

中原大學資訊工程學系學士論文

隨機性對於量子糾錯算法的重要性

及適用於 IBM Quantum 的量子糾錯算法

作者：曾品元 11027233、江庭瑄 11027235、

王紫薰 11027238

指導教授：黃琮暉教授、張元翔教授

中華民國 113 年 2 月

目錄

中文摘要

Shor's Error Correction Code

第一章 量子運算介紹

1.1 量子運算

1.2 為何需要糾錯

第二章 糾錯演算法的實作與分析

模擬器 CODE

參考文獻

圖目錄

表目錄

中文摘要

隨著古典電腦技術的發展，它已經面臨著瓶頸。在馮紐曼架構的古典電腦體系下，資料儲存技術進展緩慢，致使古典電腦的計算能力難以實現顯著的提升。相較之下，量子電腦雖然仍屬較不成熟的技術，但在處理特定問題的時間複雜度遠優於古典電腦。

古典電腦的位元狀態僅能為1或0，而量子電腦的位元狀態可以同時為1和0。這種獨特的特性使得量子電腦的 n 個bit能夠同時表示 2^n 種狀態，使得在搜尋計算方面，量子電腦的速度遠超古典電腦。然而，正因為這種同時處於0和1的特性，量子電腦的錯誤率相對較高，糾錯變得極具挑戰性，需要採用與以往不同的思維方式來滿足糾錯需求。

Shor's Error Correction Code

Shor 糾錯碼 (Shor's Error Correction Code) 是量子糾錯碼的一種，於 1995 年由美國物理學家彼得·肖爾 (Peter Shor) 提出。其基本原理是將一個 qubit 的資料分佈在多個 qubit 上。

重複碼 (Repetition Code) :

首先，使用重複碼將一個 qubit 的資料擴展到三個 qubit。對於狀態 $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ ，其編碼狀態為 $\alpha|000\rangle + \beta|111\rangle$ 。

相位糾錯碼 (Phase Code) :

接著，對每一個重複碼 (即每一個三 qubit 組)，應用相位糾錯碼。具體操作是對每一個物理 qubit 再編碼為三個 qubit。

最終編碼後的狀態是 9 qubit，每三個 qubit 對應一個相同狀態，表示初始的一個邏輯 qubit。

位元翻轉錯誤 (Bit-flip error) :

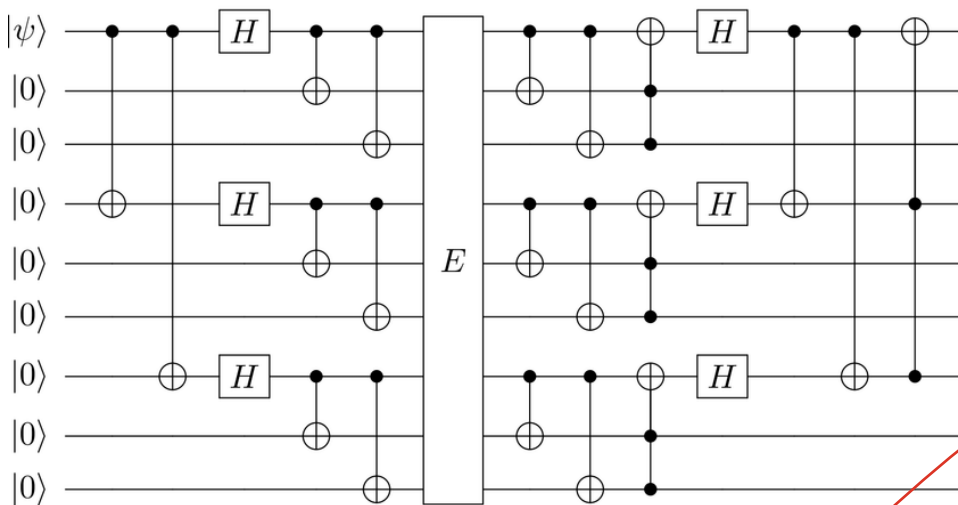
通過對每個三 qubit 組進行多數表決 (majority voting)，可以檢測並糾正比特翻轉錯誤。例如，如果某個三 qubit 組為 $|001\rangle$

，則判斷中間的 qubit 可能發生了 qubit 翻轉錯誤，並糾正回 $|000\rangle$ 。

相位翻轉錯誤 (Phase-flip error) :

通過相位糾錯碼的編碼方式來檢測和糾正相位翻轉錯誤。

Shor 糾錯碼是量子計算領域的基礎技術之一，它將一個量子比特的信息分佈到多個量子 qubit 上，利用冗餘來檢測和糾錯量子計算過程中常見的錯誤。Shor 提供了一種保護量子信息免受外界干擾和內部噪聲的方法，且能同時糾正 qubit 的翻轉和相位翻轉錯誤，這是單獨的重複碼或相位糾錯碼無法實現的，在理論上提供了對量子信息的強大保護，為實現實際可行的量子計算奠定了基礎。



第一章 量子運算介紹

1.1 量子運算

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\frac{\theta}{2}|1\rangle$$

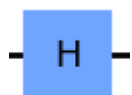
其中 $\cos\frac{\theta}{2}$ 和 $e^{i\phi}\sin\frac{\theta}{2}$ 表示相位，正負為方向，大小為振幅。相位的平方為機率，且和為1。

量子邏輯閘使用么正矩陣表示，而基本狀態 $|0\rangle$ 以 $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ 表示， $|1\rangle$ 以 $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ 表示。常見的量子閘有 CNOT gate、Hadamard gate、Pauli-X gate、Pauli-Y gate、Pauli-Z gate、Toffoli gate：

