}^{q1}A_{00}^{q0}p_{00}^{ideal} + A_{10}^{q1}A_{01}^{q0}p_{01}^{ideal} + A_{11}^{q1}A_{00}^{q0}p_{10}^{ideal} + A_{11}^{q1}A_{01}^{q0}p_{11}^{ideal}$$

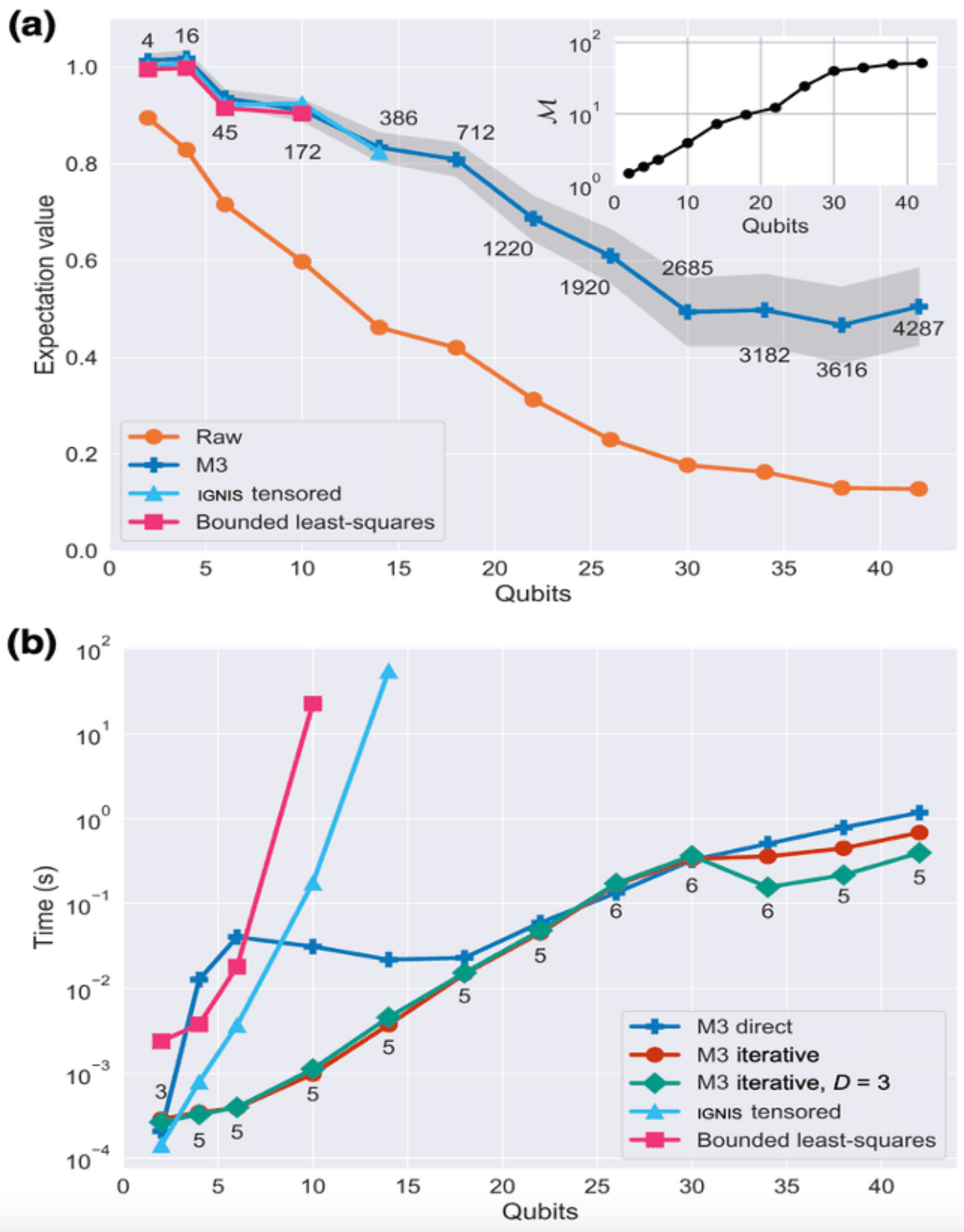
$$p_{11}^{noisy} = A_{10}^{q1}A_{10}^{q0}p_{00}^{ideal} + A_{10}^{q1}A_{11}^{q0}p_{01}^{ideal} + A_{11}^{q1}A_{10}^{q0}p_{10}^{ideal} + A_{11}^{q1}A_{11}^{q0}p_{11}^{ideal}$$

Of course, there are also cases where the number of output bit strings is so high that even the reduced assignment matrix, $\widetilde{A}$, could be larger than the available memory. In these cases, M3 solves the systems of linear equation iteratively, without constructing any matrix.

