CMSC657 Final Project: Implementation of $[[5, 1, 3]]$ Quantum Error Correction Code

Wance Wang

Instructor: Daniel Gottesman

December 15, 2021

1 Project Introduction

This project aims to implement 5-qubit $[[5, 1, 3]]$ error correction code on a real quantum computer. Five-qubit code (the "perfect" code proposed in 1996 [2]) can protect a logical qubit from an arbitrary single-qubit error using the minimal number of physical qubits. Demonstration of 5-qubit code has been experimentally realized in NMR system [3], and recently in superconducting qubit system [4]. This project mainly reproduce some results in Ref. [4] using IBM online free superconducting quantum computer. It is structured: Sec. 2 discusses the encoding of logical qubit, Sec. 3 verifies the error syndrome of all single-qubit error in 5 physical qubits. To do that, basic tomography methods are used. Sec. 4 discusses the fidelity of encoding into code space.

1.1 Hardware

In this project I used ibmq_manila, which is one of the IBM Quantum Falcon r5.11L Processors [6, 7] which was updated in 01/2021. This is 5-qubit superconducting quantum computer, with quantum volume QV = 32. Around the time (12/12/2021) my experiments running on the computer, its system configuration (remains static on the timescale of typical experiments) and system properties (dynamically calibrated once over a 24-hour period, see [8]) are shown in Fig. 1.

During my using of ibmq_manila, waiting computing jobs in queue are 20-50 on average. The wait time from new job is submitted to the job starts to be computed is usually one hour. For the computing time of one of my job (60 independent circuits, each repeats for 1024 shots) takes about 15 min.

1.2 Software

In companion with IBM hardware, I used their python SDK package qiskit [9] and IBM quantum Lab cloud programming environment. In my program, I mainly used provider modules (`providers`, `providers.Aer`, `providers.ibmq`), circuit building and compiling modules (`circuit`, `qobj`, `quantum_info`, `compiler`) and analysis modules (`result`, `visualization`) in qiskit. For more detail, please check Appendix 5 for my codes.

2 Encoding

2.1 Encoding circuit with nearest-neighbor gates

Encoding a logical qubit is not trivial. On one thing, one needs an efficient encoding circuit [10]. On the other, real-world devices put experimental constraints on operations one can do, for example, superconducting qubits can only do two-qubit gate with its nearest neighbor. Luckily, 5-qubit code encoding circuit has been described [2]. Then in Ref. [4], the qubits are relabeled from conventional $1, 2, 3, 4, 5$ to $1', 3', 5', 4', 2'$ in order to make the two-qubit gates to be nearest-neighbour gates, see below Fig. 2. Here the gates used in the circuit

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

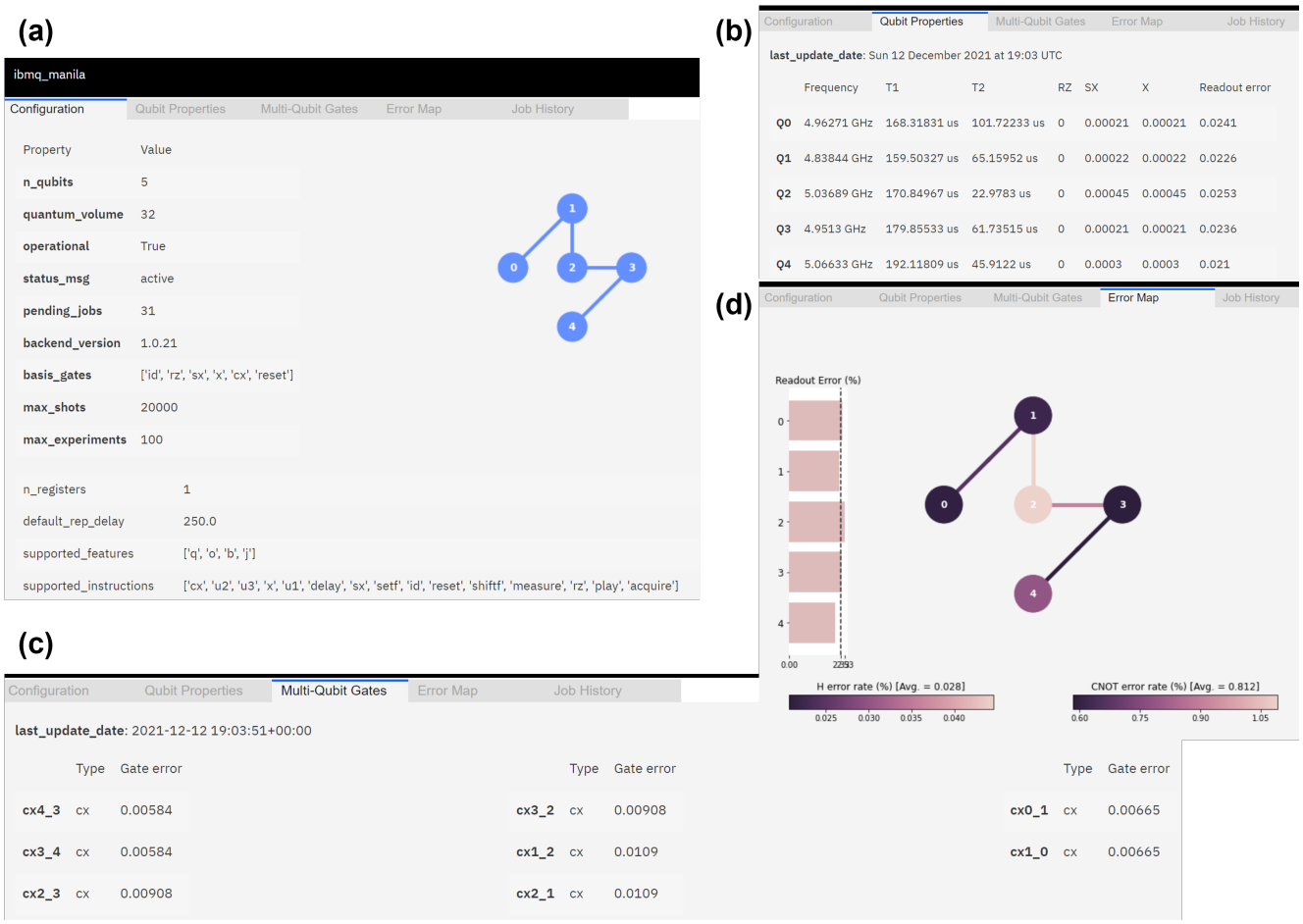


Figure 1: (a) General configurations (b) Qubit frequencies, coherence and single-qubit gate errors and readout errors. T_1 and T_2 are the energy relaxation time and dephasing time, respectively. SX is $\sqrt{X}$ single-qubit gate error, X is single-qubit Pauli-X error. (c) CNOT gate errors (d) Error maps of ibmq_manila.

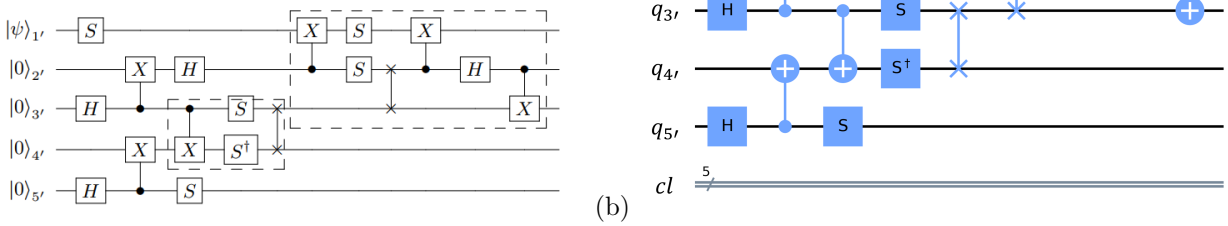


Figure 2: Encoding circuit with qubit relabeled from conventional $1, 2, 3, 4, 5$ to $1', 3', 5', 4', 2'$. $\bullet - \oplus$ is CNOT gate, $\times - \times$ is swap gate. (a) Figure copied from Fig. S1 of Ref. [4]. (b) Draw the same circuit by qiskit.

$$\text{CNOT}(1,2) = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{pmatrix}, \text{SWAP} = \text{CNOT}(1,2)\text{CNOT}(2,1)\text{CNOT}(1,2) = \begin{pmatrix} 1 & & & \\ & & 1 & \\ & 1 & & \\ & & & 1 \end{pmatrix}$$

It only involves 6 nearest-neighbour CNOT gates and 2 swap gates. Since a swap gate can be implemented by 3 CNOT gates, there are 12 CNOT gates in total.