CNOT gate

為雙量子邏輯閘，有輸入與輸出，其中輸入的位元分別稱作控制位元(control qubit)及目標位元(target qubit)。若控制位元為1則目標位元進行反閘(X閘)運算，反之則目標位元不做任何運算。類似古典位元的 XOR 閘。

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$



Hadamard gate

是基本量子邏輯閘之一，將基本狀態 $|0\rangle$ 變成 $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ ，並將 $|1\rangle$ 變成 $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$ 。

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle)$$

$$H|1\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{1}{\sqrt{2}} (|0\rangle - |1\rangle)$$



Pauli-X gate(NOT)

繞 Bloch 球體的 x 軸旋轉 180 度。

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

$$X|1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$$



Pauli-Y gate

繞 Bloch 球體的 y 軸旋轉 180 度。

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

$$Y|0\rangle = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ i \end{bmatrix} = i|1\rangle$$

$$Y|1\rangle = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -i \\ 0 \end{bmatrix} = -i|0\rangle$$

Pauli-Z gate



繞 Bloch 球體的 z 軸旋轉 180 度。

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$Z|0\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$$

$$Z|1\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = -\begin{bmatrix} 0 \\ 1 \end{bmatrix} = -|1\rangle$$

Toffoli gate (CCX gate)

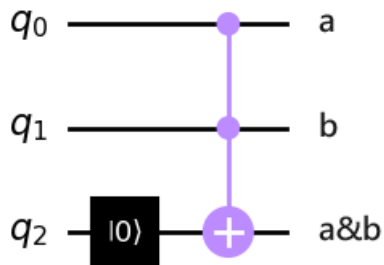
操作三個量子位元的閘, 有 2-qubit 輸入與 1-qubit 輸出。若前兩個 qubit 皆為 $|0\rangle$ 則對第三個 qubit 做 Pauli-X gate 運算, 否則不動作。

$$\text{CCNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

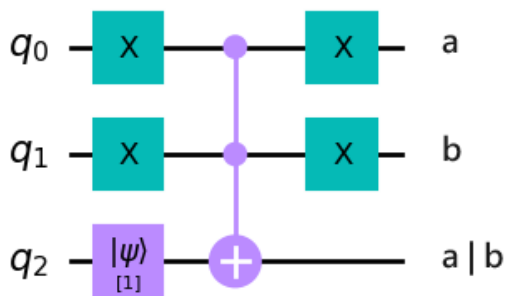


用 CNOT gate 實作量子位元的 AND 閘和 OR 閘：

Quantum AND gate



Quantum OR Gate



1.2 為何需要糾錯

量子錯誤有很多種，以下為常見的量子錯誤：

位元翻轉錯誤 (Bit Flip Errors)： 這種錯誤發生在量子位元 (qubit) 在計算過程中由於外部干擾或其他原因而由 0 翻轉為 1，或由 1 翻轉為 0。

隨機相位錯誤 (Phase Flip Errors)： 這種錯誤涉及到量子位元的相位信息，其中量子位元可能在計算過程中受到外部因素影響，導致其相位發生錯誤。

閘操作錯誤 (Gate Operation Errors)： 量子電腦使用閘操作來執行計算，這些操作可能受到外部環境的干擾，或者在實際實現中可能存在一些不完美性能，導致閘操作錯誤。

量子比特間交互錯誤 (Qubit Crosstalk Errors): 不同量子位元之間的相互作用可能導致它們之間的信息傳遞錯誤，這可能發生在相鄰的量子位元之間。

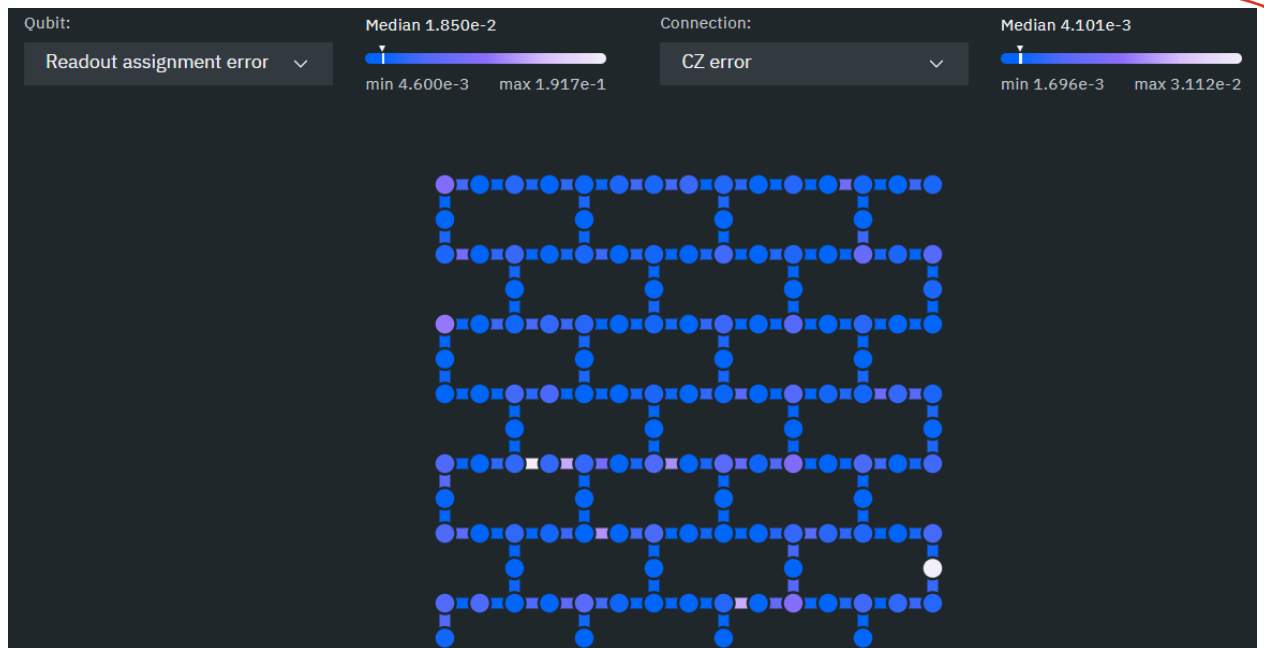
相位糾正錯誤 (Phase Correction Errors): 這種錯誤發生在嘗試糾正相位錯誤時出現的錯誤，可能導致進一步的計算錯誤。