Iterative methods are usually slow. However, M3 takes advantage of the fact that $\widetilde{A}$ is diagonal dominant thanks to the small size of the measurement error. Applying a preconditioning technique and starting the iterations at $\vec{p}_{noisy}$ results in rapid convergence in just a few iterations using orders of magnitude less memory.

Below is a figure from our paper showing M3 performance compared to other mitigation methods in

At 42 qubits, M3 iteratively mitigates the errors in 1.2 seconds at most and uses about 1 megabyte of memory. This is impressive given that the size of the memory required to store sparse full A matrix for this case is about 580 PiB. That's 120x more memory than that of Fugaku, widely regarded as one of the fastest supercomputers in the world.



However, being able to mitigate the impact of measurement error efficiently, even at large scales, doesn't come for free. Applying M3 will improve the accuracy of your results, but it also increases the

We can compensate for this by taking more samplings. The size of the sampling overhead is bounded by a multiplicative factor, M, which we can estimate using the one-norm of the inverse of the reduced assignment matrix as $M = \| \widetilde{A}^{-1} \|^2$. This indicates the number of additional shots M3 needs to take to make up for the unavoidable loss of precision.

M3 in action

Now that we have a good theoretical understanding of the method that powers M3, let's wrap things up with a simple example that will show you how to actually use it. To execute the codes shown below, you'll need to install M3 by running the command found on this page.

The following code constructs Bernstein-Vazirani (BV) algorithm circuits and executes them on a hypothetical system we call *FakeSherbrooke*. There are a total of eight circuits and the number of qubits in each circuit grows by one starting from two. All the BV circuits here are designed to produce m-bit strings of 1, where m is the number of qubits in the circuit. So, for example, a circuit with three qubits should generate 111 when executed.

```
from qiskit import *
from qiskit_ibm_runtime.fake_provider import FakeSherbrooke
from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manager
import mthree as m3
import numpy as np
import matplotlib.pyplot as plt

def myfunc(my_str):
    num_str = len(my_str)
    ind = num_str - 1 - np.where(np.array(list(my_str)) == '1')[0]
    qc = QuantumCircuit(num_str+1, name ='oracle')
    qc.cx(ind, num_str)
    U = qc.to_gate()
    return U


def BV(U):
    num_str = U.num_qubits - 1
    qc = QuantumCircuit(num_str + 1, num_str)
    qc.x(num_str)
    qc.h(range(num_str + 1))
    qc.barrier()
    qc.append(U, range(num_str+1))
    qc.barrier()
    qc.h(range(num_str))
    qc.measure(range(num_str), range(num_str))
return qc

bv_all = []
bit_range = range(2, 10)
for k in bit_range:
    U = myfunc('1'*k)
    bv_all.append(BV(U))

backend = FakeSherbrooke()
shots = int(1e4)
pm = generate_preset_pass_manger(optimization_level=3, backend=backend)
trans_BVs = pm.run(bv_all)
counts = backend.run(trans_BVs, shots=shots).result().get_counts()
```