With new labels, the $[[5, 1, 3]]$ stabilizers become

$$\begin{aligned} X_1Z_2Z_3X_4I_5 &\to g_1 = X_{1'}I_{2'}Z_{3'}X_{4'}Z_{5'} \\ I_1X_2Z_3Z_4X_5 &\to g_2 = I_{1'}X_{2'}X_{3'}Z_{4'}Z_{5'} \\ X_1I_2X_3Z_4Z_5 &\to g_3 = X_{1'}Z_{2'}I_{3'}Z_{4'}X_{5'} \\ Z_1X_2I_3X_4Z_5 &\to g_4 = Z_{1'}Z_{2'}X_{3'}X_{4'}I_{5'} \end{aligned}$$

Except for explicit explanation, I will use new label $1', 2', 3', 4', 5'$ and sometimes ignore the prime in remaining parts of this report. Denote $\{g_1, g_2, g_3, g_4\}$ as stabilizer group S, its code space is

$$T(S) = \{|\phi\rangle | M|\phi\rangle = |\phi\rangle \forall M \in S\}$$

2.2 Calculation of logical qubit state

classical simulation To know if the encoding circuit and my programming work, I use qiskit `quantum_info` module to simulate output state.

The logical qubit states after relabeling from conventional form (Eq. (10.104, 10.105) in Ref. [1]) to $|q_{1'}q_{2'}q_{3'}q_{4'}q_{5'}\rangle$ basis are

$$\begin{aligned} |0_L\rangle =& \frac{1}{4}[|00000\rangle + |10010\rangle + |01100\rangle + |10001\rangle \\ &+ |00110\rangle - |11110\rangle - |00011\rangle - |10100\rangle \\ &- |11101\rangle - |01010\rangle - |10111\rangle - |01111\rangle \\ &- |11000\rangle - |00101\rangle - |11011\rangle + |01001\rangle], \\ |1_L\rangle =& \frac{1}{4}[|11111\rangle + |01101\rangle + |10011\rangle + |01110\rangle \\ &+ |11001\rangle - |00001\rangle - |11100\rangle - |01011\rangle \\ &- |00010\rangle - |10101\rangle - |01000\rangle - |10000\rangle \\ &- |00111\rangle - |11010\rangle - |00100\rangle + |10110\rangle]. \end{aligned} \tag{1}$$

States $|\Psi_L\rangle = a|0_L\rangle + b|1_L\rangle$ are in code space, $|\Psi_L\rangle \in T(S)$. The output states and their Hinton plots from classical unitary simulator are shown in Fig. 3. One thing must take care is the annoying qubit ordering convention used in qiskit: n-th qubit is on the left side of the tensor product, so that the basis vectors are labeled as $q_n \otimes ... \otimes q_1 \otimes q_0$. Keep this in mind and I checked the results match Eq. 1.

(a)

logical $|0_L\rangle$

$$\frac{1}{4}|00000\rangle - \frac{1}{4}|00011\rangle - \frac{1}{4}|00101\rangle + \frac{1}{4}|00110\rangle$$
$$\frac{1}{4}|01001\rangle - \frac{1}{4}|01010\rangle + \frac{1}{4}|01100\rangle - \frac{1}{4}|01111\rangle$$
$$\frac{1}{4}|10001\rangle + \frac{1}{4}|10010\rangle - \frac{1}{4}|10100\rangle - \frac{1}{4}|10111\rangle$$
$$-\frac{1}{4}|11000\rangle - \frac{1}{4}|11011\rangle - \frac{1}{4}|11101\rangle - \frac{1}{4}|11110\rangle$$

logical $|1_L\rangle$

$$-\frac{1}{4}|00001\rangle - \frac{1}{4}|00010\rangle - \frac{1}{4}|00100\rangle - \frac{1}{4}|00111\rangle$$
$$-\frac{1}{4}|01000\rangle - \frac{1}{4}|01011\rangle + \frac{1}{4}|01101\rangle + \frac{1}{4}|01110\rangle$$
$$-\frac{1}{4}|10000\rangle + \frac{1}{4}|10011\rangle - \frac{1}{4}|10101\rangle + \frac{1}{4}|10110\rangle$$
$$\frac{1}{4}|11001\rangle - \frac{1}{4}|11010\rangle - \frac{1}{4}|11100\rangle + \frac{1}{4}|11111\rangle$$

(b)

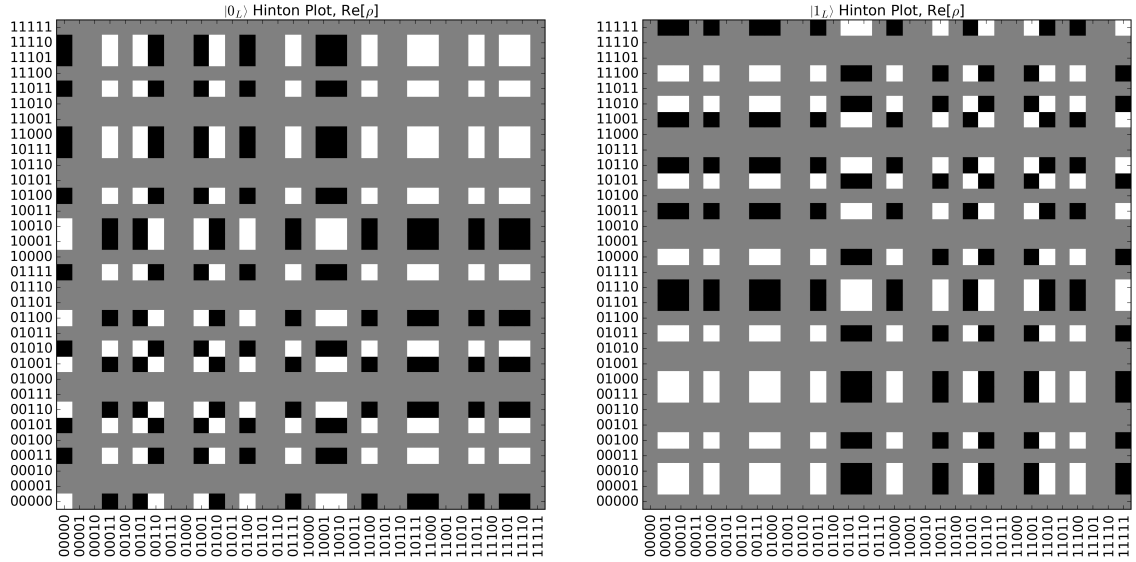
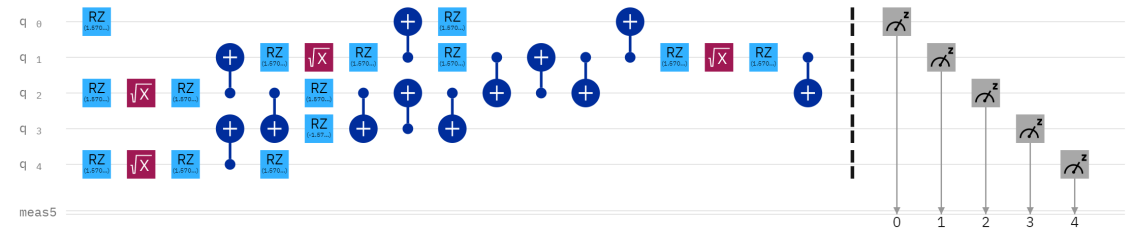


Figure 3: Logical state $|0_L\rangle, |1_L\rangle$ unitary simulation output (a) in Latex form (b) Hinton plot of $\mathrm{Re}\,[\rho]$ (no noise is added, so $\mathrm{Im}\,[\rho]$ are zero.). White block means $\rho_{ij} > 0$, black block means $\rho_{ij} < 0$, and grey block means $\rho_{ij} = 0$. Pay attention, the qubit ordering is $q_n \otimes ... \otimes q_1 \otimes q_0$.

(a)



(b)

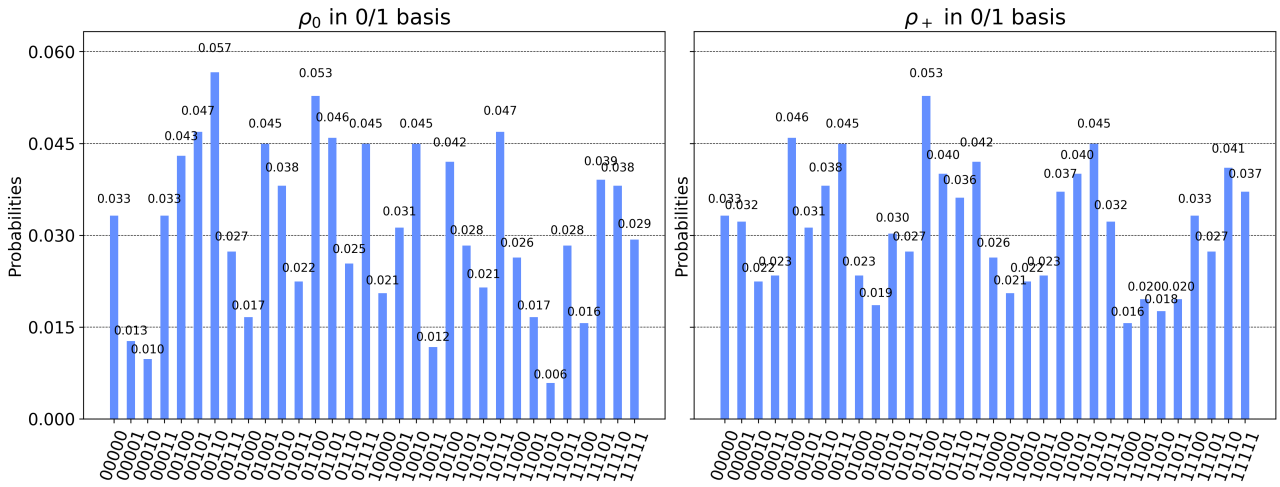


Figure 4: (a) Compiled circuit downloaded from backend job details (b) Probability histograms of encoded logical state $|0_L\rangle$ and $|+_L\rangle$ on basis states, 1024 shots.

on quantum computer Logical states $|0_L\rangle$ and $|+_L\rangle = \frac{1}{\sqrt{2}}(|0_L\rangle + |1_L\rangle)$ are implemented on ibmq_manila quantum computer, density matrices ρ_0, ρ_+ with each circuit repeated for 1024 times. Here, one can only measure qubits then get state counts or probabilities on basis states, so histogram plots of ρ_0, ρ_+ as well as IBM compiled circuit (`qiskit.compiler.transpile()`) are shown in Fig. 4. All 32 basis states appear in two histograms, which shows that there is error in real quantum circuit. But I can also find that, some low counts basis, for example, $00001, 11001$, are not present in $|0_L\rangle$.