連結錯誤 (Connectivity Errors): 由於量子位元之間的連結關係可能受到局限，連結錯誤可能會發生在計算過程中，影響計算的正確執行。

計算時序錯誤 (Timing Errors): 這種錯誤可能由於計算過程中的計時不準確，導致門操作和量子位元的不正確同步。

測量錯誤 (Measurement Errors): 在量子計算結束時，測量操作可能受到干擾，導致計算結果的錯誤。

這是目前 IBM Heron r1 processor 的 error rate map



From:

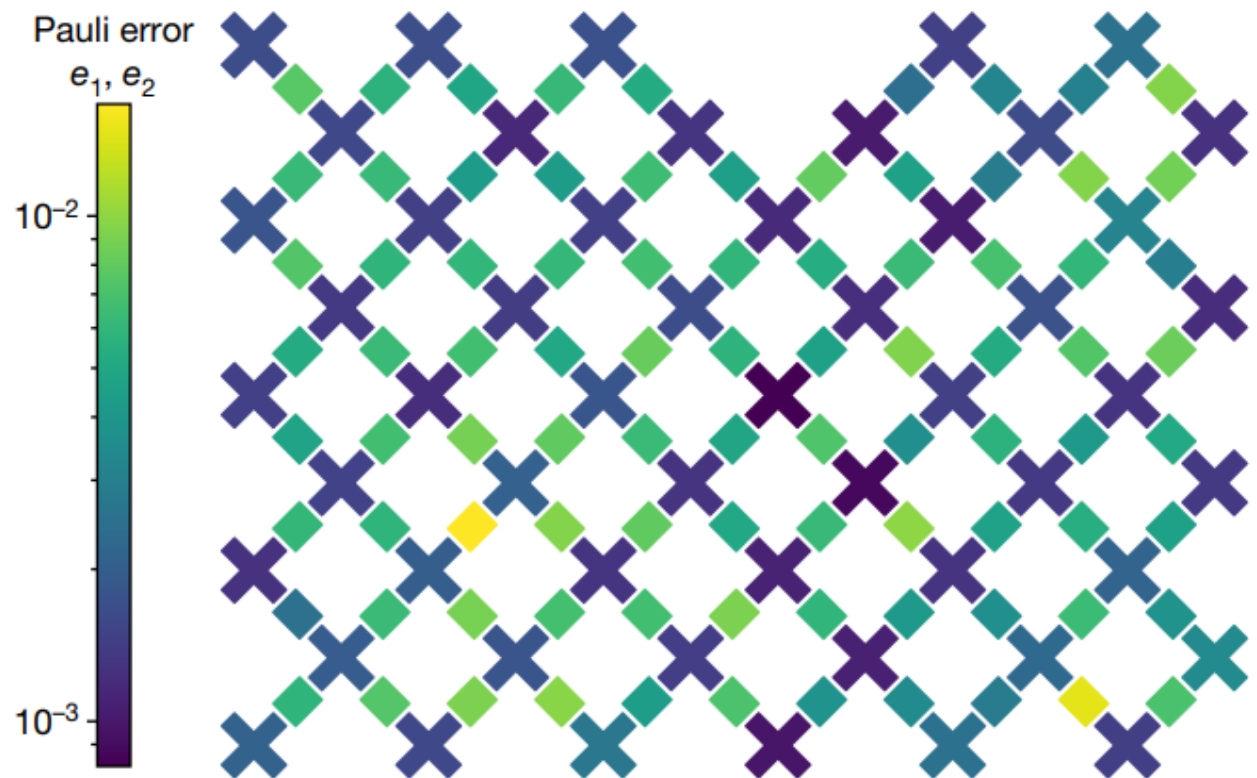
https://quantum.ibm.com/services/resources?tab=systems&system=ibm_torino

CZgate 是這個 quantum processor 主要使用的 two-qubit gate，錯誤率的中位數是 0.004101，由於量子錯誤不僅僅只有 bit-flip error，還有 phase error，這 0.004101 的錯誤率在經過幾十次以上的運算後就會大到無法得到正確答案。所以量子糾錯演算法對於量子電腦的成功運行是非常重要的。量子糾錯算法不

僅要考慮糾錯之後的正確率，糾錯所用到的 two-qubit gate 還要符合量子電腦 qubit 網格的排列方式。

量子糾錯算法的設計不僅僅於程式方面，一個大型的量子電腦系統在開發時也要考慮如何優化量子電路布局使量子糾錯算法更容易實作。

附圖 Google Sycamore processor



Form: <https://blog.research.google/2019/10/quantum-supremacy-using-programmable.html>