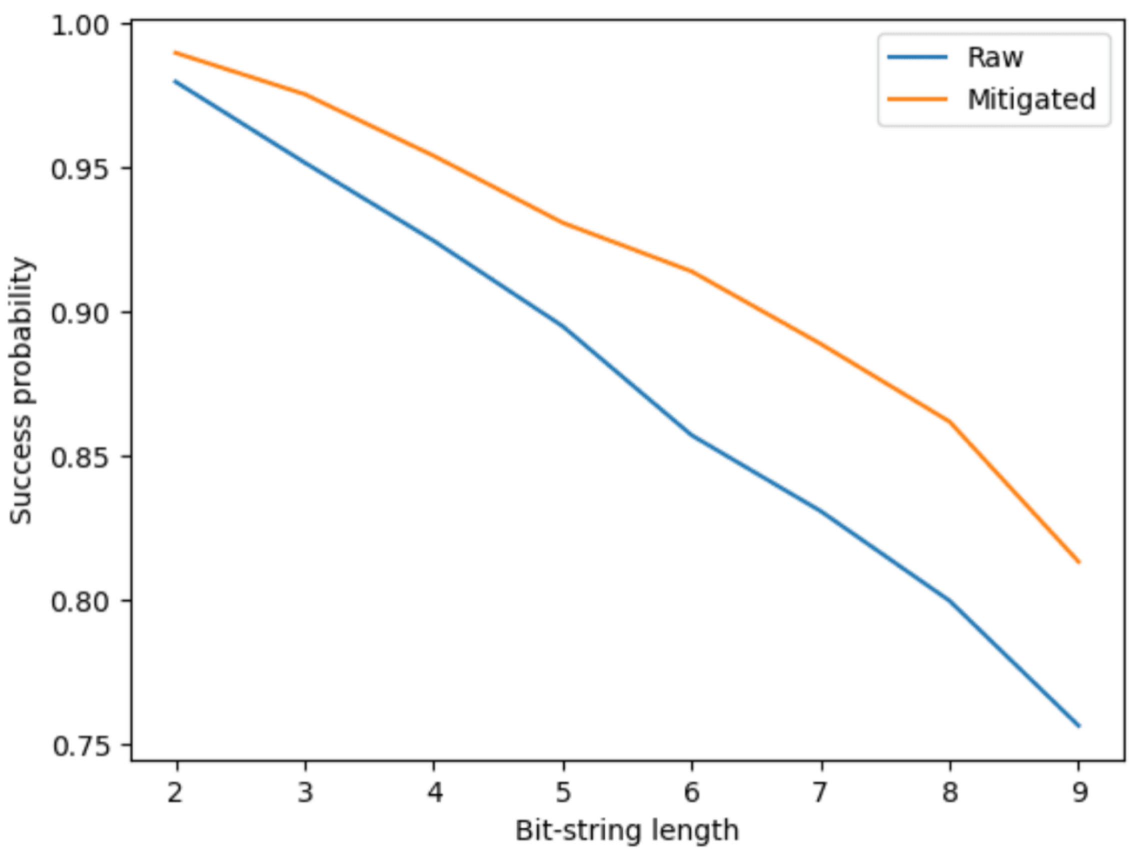
Results we obtain from the *FakeSherbrooke* backend are impacted by various types of noise including measurement noise. The effects of this noise increase with the size of the circuit as shown in the success probability figure below (labeled as "Raw").

We implement M3 on the noisy impacted results by following three steps. First, we find the physical qubits that the virtual qubits map to. Next, we perform calibration on the physical qubits to obtain their run. We can see how success probability improves in comparison to the raw results in the same figure below (labeled as “Mitigated”).

```
mappings = m3.utils.final_measurement_mapping(trans_BVs)
mit = m3.M3Mitigation(backend)
mit.cals_from_system(mappings)
quasis = mit.apply_correction(counts, mappings)

count_probs = [counts[idx].get('1'*num_bits)/shots for idx, num_bits in enumerate(bit_range)]
quasi_probs = [quasis[idx].get('1'*num_bits) for idx, num_bits in enumerate(bit_range)]

fig, ax = plt.subplots()
ax.plot(bit_range, count_probs, label='Raw')
ax.plot(bit_range, quasi_probs, label='Mitigated')
ax.set_ylabel('Success probability')
ax.set_xlabel('Bit-string length')
ax.legend();
```



Making M3 even better

The Bernstein-Vazirani algorithm can also be realized a lot more efficiently by utilizing mid-circuit measurements. And by combining mid-circuit measurements with M3 measurement error mitigation together, we can achieve an enormous improvement in the success probability, as illustrated here.

Of course, as with any piece of code, there are still areas for improvement. One of our chief research renormalization step for constructing A. This is an embarrassingly parallel task involving simple floating-point arithmetic. This is an ideal use case for Graphics Processing Units (GPUs), where the many parallel threads can offer a dramatic speed up.