2.3 Encoding circuit with minimal number of CZ gates

To reduce errors, it is better to have minimal number of two-qubit gates. In Ref. [4], this is achieved by numerical optimization of 2 dashed circuit blocks in Fig. 2(a), and Fig. 5 is the result with minimal number of 8 CZ gates. Meanwhile, single-qubit gates becomes complicated. I don't try this circuit in this project.

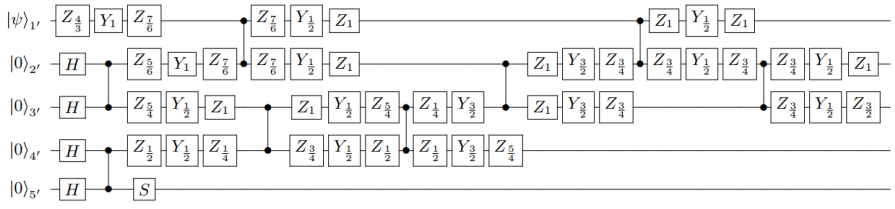


Figure 5: Numerically optimized circuit with minimal 8 two-qubit CZ gates. Copied from Fig. S4 of Ref. [4].

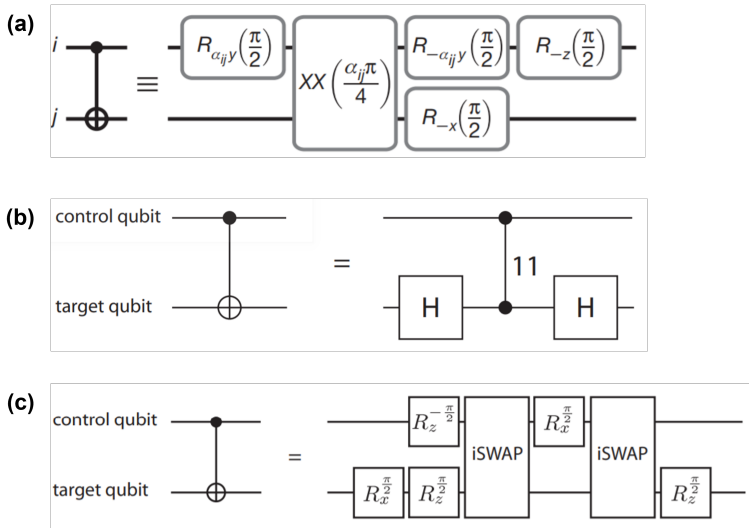


Figure 6: (a) Copied from Fig. 2(a) of Ref. [11]. Decomposition of the CNOT gate into XX-gate. The geometric phase χ_{ij} of the XX-gate is $\pm\pi/4$, and define $\alpha_{ij} = \text{sgn}(\chi_{ij})$. (b) Copied from Fig. 2.4 of Ref. [14]. Decomposition of the CNOT gate into C-Phase gate $cU_{11} = \exp(i\pi/4)\text{diag}(1, 1, 1, -1)$. (c) Copied from Fig. 2.5 of Ref. [14]. Decomposition of the CNOT gate into iSWAP gate.

2.4 Encoding circuit with hardware-compatible gates

In many quantum computing architectures, CNOT is not the intrinsic hardware gates. Thus it is necessary to express circuits in terms of hardware-compatible gates when interfacing and optimizating a quantum processor. It is usually called compilation. Hardware-compatible gates for two most popular hardware architectures:

- Trapped ion: Molmer-Sorensen gate (XX-gate), see Fig. (a) The implementation of CNOT can be found in [11] or thesis [12].
- Superconducting qubits: C-Phase (ZZ-gate) or iSWAP gate (XY-gate), related papers can be [13] or thesis [14, 15].

Inserting these decomposition into Fig. 2 to replace CNOT, we get hardware-compatible circuits for different architectures. Particularly, Fig. 5 is already such a gate with CZ gate. I don't implement them in my project because I find IBM platform will compile the circuit with its own way (`qiskit.compiler.transpile()`), like Fig. 4(a). Therefore, it is meaningless to submit a hardware-compatible gates. Besides, it is meaningless to use a trapped-ion compatible circuit on superconducting qubit processor as well.

3 Verification of Error Syndrome (destructively)

With encoded logical states, I am going to verify whether measuring stabilizer g_m generates expected syndrome. Since the device used here only has 5 qubit, correction assisted by ancilla qubits is impossible, so the measurements are destructively performed on all physical qubits.

3.1 Ideal error syndromes

With relabeled stabilizers,

$$
\begin{aligned}
g_1 &= XIZXZ \\
g_2 &= IXXZZ \\
g_3 &= XZIZX \\
g_4 &= ZZXXI
\end{aligned}
$$

ideal error syndromes are listed in Table 1.

$g_1g_2g_3g_4$	X error	Y error	Z error
$q_{1'}$	$1,1,1,-1$	$-1,1,-1,-1$	$-1,1,-1,1$
$q_{2'}$	$1,1,-1,-1$	$1,-1,-1,-1$	$1,-1,1,1$
$q_{3'}$	$-1,1,1,1$	$-1,-1,1,-1$	$1,-1,1,-1$
$q_{4'}$	$1,-1,-1,1$	$-1,-1,-1,-1$	$-1,1,1,-1$
$q_{5'}$	$-1,-1,1,1$	$-1,-1,-1,1$	$1,1,-1,1$

Table 1: Ideal error syndrome table for g_m, $m = 1, 2, 3, 4$

3.2 Actual error syndromes verification

After encoding logical states, X, Y, Z errors are intentionally injected into each qubit. What I need is expectation value of stabilizer $\langle g_m \rangle$. However, this is not straightforward. If I apply g_m stabilizer in the end of circuit and measure all qubits in $|0\rangle, |1\rangle$ basis, probabilities of getting each basis state are generated like Fig. 4(b),

$$|q_5q_4q_3q_2q_1\rangle : \text{Prob}\, p_\alpha$$

Basis state $|q_5q_4q_3q_2q_1\rangle \notin T(S)$ ($T(S)$ is code space), but one cannot say anything about the value of g_m based on p_α: if final state $|\psi\rangle \in T(S)$, the value of g_m is 1, if not, -1. Due to the complexity of logical state Eq. 1, relating any basis state measurement to logical state is not direct. Thus I didn't know how to calculate $\langle g_m \rangle$ in this way.

Method 1: full tomography A straight way is to do full quantum state tomography (QST) to get full elements of density matrix ρ_q (ρ_q is a encoded state), then calculate $\langle g_m \rangle$, or do quantum process tomography (QPT) [1] directly to the result. Ref. [4] did QPT to calculate $\langle g_m \rangle$ and fidelities. But it is hard for me to implement a full tomography program in my limited time and effort for this project.

See a brief introduction of principle of QST in Appendix. 5.

Therefore, if we measure all Stokes-like parameters $S_{i_1,i_2,\ldots i_n}$ mentioned in the appendix, we get full information of ρ using Stokes representation Eq. 10. To get all expectation values for N-qubit system, 4^N parameters $\langle \sigma_i\sigma_j \rangle$ are required, 3^N tomographic gates and measurements ($\sigma_x, \sigma_y, \sigma_z$ each needs individual measurement for one qubit) are needed and 2^N probability results are collected in one measurement, as suggested in Ref. [22] supplemental material. For the $[[5, 1, 3]]$ code, it needs $4^5 = 1024$ values, $3^5 = 243$ measurements and produces 6^5 probabilities. In addition, real tomography need to perform a "maximum likelihood technique" in order to reconstruct a legitimate density matrix, since various errors don't guarantee a positive semi-definite reconstruction. The complexity of a real tomography prevents me from doing this method.