第二章 糾錯演算法的實作與分析

three-bit repetition code

將一個待糾錯的 qubit 運算重複三次，理應獲得三個一模一樣的結果，然而運算過程可能會出錯，導致三個 qubit 不全相同。three-bit repetition code 即為將這三個 qubit 統一更正為佔多數的答案，意即將此三 qubit 放入全加器並取 carry 位。

three-bit repetition code 運作如下範例：

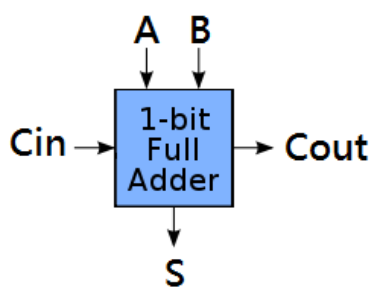
1 → 111

0 → 000

000, 001, 010, 100 → 0

111, 110, 101, 011 → 1

下圖為傳統電路的全加器與真值表

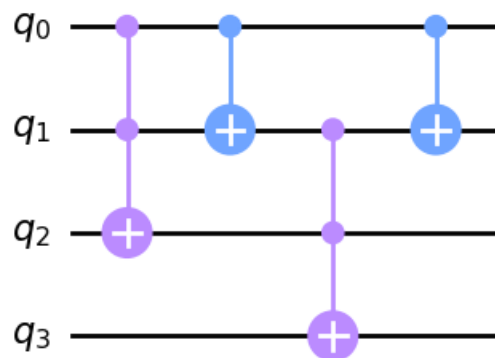


Input			Output	
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

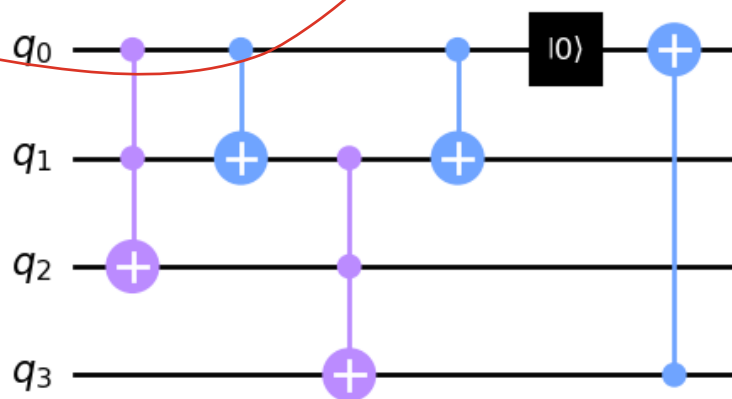
量子電路實作全加器

input : q_0, q_1, q_2

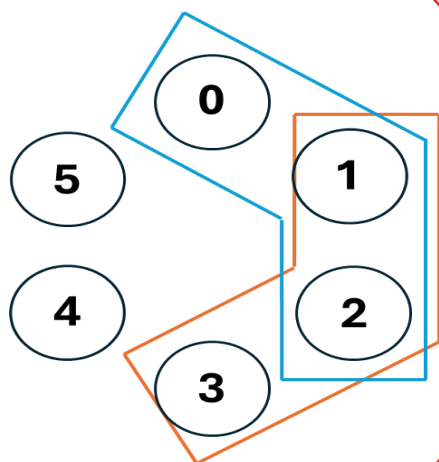
output : q_3 (carry)



再將結果寫回 q_0 :



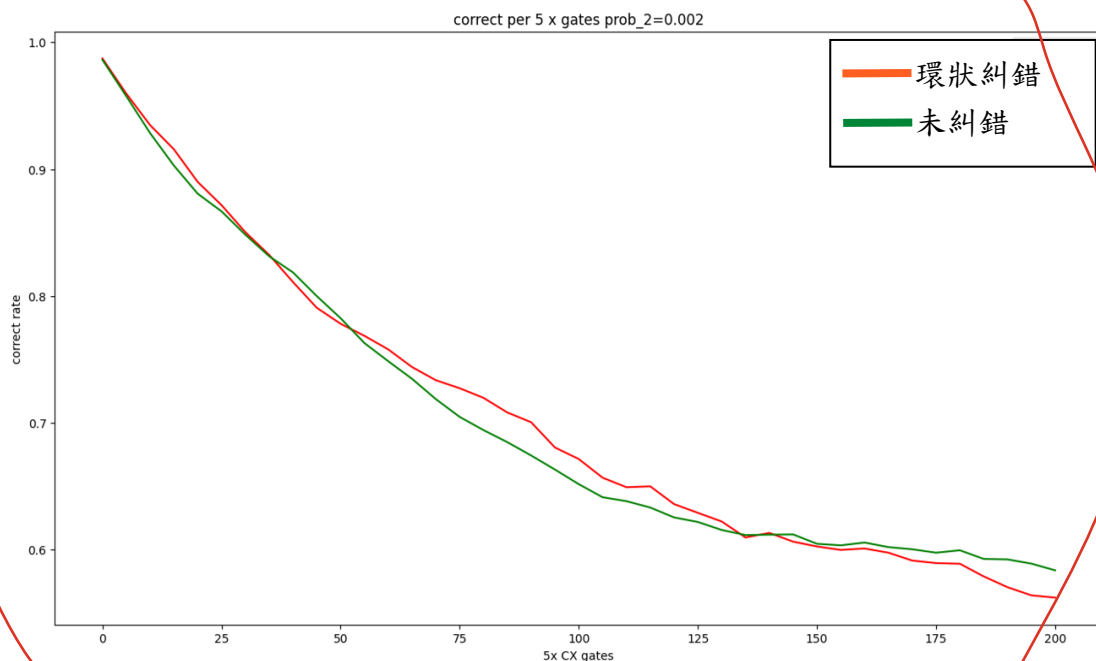
環狀分組糾錯



beauty

將一個待糾正的 qubit 運算進行六次，並運用六個結果循環分組進行糾錯。依序將 012, 123, 234, 345, 450, 501 分為一組形成 three-bit repetition code 並執行糾錯。