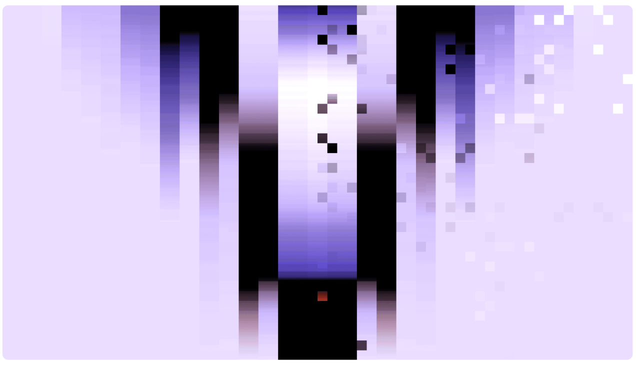
We hope other members of the quantum community will find M3 useful, and we invite everyone to contribute to our open-source project. Together, we can push the limits of quantum error mitigation methods, and discover the true utility — and one day, even the advantage — of quantum computation.

Get started with Qiskit

View documentation

Keep exploring

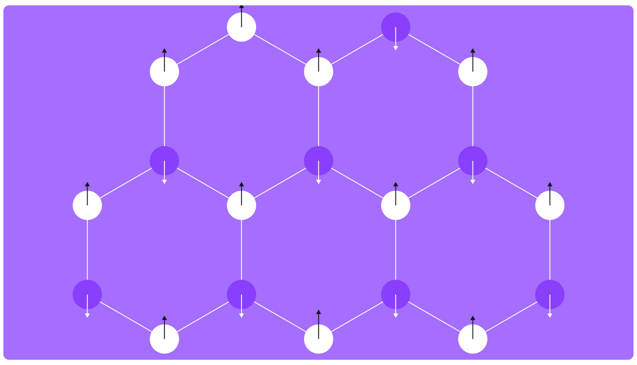
View all blogs →



Simulating the universe’s most extreme environments with utility-scale quantum computation

10 Jul 2024 • Roland C. Farrell, Marc Illa, Anthony N. Ciavarella, Martin J. Savage, Robert Davis

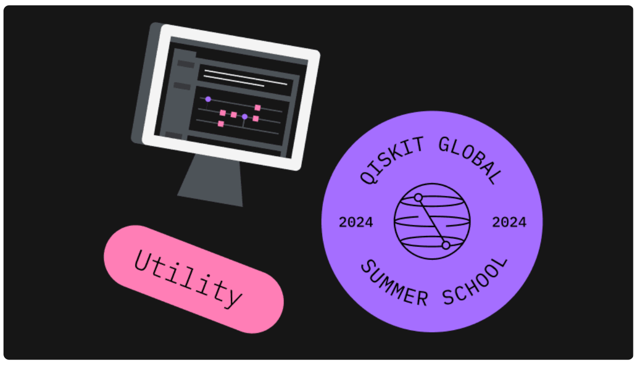
Error Correction & Mitigation | Research | Use Cases | Qiskit



Defining — and citing — the Qiskit SDK

26 Jun 2024 • Ali Javadi-Abhari, Matthew Treinish, Blake Johnson, Robert Davis

Qiskit



[Registration closed] Charting the path to quantum utility at Qiskit Global Summer School 2024

10 Jun 2024 • Olivia Lanes, Aaliyah Fowler, Robert Davis

Community | Enablement | Qiskit

```
from qiskit.circuit.library import RealAmplitudes
from qiskit.quantum_info import SparsePauliOp
from qiskit.transpiler.preset_passmanagers import
generate_preset_pass_manager
from qiskit_ibm_runtime import
QiskitRuntimeService, EstimatorV2 as Estimator

service = QiskitRuntimeService()
backend = service.least_busy(operational=True,
simulator=False)

psi = RealAmplitudes(num_qubits=2, reps=2)
hamiltonian = SparsePauliOp.from_list([("II", 1),
("IZ", 2), ("XI", 3)])
```

Explore newly recommended notebook environments for Qiskit

3 Jun 2024 • Francisco Martin, Suhare Nur, Robert Davis

Enablement | Qiskit