Method 2: $\langle g_m \rangle$ derived from Stokes-like parameter Fortunately, stabilizer operators g_m are special: g_m are in Pauli group, so they are naturally in form $\sigma_{i_1} \otimes \sigma_{i_2} \otimes \ldots \otimes \sigma_{i_n}$, and $\langle g_m \rangle$ is naturally Stokes-like parameters Eq. 11. If expectation values are arranged in an order as Eq. 13, then $\langle \sigma_{i_1}\sigma_{i_2} \ldots \sigma_{i_n} \rangle$ is at

$$r = (1 - \delta_{0i_1})2^{n-1} + (1 - \delta_{0i_2})2^{n-2} + .., +(1 - \delta_{0i_n}) + 1$$

-th row (pay attention qiskit ordering is the reverse of this), δ_{0i} is Kronecker delta notation so that when $\sigma_i \neq I$, $\delta_{0i} = 0$. In the right hand of Eq. 11, also pick up the r-th row in matrix T, arrange measurement probabilities column $\big(\langle 0\ldots|\rho_q|0\ldots\rangle \quad \langle 0\ldots1|\rho_q|0\ldots1\rangle \quad \ldots \quad \langle 1\ldots|\rho_q|1\ldots\rangle\big)^\intercal$ in the corresponding order. The expectation value is derived by

$$\langle g_m \rangle = \langle \sigma_{i_1}\sigma_{i_2} \ldots \sigma_{i_n} \rangle = (T)_r \big(\langle 0\ldots|\rho_q|0\ldots\rangle \quad \langle 0\ldots1|\rho_q|0\ldots1\rangle \quad \ldots \quad \langle 1\ldots|\rho_q|1\ldots\rangle\big)^\intercal \tag{2}$$

$$T = \otimes_{j=1}^{n} T_j,\ T_j = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Thus I avoid programming full tomography, this is only a "partial tomography" using the principle. Furthermore, standard deviation is

$$\text{std} := \sqrt{\langle g_m^2 \rangle - \langle g_m \rangle^2} = \sqrt{\langle I \rangle - \langle g_m \rangle^2} = \sqrt{1 - \langle g_m \rangle^2} \tag{3}$$

Again, thanks to g_m in Pauli group, std is easy to calculate. An example circuit to calculate $\langle g_1 \rangle$ by encoded $|+_L\rangle$ state with injected X error on qubit 2 is shown in Fig. 7.

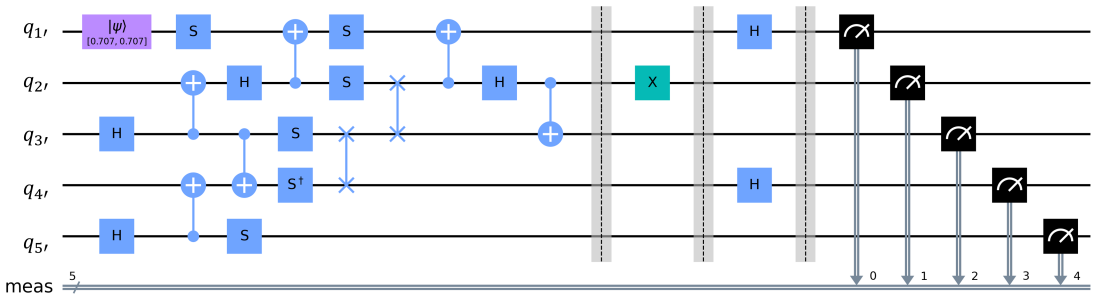


Figure 7: Circuit for calculate $\langle g_1 \rangle$ by encoded $|+_L\rangle$ state. Except for the initial encoding and measurement parts, the green X gate in the middle is injected X error. The blue gates behind it are for "partial tomography" of $g_1 = XIZXZ$, where H is applied for transforming into $|\pm\rangle$.

Error syndromes are combinations of expectation values $\langle g_m \rangle, m = 1 - 4$ with respect to different qubits having different single-qubit errors. Using method 2, actual syndromes are plotted in Fig. 8. Figure 8 (a) and (b) are syndromes calculated by method 2 for $|0_L\rangle$, $|+_L\rangle$. 3 type of single-qubit errors are injected into 5 different qubits respectively, and 4 stabilizers expectation values are calculated in each error, results in 60 different circuits (each repeats 1024 times) for one logical states. Although syndromes from g_1 and g_4 are small, all syndromes have correct sign and patterns comparing to the ideal syndromes in grey bars. This shows both the encoding circuits and $[[5, 1, 3]]$ code work qualitatively correctly. The reason for small $\langle g_1 \rangle$ and $\langle g_4 \rangle$ is unknown, but it is less likely that my program is wrong since all syndromes won't all match the ideal in that case.

For comparison, Figure 8 (c) is Fig. 2 from Ref. [4] where they measured error syndromes for magic state $|T_L\rangle = 1/\sqrt{2}\left(|0_L\rangle + e^{i\pi/4}|1_L\rangle\right)$. It is a destructive measurement on 5 qubits, but full QPT and QST are performed in their experiment. Each circuit is repeated 5000 times. Their syndromes are more significant. The properties for the 12 transmon superconducting qubits quantum processor [5] in their experiments is shown in Fig. 9. Comparing to their device, ibmq_manila that I used has > 5 times longer T_1, T_2^* and ~ 2 times better single-qubit and two-qubit gate errors. Thus my syndrome should be more significant than them. Without more effort to check my program and more knowledge of how IBM online platform executes circuits on the processor, it is hard to explain why my results are poor. From the calculation of standard deviation below, one hypothesis is that IBM platform doesn't have as careful and real-time device calibrations as in Ref. [4] between computation of circuits, during a long set of circuits like 60 circuits I submitted to their processor, errors and drifts accumulate without real-time calibration. Another reason is the tomography, they applied full QST and QPT as well as maximum likelihood technique to generate more reliable reconstruction, while I can only implement method 2 due to limited time. On the other hand, my programming could have mistakes. But 1. all syndromes indeed match without exception. 2. For some parts of program that I worry about, for example, the inner product Eq. 2 in method 2, if I intentionally make the row r wrong or change the transformation matrix T, the output syndromes are randomly scrambled and become much more insignificant. Thus currently I still think my program is correct.

I also calculate the standard deviation std according to Eq. 3, see Fig. 10, the results are very poor because $\langle g_m \rangle$ itself is small. However, std doesn't improve when I increased the number of shots from $N = 2^{10}$ to 2^{14} for each circuit run, which is close to the maximum shots ibmq_manila processor can take. Standard deviation should scales as $\text{std} = \text{std}_0/\sqrt{N}$, so the later std should get 4 times of improvement. Clearly it is not as in Fig. 10(b). What's more, the syndromes even becomes worse. Therefore, as the hypothesis I mentioned above, IBM platform might not have careful and real-time device calibrations between computation of two adjacent circuit runs, so the performance of device gets bad in a large set and large shots of circuits. As for comparison with error bar in Fig. 8(c), it is meaningless because their std should be large as well according to Eq. 3. They actually small error bars should be derived in full tomography with advance algorithms.

4 Fidelity

Once the logical state is encoded, fidelity should be evaluated. But it turns out to be hard without a good tomography, so I discuss it in the latter part of this report.