6-qubit 環狀糾錯成果



成效不彰，有時正確率甚至不如糾錯前，如上圖。

錯誤情形

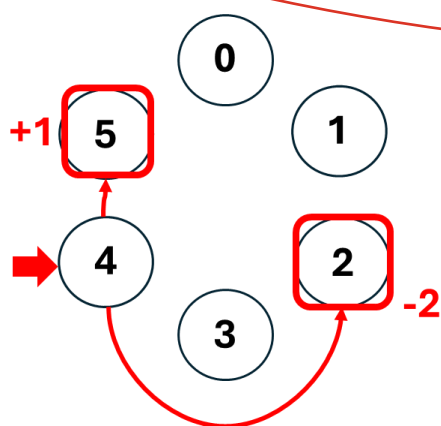
000101 \rightarrow 000000

000110 \rightarrow 000110 (Error)

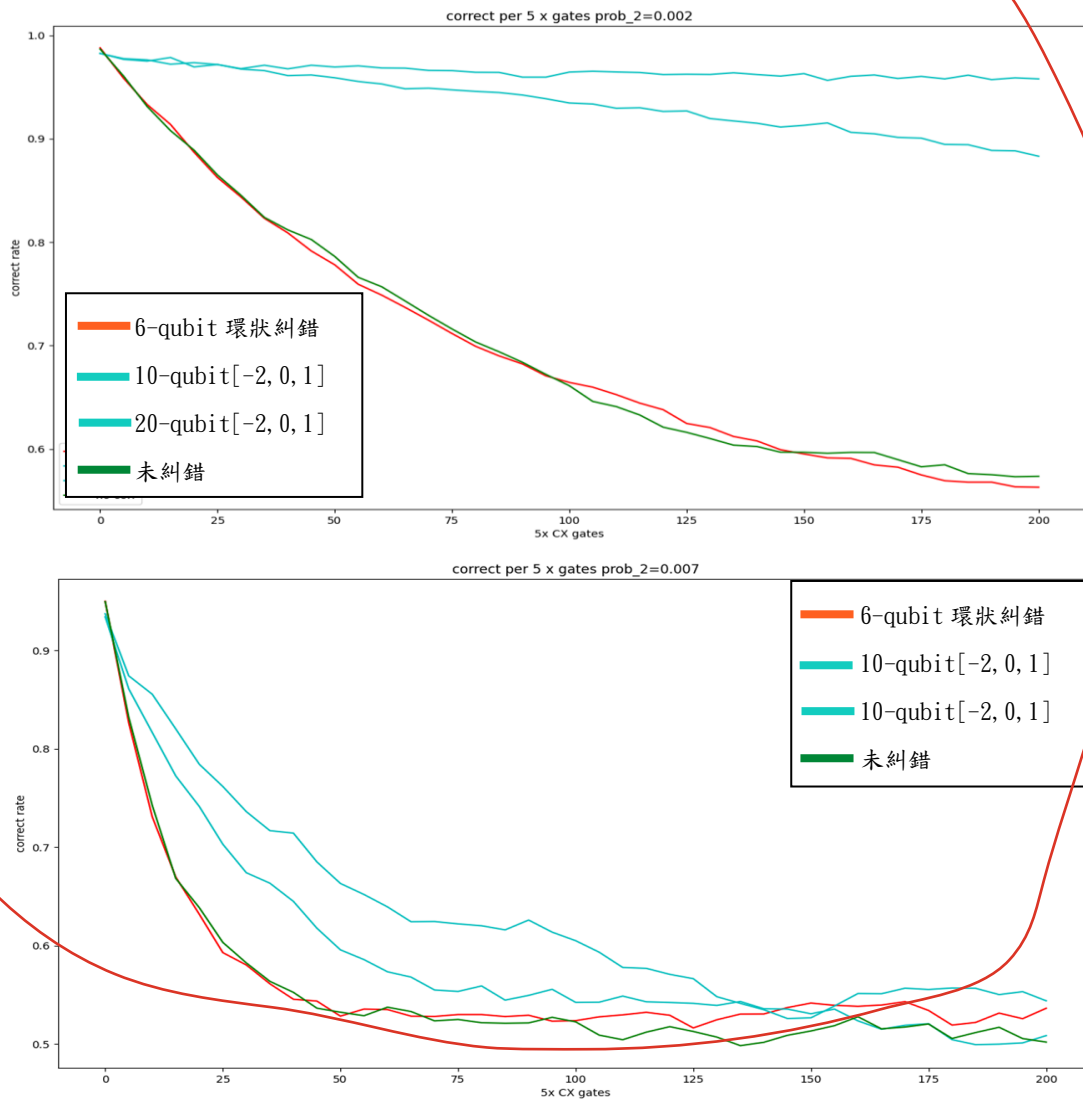
假設目的是將所有 qubit 糾正為 0，則當遇到連續兩個 1 時會發生死循環，無論糾錯幾次都無法修正為 0。意即，只要有連續的錯誤就是就會沒辦法修正，連續兩個 qubit 同時壞掉是很容易發生的，尤其 cx-gate(或其他雙 qubit gate)。為解開死循環，須將取樣順序打亂。

如何取樣

經實驗發現，依照 Index： $[-2, 0, 1]$ 取樣的糾錯效果最佳。假設欲修正 index 為 4 的 qubit，則取 index 為 $2(4-2)$ 、 $4(4+0)$ 與 $5(4+1)$ 的 qubit 組成 three-bit repetition code，獲得糾錯結果後寫回 index 為 4 的 qubit。



index	0	1	2	3	4	5
	0	0	0	1	1	0
\rightarrow	0	0	0	1	0	0



量子電腦的錯誤率很關鍵

以下兩圖分別展示當量子電腦本身的錯誤率為 0.002 以及 0.007 時的糾錯成果，可見當量子電腦本身的錯誤率為 0.007 時效果遠不如錯誤率為 0.002 的量子電腦。由此可知，量子電腦本身的錯誤率很大程度的影響糾錯演算法成功率。

新的死循環

假設欲將 6-qubit 以 $\text{index}=[-2, 0, 1]$ 取樣糾錯以修正為全 0，而當錯誤情形如下時，越是糾錯會越錯越多。

100001

->100001

->110001

->111001 ...