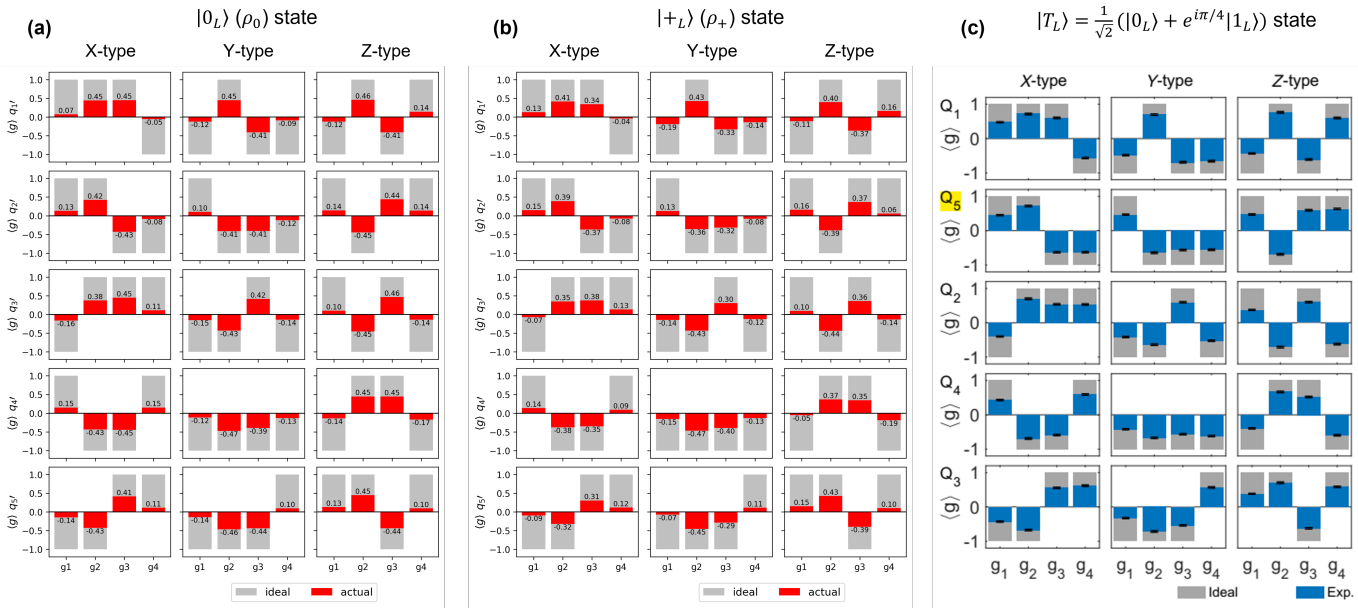


Figure 8: Error syndromes. The vertical axes are different qubits, the horizontal axes are single-qubit errors. Grey bars are ideal syndromes, red or blue bars are actual syndromes got from experiments. (a) Error syndromes for $|0_L\rangle$ state implemented on ibmq_manila in this project. X, Y, Z errors are intentionally injected into each qubit, and for each error, the additional gates to change basis states (see the technique in Appendix 5) are applied and then method 2 is performed. Thus $5 \times 3 \times 4 = 60$ circuits in total are computed to calculate $\langle g_m \rangle$s in $|0_L\rangle$, each circuit has 1024 shots. (b) Error syndromes for $|+_L\rangle$ in this project. The method applied are the same. (c) For comparison, error syndromes for magic state $|T_L\rangle = 1/\sqrt{2}\left(|0_L\rangle + e^{i\pi/4}|1_L\rangle\right)$ implemented in Ref. [4] with their 12 transmon superconducting qubits quantum processor. This figure is copied from Fig. 2 in the paper, note original qubit labels are shown in vertical axes, but they are $1', 2', 3', 4', 5'$ in relabeled order. It is a destructive measurement on 5 qubits, but full QPT and QST are performed in their experiment. They repeated each circuit 5000 times.

Qubit	Q_1	Q_5	Q_2	Q_4	Q_3	AVG.
$\omega_{10}/2\pi$ (GHz)	5.124	4.266	5.006	4.134	4.884	-
T_1 (μs)	27.5	34.0	33.0	36.8	48.6	36.0
T_2^* (μs)	5.5	4.1	5.6	2.7	3.3	4.2
f_{00}	0.982	0.932	0.931	0.934	0.963	0.945
f_{11}	0.831	0.874	0.885	0.899	0.916	0.872
X/2 gate errors per Clifford sequence for the reference	0.0014	0.0017	0.0024	0.0018	0.0015	
X/2 gate errors per Clifford sequence for the interleaved RB	0.0020	0.0023	0.0031	0.0025	0.0022	
X/2 gate fidelity	0.9994	0.9994	0.9992	0.9993	0.9993	0.9993
CZ gate errors per Clifford sequence for the reference	0.037	0.035	0.038	0.026		
CZ gate errors per Clifford sequence for the interleaved RB	0.056	0.045	0.053	0.038		
CZ gate fidelity	0.980	0.990	0.984	0.988		0.986

TABLE S3. Performance of qubits. ω_{10} is idle points of the qubits. T_1 and T_2^* are the energy relaxation time and dephasing time, respectively. f_{00} (f_{11}) is the probability of correctly readout of qubit state in $|0\rangle$ ($|1\rangle$) after successfully initialized in $|0\rangle$ ($|1\rangle$) state. X/2 gate fidelity and CZ gate fidelity are single- and two-qubit gate fidelities obtained via performing randomized benchmarking.

Figure 9: The properties for the 12 transmon superconducting qubits quantum computer in experiments of Ref. [4], figure copied from Table S3 in their paper. The authors mention "After careful calibrations and gate optimizations, we have the average gate fidelities as high as 0.9993 for single-qubit gates and 0.986 for two-qubit gates."

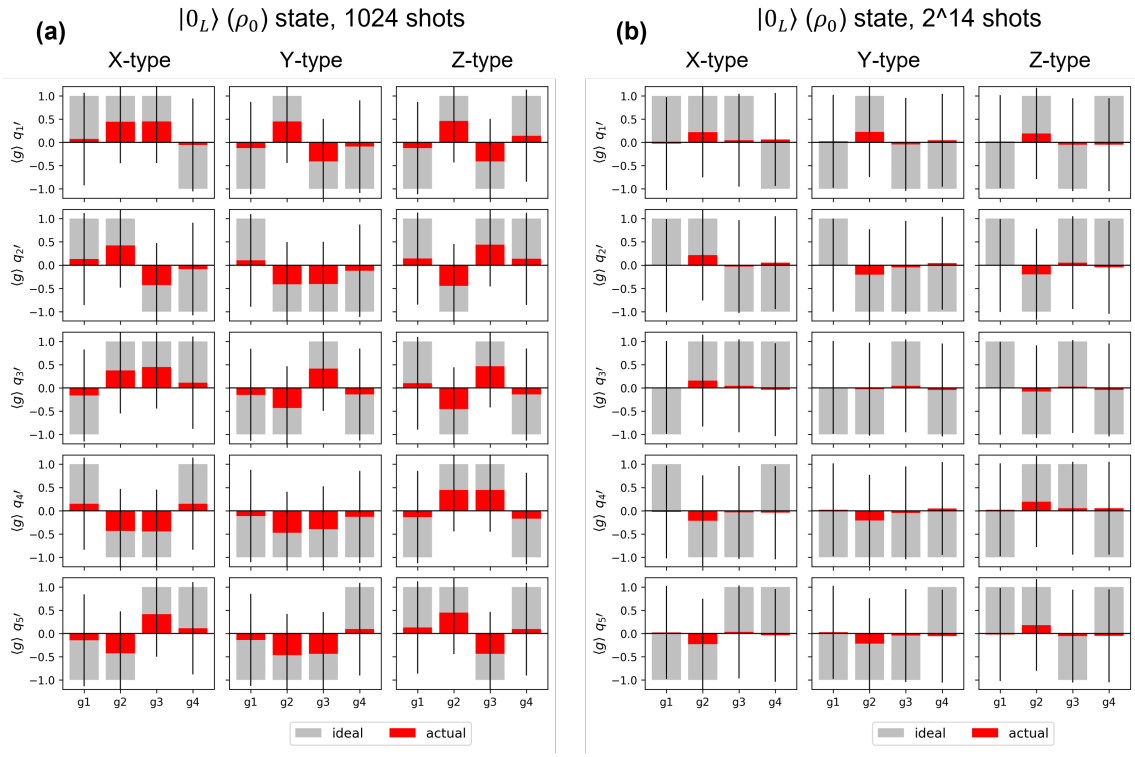


Figure 10: Error syndromes for $|0_L\rangle$ with error bars. (a) Each circuit is repeated 1024 times. (b) Each circuit is repeated 2^{14} times.

4.1 Fidelity of encoding logical qubit states

Suppose the logical state we get is ρ_q after the encoding, the fidelity between experimentally encoded state ρ_q and ideal target logical state $|\Psi_L\rangle = a|0_L\rangle + b|1_L\rangle$ is

$$\mathcal{F} = \langle\Psi_L|\rho_q \mid \Psi_L\rangle = \mathrm{Tr}\left[\rho_q P_\Psi\right]$$

P_Ψ is the projection operator for $|\Psi_L\rangle$, it can be decomposed as

$$P_\Psi = P_C P_{ab} \tag{4}$$

$$P_C = \frac{1}{2^4}\prod_{i=1}^{4}(I + g_i) \tag{5}$$

P_C is the projector into code space. Define $\mathcal{F}_C = \mathrm{Tr}\left[\rho_q P_C\right]$ as the fidelity encoding a state into code space (but not specifying a, b).

$$\begin{aligned} P_{ab} &= \frac{1}{2}\left(I + g_5\right) \\ g_5 &= |\Psi_L\rangle\langle\Psi_L| - |\Psi_L^\perp\rangle\langle\Psi_L^\perp| \\ &= \left(aa^\star - bb^\star\right) Z_L + \left(a^\star b + b^\star a\right) X_L - i\left(a^\star b - b^\star a\right) Y_L \end{aligned} \tag{6}$$

$|\Psi_L^\perp\rangle = b^*|0_L\rangle - a^*|1_L\rangle$ is the state orthogonal to $|\Psi_L\rangle$, thus P_{ab} is the projector to state $|\Psi_L\rangle$ with certain coefficient a, b, $\mathcal{F} = \mathrm{Tr}\left[\rho_q P_\Psi\right]$ is the fidelity preparing the state specifically to $|\Psi_L\rangle$.

calculation To calculate fidelities, I need to calculate trace over P_Ψ and P_C. P_Ψ and P_C are products of stabilizers which are still stabilizers $g_m^{(p)} := g_i g_j ....$ P_C has 15 different $g_m^{(p)}$s when the product Eq. 5 is expanded, and P_Ψ has

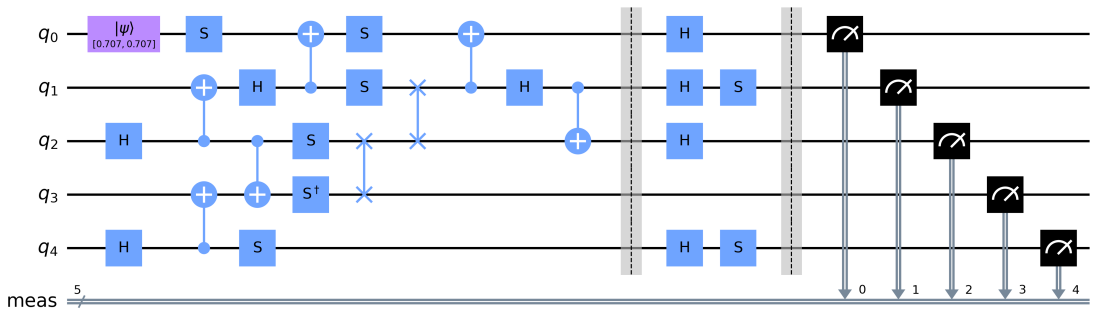


Figure 11: Circuit for calculate $\langle XYXIY\rangle$ by encoded $|+_L\rangle$ state. Except for the encoding and measurement parts, the gates in the middle are for "partial tomography" in method 2, where H is applied for transforming into $|\pm\rangle$, SH for transforming into $|\pm y\rangle$.

31 ($2^n - \binom{n}{0}$). The same method 2 can be used to get expectation of $\left\langle g_m^{(p)}\right\rangle$ because products of stabilizer is still in Pauli group, and sum over to get fidelities.

$$\left\langle g_m^{(p)}\right\rangle = \mathrm{Tr}\left[g_m^{(p)}\rho_q\right] \tag{7}$$

$$\mathcal{F}_C = \frac{1}{2^4}\left(1 + \sum_{m=1}^{15}\left\langle g_m^{(p)}\right\rangle\right) \tag{8}$$

$$\mathcal{F} = \frac{1}{2^5}\left(1 + \sum_{m=1}^{31}\left\langle g_m^{(p)}\right\rangle\right) \tag{9}$$

I first calculate $\mathcal{F}_C$. The explicit form of 15 stabilizers $g_m^{(p)}$

$$\begin{aligned} g_i &: XIZXZ, IXXZZ, XZIZX, ZZXXI \\ g_ig_j &: XXYYI, IZZYY, YZYIZ, XYXIY, ZYIYZ, YIXYX \\ g_ig_jg_k &: IYYXX, YYZZI, ZIYZY, YXIXY \\ g_1g_2g_3g_4 &: ZXZIX \end{aligned}$$

Fig. 11 shows a example circuit for applying method 2 to calculate $\langle XYXIY\rangle$ by encoded $|+_L\rangle$ state. The gates in the middle are for "partial tomography" in method 2, where H is applied for transforming into $|\pm\rangle$, SH for transforming into $|\pm y\rangle$. Do the same thing to all 15 $g_m^{(p)}$ expectation values and uses Eq. 8. The results I get for encoded $|0_L\rangle$ and $|+_L\rangle$ state are

$$\begin{aligned} \mathcal{F}_C(\rho_0) &= \langle 0_L|\rho_0|0_L\rangle = 6.702\% \\ \mathcal{F}_C(\rho_+) &= \langle +_L|\rho_+|+_L\rangle = 6.848\% \end{aligned}$$

To calculate $\mathcal{F}$, it becomes tricky because $g_5 = |\Psi_L\rangle\langle\Psi_L| - |\Psi_L^\perp\rangle\langle\Psi_L^\perp|$ no longer maps to a Stokes-like parameters, thus method 2 is invalid. Decomposition Eq. 6 can reduce it to $X_L = XXXXX, Y_L = YYYYY, Z_L = ZZZZZ$ which are form $\sigma_{i_1}\otimes\sigma_{i_2}\otimes\ldots\otimes\sigma_{i_n}$ and their expectation are Stokes-like parameters, but it again creates problems in calculating the product of g_5 with other g_m in Eq. 4. I don't have enough time to finish the programming of this part.

Discussion The fidelity is quite low. Comparing to Ref. [4] Table S4, they get fidelities 58.6% for $|0_L\rangle$ and 54.1% for $|+_L\rangle$ (summing over stabilizer $\left\langle g_m^{(p)}\right\rangle$, but I think the way they get expectation values are still from tomography), 56.7% for $|0_L\rangle$ and 52.7% for $|+_L\rangle$ (full QST and directly calculate $\mathrm{Tr}\left[\rho_q P_\Psi\right]$). These fidelities are fidelity $\mathcal{F}$, so I

should use my $\mathcal{F}$ to compare. But since $\mathcal{F} \leq \mathcal{F}_C$ and my $\mathcal{F}_C$ are already much poorer than their $\mathcal{F}$, if my fidelity calculation is correct, the encoding on ibmq_manila is very poor, which might show a difference on whether one has full control on the hardware and hardware-related optimization as the hypothesis I mentioned in Sec. 3. Also, the fidelity calculations in Ref. [4] are different from mine.

The poor performance of IBM's quantum processor is not surprise. Although the qubit coherence and gate errors shown in Fig. 1, recent performance benchmarks [23] still show that many quantum processors are poor dealing with application-oriented computation with large circuit width and depth. To evaluate $[[5, 1, 3]]$ code syndrome and fidelities, the circuits I used (for example, Fig. 11) have circuit width 5 (5 qubits) and usually circuit's physical depth more than 20, in Ref. [23], I find the fidelity of IBM's quantum processors (QV = 32) often go down to $\sim 20\%$. Thus the fidelities I get don't seem ridiculously low comparing to that.

4.2 Fidelity within the Code Space

In Ref. [4], the authors also evaluate a "fidelity within the code space" $\mathcal{F}_L$ which is defined as

$$\rho_L = \frac{I + \bar{P}_X X_L + \bar{P}_Y Y_L + \bar{P}_Z Z_L}{2}$$

$\bar{P}_j = P_j / P_I$, $P_j = \mathrm{Tr}\left[\rho_q j_L\right]$, $j_L = I, X_L, Y_L, Z_L$. This is just a Stokes representation based on logical Pauli matrices.

$$\mathcal{F}_L = \langle \Psi_L | \rho_L \mid \Psi_L \rangle$$

This fidelity evaluates how the state ρ_L overlaps with target logical state where encoded state ρ_q has been projected from into code space. I don't calculate $\mathcal{F}_L$. Ref. [4] gets $\mathcal{F}_L$ as high as 98.6% on average. It means the encoding infidelity is mainly encoding the state into code space, the encoding within code space is not a problem. This is not surprise since encoding within code space is mainly encoding with particular coefficients a, b in $|\Psi_L\rangle = a|0_L\rangle + b|1_L\rangle$, and that is just accomplished by preparing $|q_1\rangle = a|0\rangle + b|1\rangle$ which is just a single-qubit gate on the 1st qubit.

5 Summary

In this project, I successfully accomplished most of goals in my original proposal.

1. Worked out the encoding circuits, expectation values from tomography, as well as syndrome and fidelity calculation based on papers and some derivations.
2. Chose a suitable IBM processor ibmq_manila. Extracted its hardware properties.
3. Learned qiskit to perform basic operations and various tools to interfacing with online quantum processor/simulators, data processing and result visualizations.
4. Programmed and experimentally encoded $[[5, 1, 3]]$ QEC code on IBM quantum processor, verified all error syndromes correctly matched with acceptable significance, and calculated fidelities on different logical states. Successfully reproduced published work Ref. [4] in a short time.
5. Did basic analysis on the errors and performance and presented some hypothesis on the reason of poor performance. Did necessary comparison with Ref. [4] from hardware to results.
6. Found out decompositions of initial encoding circuits into hardware-compatible circuits, and discussed why it is meaningless to implement.
7. Published my programming code on github.

Failed to finish some alternatives due to limited time and the difficulties of goals:

1. Try Amazon IonQ and compare performances of different quantum computer architectures.
2. Add ancilla qubits so that it is possible to measure syndrome and correct errors non-destructively.

References

[1] Nielsen, Michael A., and Isaac Chuang. "Quantum computation and quantum information." (2002): 558-559. 2.2, 3.2

[2] Laflamme, Raymond, et al. "Perfect quantum error correcting code." Physical Review Letters 77.1 (1996): 198. 1, 2.1

[3] Knill, Emanuel, et al. "Benchmarking quantum computers: The five-qubit error correcting code." Physical Review Letters 86.25 (2001): 5811. 1

[4] Gong, Ming, et al. "Experimental exploration of five-qubit quantum error correcting code with superconducting qubits." arXiv preprint arXiv:1907.04507v2 (2021). 1, 2.1, 2, 2.3, 5, 3.2, 3.2, 8, 9, 4.1, 4.2, 4, 5

[5] Gong, Ming, et al. "Genuine 12-qubit entanglement on a superconducting quantum processor." Physical review letters 122.11 (2019): 110501. 3.2

[6] IBM Quantum. https://quantum-computing.ibm.com/, 2021 1.1

[7] IBM r5.11 Quantum processor, https://quantum-computing.ibm.com/composer/docs/iqx/manage/systems/processors 1.1

[8] IBM Quantum Services, System properties. https://quantum-computing.ibm.com/services/docs/services/manage/systems/p 2021 1.1

[9] ANIS, MD SAJID, et al. Qiskit: An Open-Source Framework for Quantum Computing. 2021, https://doi.org/10.5281/zenodo.2573505. 1.2

[10] Cleve, Richard, and Daniel Gottesman. "Efficient computations of encodings for quantum error correction." Physical Review A 56.1 (1997): 76. 2.1