原因是若第一組 three bit repetition code 糾錯出來的結果仍是錯的，寫回第 0-qubit 後，會在被第二組 three bit repetition code 取樣進行糾錯。這顯然將導致一步錯，步步錯的結果。

解決方法

更好地辦法應是，將每一組 three bit repetition code 糾錯出來的結果先暫存至另一組 6-qubit 相對應 index 的 qubit 中，取樣下一組 three bit repetition code 時取的都是原始未糾錯過過的，以避免每次糾錯互相影響。待所有糾錯都完成，再將結果寫回原本的 q0。

data qubit: 100001

support qubit:000000 calculate->

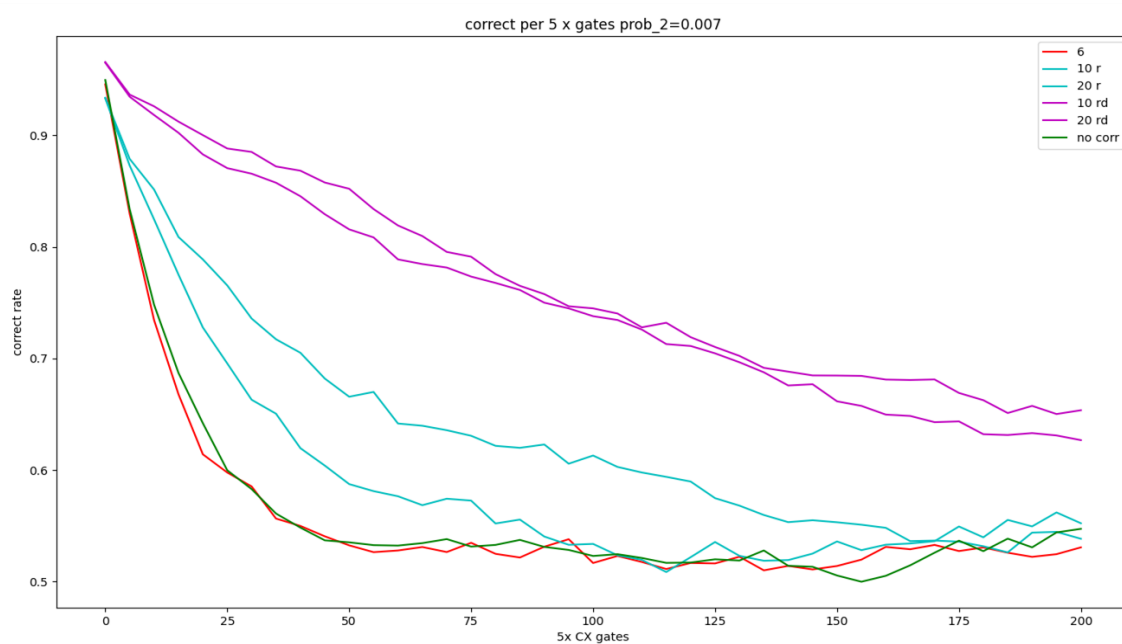
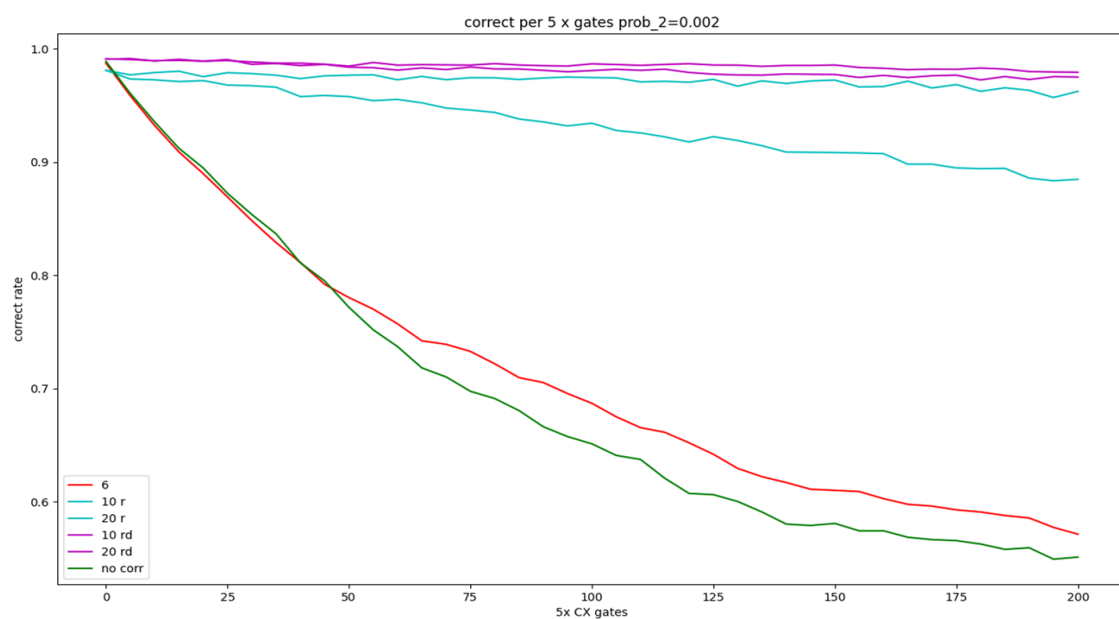
data qubit: 100001

support qubit:000001 copy(write back)

data qubit: 000001

support qubit:000001

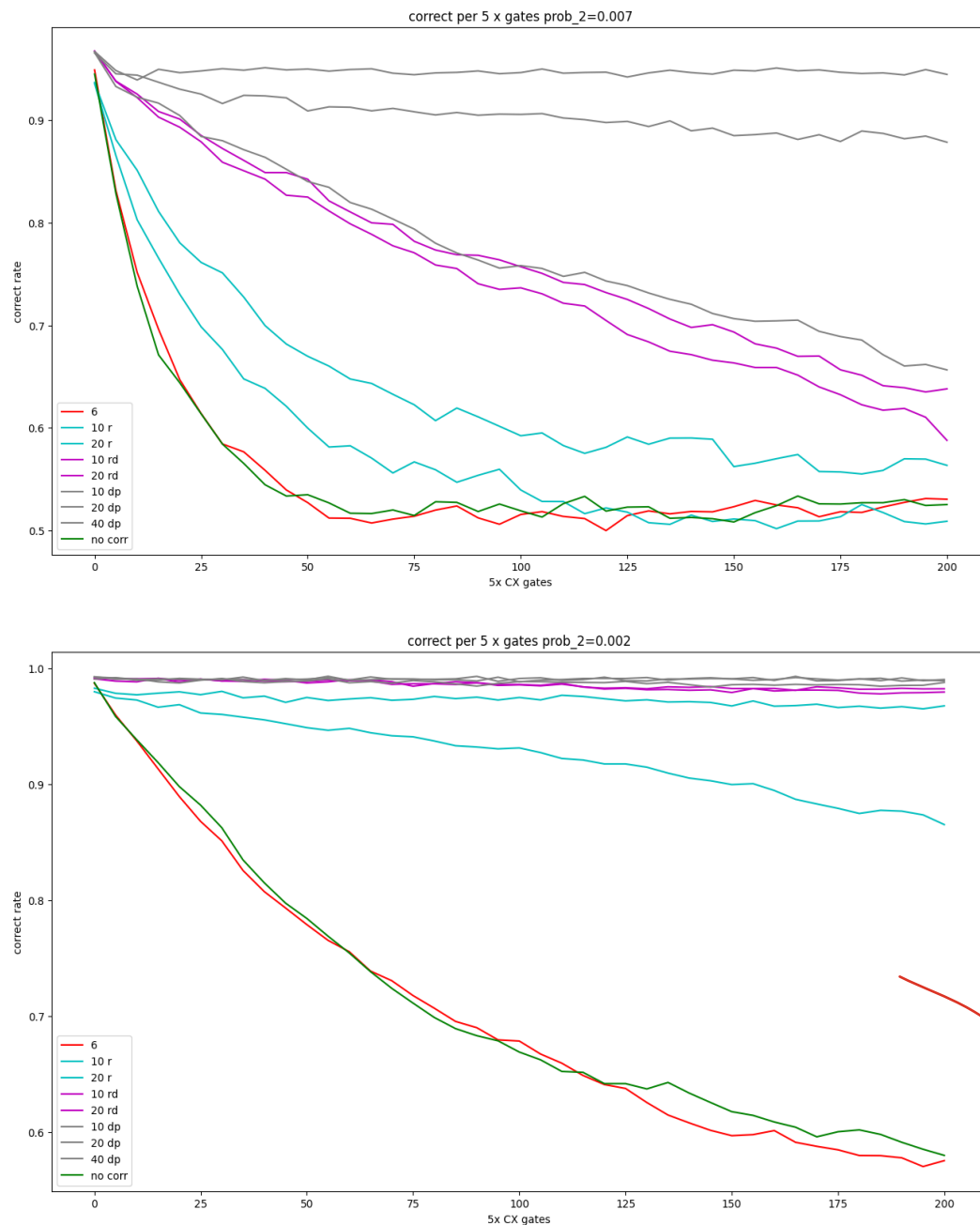
以下兩圖分別為錯率 0.002 與 0.007 的糾錯結果：



可以發現錯誤率稍微變大，糾錯率就大幅降低。

隨機打亂

同樣錯誤率為 0.002 與 0.007 的糾錯結果，增加了另一種作法的觀察——模擬 qubits 隨機重新排列(灰色)。

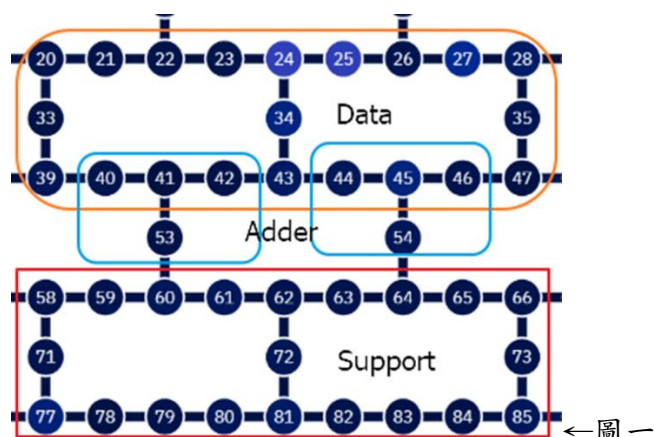


雖然現實中不存在這種隨機打亂的方法，但可以觀察「隨機打亂」是否有助於大幅提升糾錯能力，結果發現確實是有助於的。