[11] Debnath, Shantanu, et al. "Demonstration of a small programmable quantum computer with atomic qubits." Nature 536.7614 (2016): 63-66. 6, 2.4

[12] Debnath, Shantanu. A programmable five qubit quantum computer using trapped atomic ions. Diss. University of Maryland, College Park, 2016. 2.4

[13] DiCarlo, Leonardo, et al. "Demonstration of two-qubit algorithms with a superconducting quantum processor." Nature 460.7252 (2009): 240-244. 2.4

[14] Chow, Jerry Moy. Quantum information processing with superconducting qubits. Yale University, 2010. 6, 2.4

[15] Reed, Matthew David. Entanglement and quantum error correction with superconducting qubits. Yale University, 2014. 2.4

[16] James, Daniel F. V., et al. “Measurement of Qubits.” Physical Review A, vol. 64, no. 5, American Physical Society, Oct. 2001, p. 052312. 5, 5

[17] Altepeter, Joseph B., Daniel FV James, and Paul G. Kwiat. "4 qubit quantum state tomography." Quantum state estimation. Springer, Berlin, Heidelberg, 2004. 113-145. 5, 5, 5

[18] Roos, Christian F., et al. "Control and measurement of three-qubit entangled states." science 304.5676 (2004): 1478-1480. 5

[19] Roos, C. F., et al. "Tomography of entangled massive particles." arXiv preprint quant-ph/0307210 (2003). 5

[20] Steffen, Matthias, et al. "Measurement of the entanglement of two superconducting qubits via state tomography." Science 313.5792 (2006): 1423-1425. 5

[21] DiCarlo, Leonardo, et al. "Preparation and measurement of three-qubit entanglement in a superconducting circuit." Nature 467.7315 (2010): 574-578. 5

[22] Song, Chao, et al. "10-qubit entanglement and parallel logic operations with a superconducting circuit." Physical review letters 119.18 (2017): 180511. 3.2, 5

[23] Lubinski, Thomas, et al. "Application-Oriented Performance Benchmarks for Quantum Computing." arXiv preprint arXiv:2110.03137 (2021).

4.1

Appendix I: principle of quantum state tomography (QST)

Multiple qubits density matrix has an Stokes representation [16, 17],

$$\rho = \frac{1}{2^n} \sum_{i_1,i_2,\ldots i_n=0}^{3} S_{i_1,i_2,\ldots i_n} \sigma_{i_1} \otimes \sigma_{i_2} \otimes \ldots \otimes \sigma_{i_n} \tag{10}$$

$$S_{i_1,i_2,\ldots i_n} = \mathrm{Tr}\left[\sigma_{i_1} \otimes \sigma_{i_2} \otimes \ldots \otimes \sigma_{i_n} \rho\right] \tag{11}$$

$$= \langle \sigma_{i_1} \sigma_{i_2} \ldots \sigma_{i_n} \rangle \tag{12}$$

Stokes-like parameter (the multiple-qubit analog of the single-qubit Stokes parameters) $S_{i_1,i_2,\ldots i_n}$ is the expectation value of operator $\sigma_{i_1} \otimes \sigma_{i_2} \otimes \ldots \otimes \sigma_{i_n}$. These parameters can be derived using QST [16, 17] (see applications in trapped ion in [18, 19] and superconducting qubits in [20, 21, 22]). Here I use a single qubit as example, given a state ρ, if we measure in σ_z basis, $|0\rangle, |1\rangle$, the Stokes parameters can be expressed as

$$\langle I \rangle = \langle 0|\rho|0\rangle + \langle 1|\rho|1\rangle$$
$$\langle \sigma_z \rangle = \langle 0|\rho|0\rangle - \langle 1|\rho|1\rangle$$
$$\begin{pmatrix} \langle I \rangle \\ \langle \sigma_z \rangle \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} \langle 0|\rho|0\rangle \\ \langle 1|\rho|1\rangle \end{pmatrix} = T \begin{pmatrix} \langle 0|\rho|0\rangle \\ \langle 1|\rho|1\rangle \end{pmatrix}$$

T is the transformation matrix (the matrix form of T is $\sqrt{2}H$, H is matrix of Hadamard gate). If we want $\langle \sigma_x \rangle$, we need to measure in σ_x basis $|\pm\rangle$,

$$\begin{pmatrix} \langle I \rangle \\ \langle \sigma_x \rangle \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} \langle +|\rho|+\rangle \\ \langle -|\rho|-\rangle \end{pmatrix}$$

Thus additional gate H has to be applied to ρ so that measuring $|0\rangle, |1\rangle$ is equivalent to measuring $|\pm\rangle$, $H|0\rangle = |+\rangle, H|1\rangle = |-\rangle$. Similarly,

$$\begin{pmatrix} \langle I \rangle \\ \langle \sigma_y \rangle \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} \langle +y|\rho|+y\rangle \\ \langle -y|\rho|-y\rangle \end{pmatrix}$$

Additional gate $SH = \begin{pmatrix} 1 & \\ & i \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ i & -i \end{pmatrix}$ has to be applied to ρ so that measuring $|0\rangle, |1\rangle$ is equivalent to measuring $|\pm y\rangle$, $SH|0\rangle = |+y\rangle, SH|1\rangle = |-y\rangle$. Ref. [19] used in principle same gates to measure in different bases. These relations can be generalized to multiple qubits. For 2 qubits, if we want to know $\langle \sigma_x \sigma_z \rangle$, we apply H to the 1st qubit and measure both qubits in $|0\rangle, |1\rangle$ basis,

$$\begin{pmatrix} \langle II \rangle \\ \langle I\sigma_z \rangle \\ \langle \sigma_x I \rangle \\ \langle \sigma_x \sigma_z \rangle \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} \langle +0|\rho|+0\rangle \\ \langle +1|\rho|+1\rangle \\ \langle -0|\rho|-0\rangle \\ \langle -1|\rho|-1\rangle \end{pmatrix} = T_1 \otimes T_2 \begin{pmatrix} \langle +0|\rho|+0\rangle \\ \langle +1|\rho|+1\rangle \\ \langle -0|\rho|-0\rangle \\ \langle -1|\rho|-1\rangle \end{pmatrix} \tag{13}$$

Transformation matrix $T_1 \otimes T_2$ is the special case in Eq. (21) in Ref. [17], in which orthogonal bases $\tau_0 = \begin{pmatrix} 1 & & \\ & & \\ & & \end{pmatrix}, \tau_1 = \begin{pmatrix} & & \\ & 1 & \\ & & \end{pmatrix}$, ... corresponds to the gates and measurements used here, and $(T_1 \otimes T_2)_{ij} = \mathrm{Tr}\left[\tau_i \left(\sigma \otimes \sigma\right)_j\right]$ ($(\sigma \otimes \sigma)_0 = I \otimes I, (\sigma \otimes \sigma)_1 = I \otimes \sigma_z$, $(\sigma \otimes \sigma)_2 = \sigma_x \otimes I$, $(\sigma \otimes \sigma)_3 = \sigma_x \otimes \sigma_z$).

Appendix II: code

The entire programming codes are long, I have published them on my github repository github.com/wwcphy/5-qubit_QEC. See jupyter notebook github.com/wwcphy/5-qubit_QEC/blob/main/5-qubit_QEC_20211215.ipynb.

Some important parts of codes are listed below:

- Encoding

```
# Define encoding circuit function.
def encoding_5q_code(circ, reg_q, reg_c=()):
    '''
    Take a QuantumCircuit() and encode 5-qubit
    error correction code.
    '''

    # 1
    circ.s(reg_q[0])
    circ.h(reg_q[2])
    circ.h(reg_q[4])

    # 2
    circ.cx(reg_q[2], reg_q[1])
    circ.cx(reg_q[4], reg_q[3])

    # 3
    circ.h(reg_q[1])
    circ.cx(reg_q[2], reg_q[3])
    circ.s(reg_q[4])

    # 4
    circ.s(reg_q[2])
    circ.sdg(reg_q[3])
    circ.swap(reg_q[2],reg_q[3])

    # 5
    circ.cx(reg_q[1], reg_q[0])
    circ.s(reg_q[0])
    circ.s(reg_q[1])

    # 6
    circ.swap(reg_q[1],reg_q[2])

    # 7
    circ.cx(reg_q[1], reg_q[0])

    # 8
    circ.h(reg_q[1])

    # 9
    circ.cx(reg_q[1], reg_q[2])

    return circ

# Encoding circuit diagram
from qiskit import QuantumRegister, ClassicalRegister
import qiskit.quantum_info as qqi

reg_q = QuantumRegister(5,'q')
reg_c = ClassicalRegister(5,'cl')
a,b = 1, 0 #1./np.sqrt(2),1./np.sqrt(2)

circ = QuantumCircuit(reg_q,reg_c)
encoding_5q_code(circ, reg_q)
circ.draw('mpl')
```

- Syndrome verification

```
# string for stabilizer operator [g1 ,g2, g3, g4]
str_g = ['xizxz','ixxzz','xzizx','zzxxi']

def tomo_gi_transform(circ,str_gi):
    '''
    Apply tomographyical transformations according to g.

    Parameters
    ----------
    circ : qiskit.QuantumCircuit()
        circ to operate
    str_gi : string
        string of stabilizer
    '''
    for i_qubit, tomo_gate in enumerate(str_gi):
        # 'i': nothing; 'x': apply H; 'y': apply SH; 'z': nothing
        if tomo_gate == 'x':
            circ.h(i_qubit)
        elif tomo_gate == 'y':
            circ.h(i_qubit)
            circ.s(i_qubit)  # phase gate diag(1, i)
    return circ

def syndrome_verification_circ(coeff):
    '''
    Create sydrome verification circuits with intentionally injected error
    and tomographically apply transformations according to g.

    Parameters
    ----------
    coeff : array_like, shape (2, )
        coefficient [a, b] for logical state. |\psi>=a|0_L>+b|1_L>
    '''
    circ_error = []
    errorname = ['x','y','z']
```

```
    for i_qubit in range(5):
        circ_temp = []
        for r_error in errorname:
            circ_r = []
            for str_gi in str_g:
                # create a new circ_s_c with r_error gate on i_qubit, then
                # generate expectation of stabilizer gi by tomography

                # 1. initialize new circuit
                reg_q_temp = QuantumRegister(5,'q')
                circ_r_s = QuantumCircuit(reg_q_temp)
                circ_r_s.initialize(params=coeff,qubits=0)
                # 2. encoding
                encoding_5q_code(circ_r_s,reg_q_temp)
                # 3. inject error
                circ_r_s.barrier()
                getattr(circ_r_s,r_error)(i_qubit)
                # 4. apply tomographyical transformation
                circ_r_s.barrier()
                tomo_gi_transform(circ_r_s,str_gi)
                # 5. measure all qubits in |0>, |1>
                circ_r_s.measure_all()

                circ_r.append(circ_r_s)
            circ_temp += circ_r
        circ_error += circ_temp
    return circ_error

# run syndrome verification circuits on ibmq
def run_circuits(circ_list,qc_name,shots,task_name):
    """
    Run a list of circuits on ibmq quantum computer and
    output job_set_id and job_set.

    Parameters
    ----------
    circ_list : array_like, shape (n, )
        list of qiskit.QuantumCircuit() to run
    qc_name : string
        name of backend
    shots : int
        Number of repetitions of each circuit, for sampling.
    task_name : string
        job_set name
    """
    provider = IBMQ.get_provider(hub='ibm-q')
    backend = provider.get_backend(qc_name)
    circ_complie = transpile(circ_list, backend=backend)
    job_manager = IBMQJobManager()
    job_set = job_manager.run(circ_complie, backend=backend, shots=shots, name=task_name)

    job_set_id = job_set.job_set_id()
    # job_set = job_manager.retrieve_job_set(job_set_id=job_set_id)

    # jobs = job_set.jobs()
    # results = job_set.results()

    return job_set_id, job_set

# generate circuit lists
coeff_0 = [1., 0.]  # |0_L>
coeff_plus = [1./np.sqrt(2), 1./np.sqrt(2)]  # |+_L>

circ_error_0 = syndrome_verification_circ(coeff_0)
circ_error_plus = syndrome_verification_circ(coeff_plus)

# run syndrome verification for |0_L> and |+_L>
run_or_not = False
qc = 'ibmq_manila'
n_shot = 1024
if run_or_not:
    job_set_id_syn_0, job_set_syn_0 = run_circuits(circ_error_0,qc_name=qc,shots=n_shot,task_name='syndrome_0')
    job_set_id_syn_plus, job_set_syn_plus = run_circuits(circ_error_plus,qc_name=qc,shots=n_shot,task_name='syndrome_plus')
else:
    job_set_id_syn_0 = '200f71978bc541578b77b3fec461cf6c-16393620386375632'
    job_set_id_syn_plus = 'd4c5eb1319a04dccb0e8a47c714a59c6-16393620409830256'
    job_manager = IBMQJobManager()
    job_set_syn_0 = job_manager.retrieve_job_set(job_set_id_syn_0,provider)
    job_set_syn_plus = job_manager.retrieve_job_set(job_set_id_syn_plus,provider)

# reshape counts in job_set
def counts_reshape(job_set):
    # counts_all = nd.array(job_set.results().get_counts())
    # counts_all = counts_all.reshape(5,3)
    results = job_set.results()
    counts_all = []
    for i_qubit in range(5):
        counts_i_qubit = []
        for j_error in range(3):
            count_j_error = []
            for k_stabilizer in range(4):
                c_index = i_qubit*3*4 + j_error*4 + k_stabilizer
                count_j_error.append(results.get_counts(c_index))
            counts_i_qubit.append(count_j_error)
        counts_all.append(counts_i_qubit)
    return np.array(counts_all)

# find the row of desired g in expectation values
def g_row(str_g):
    g_row_list = []
    for str_gi in str_g:
        row = 0
        # print(str_gi[::-1])
        for ith, i_char in enumerate(str_gi):
            if i_char != 'i':
                row += 2**ith
        g_row_list.append(row)
    return g_row_list
```

```
# generate the transformation matrix T
def transform_T5():
    matrix_T1 = np.array([[1,1],[1,-1]])
    matrix_T5 = matrix_T1
    for i in range(5-1):
        matrix_T5 = np.kron(matrix_T5,matrix_T1)
    return matrix_T5

# generate basis state in string (e.g., '00110') in corresponding order
basis_list = []
for ith in range(2**5):
    basis_list.append(str(format(ith+2**5,'b'))[1:])
basis_list = np.array(basis_list)

# matrix multiply of transform T and counts
def T5_counts(T5_row,counts_gi):
    counts_col = []
    for basis in basis_list:
        if basis in counts_gi:
            counts_col.append(counts_gi[basis])
        else:
            counts_col.append(0)
    counts_col = np.array(counts_col)
    exp_value = np.dot(T5_row,counts_col)
    return exp_value

# generate all expectation value for gi from all counts
def stabilizer_expectation(counts_all,matrix_T5,g_row_list,n_shot):
    expectation = []
    for i_qubit in range(5):
        exp_i_qubit = []
        for j_error in range(3):
            exp_j_error = []
            for k_stabilizer in range(4):
                prob_ijk = T5_counts(matrix_T5[g_row_list[k_stabilizer]],counts_all[i_qubit,j_error,k_stabilizer])/n_shot
                exp_j_error.append(prob_ijk)
            exp_i_qubit.append(exp_j_error)
        expectation.append(exp_i_qubit)
    return np.array(expectation)

# standard deviation for gi from all counts
def stabilizer_std(counts_all,matrix_T5,g_row_list,n_shot):
    std = np.sqrt(1-stabilizer_expectation(counts_all,matrix_T5,g_row_list,n_shot)**2)
    return np.array(std)

# |0_L> stabilizer expectations
counts_all_syn_0 = counts_reshape(job_set_syn_0)
g_row_list = g_row(str_g)
matrix_T5 = transform_T5()
# basis_list
expectation_syn_0 = stabilizer_expectation(counts_all_syn_0,matrix_T5,g_row_list,n_shot)
std_syn_0 = stabilizer_std(counts_all_syn_0,matrix_T5,g_row_list,n_shot)
```

- Fidelity calculation

```
def fidelity_projector_circ(coeff,str_g_prod):
    """

    Parameters
    ----------
    coeff : array_like, shape (2, )
        coefficient [a, b] for logical state. |\psi>=a|0_L>+b|1_L>
    """
    circ_proj = []

    for str_gi in str_g_prod:
        # 1. initialize new circuit
        reg_q_temp = QuantumRegister(5,'q')
        circ_r_s = QuantumCircuit(reg_q_temp)
        circ_r_s.initialize(params=coeff,qubits=0)
        # 2. encoding
        encoding_5q_code(circ_r_s,reg_q_temp)
        # 3. apply tomographyical transformation
        circ_r_s.barrier()
        tomo_gi_transform(circ_r_s,str_gi)
        # 4. measure all qubits in |0>, |1>
        circ_r_s.measure_all()

        circ_proj.append(circ_r_s)
    return circ_proj

# reshape counts in job_set
def counts_fidelity(job_set):
    results = job_set.results()
    counts_all = []
    for stabilizer in range(len(str_g_prod)):
        counts_all.append(results.get_counts(stabilizer))
    return np.array(counts_all)

# generate all expectation value for gi from all counts
def stabilizer_expectation_fidelity(counts_all,matrix_T5,g_row_list,n_shot):
    expectation = []
    for stabilizer in range(len(str_g_prod)):
        prob_ijk = T5_counts(matrix_T5[g_row_list[stabilizer]],counts_all[stabilizer])/n_shot
        expectation.append(prob_ijk)
    return np.array(expectation)

# |0_L> stabilizer expectations
counts_all_fid_0 = counts_fidelity(job_set_fid_0)
g_row_list = g_row(str_g_prod)
matrix_T5 = transform_T5()
expectation_fid_0 = stabilizer_expectation_fidelity(counts_all_fid_0,matrix_T5,g_row_list,n_shot)
fidelity_0 = 1/2**4*(1+np.sum(expectation_fid_0))
print('fidelity for |0_L>: {}'.format(fidelity_0))
```