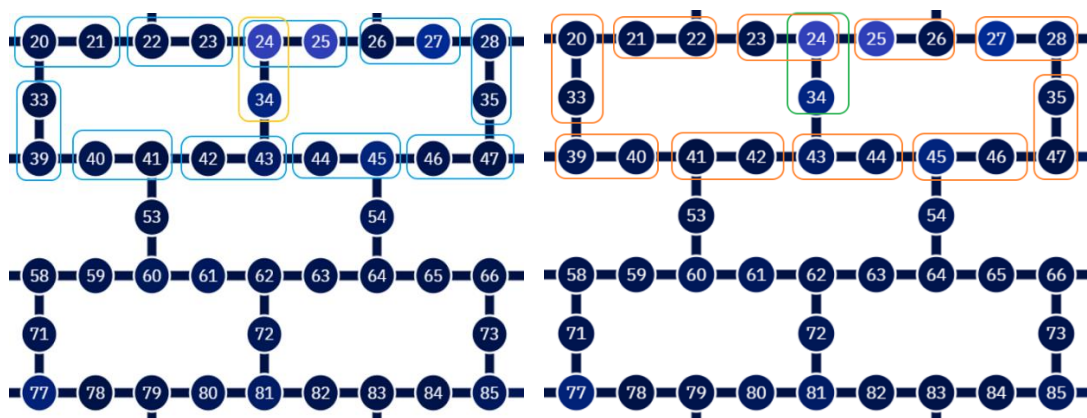
->前面說的兩個問題()都很重要，且我們需要一個(接近)隨機打亂的方法，在每次糾錯前執行。

理想中的隨機打亂??

下圖以 IBM brisbane 的量子電腦的 qubit 排列方式為例，示範隨機打亂的步驟：



圖一：橘色框內的 qubit 為待糾錯的資料，紅色框起的 qubit 為欲儲存糾錯結果的額外 qubit 空間。圖中橘框內編號 40、41、42 為相鄰 qubit，首先取這三個 qubit 進行糾錯(放進加法器取 carry 位)，將結果存放到編號 60 的 qubit。同理，取編號 44、45、46 進行糾錯並將結果存放到編號 64 的 qubit。



圖二 ↑

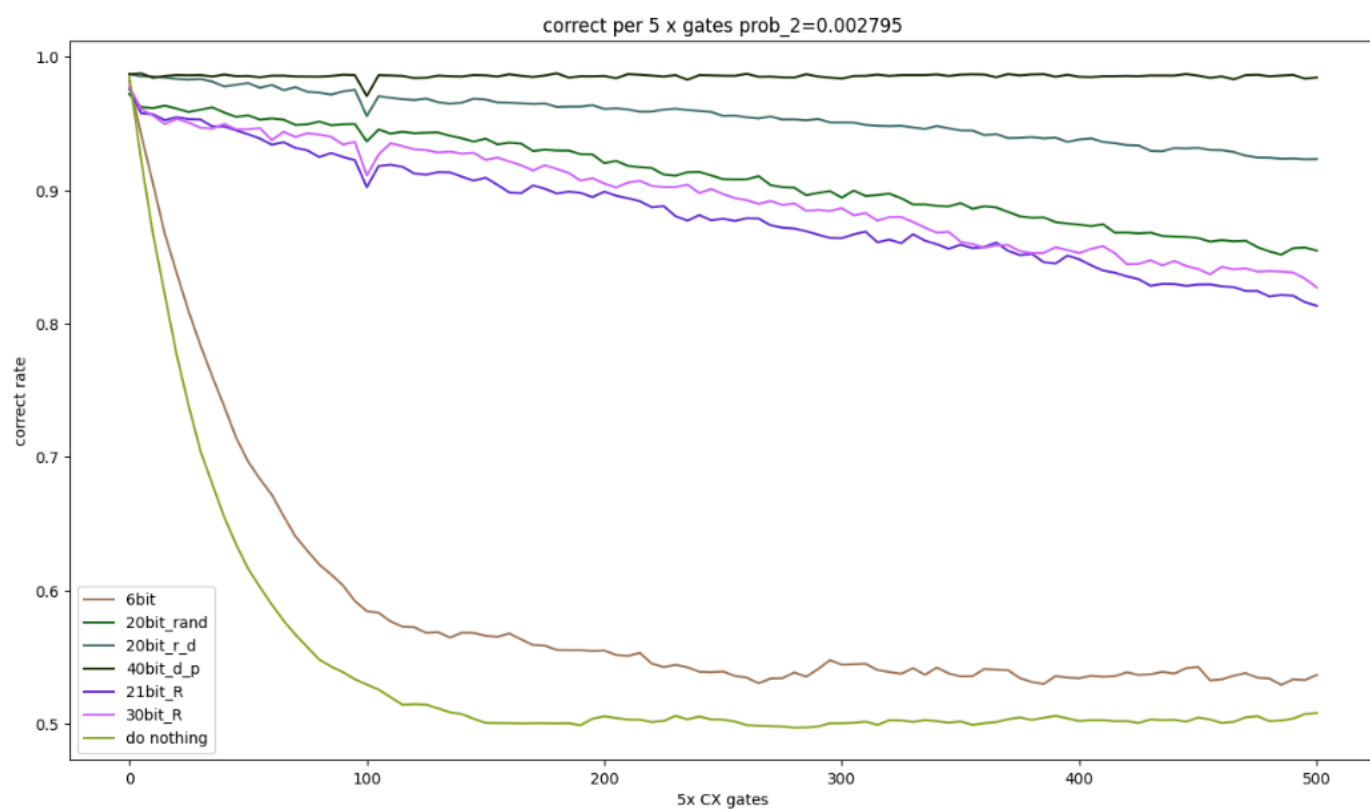
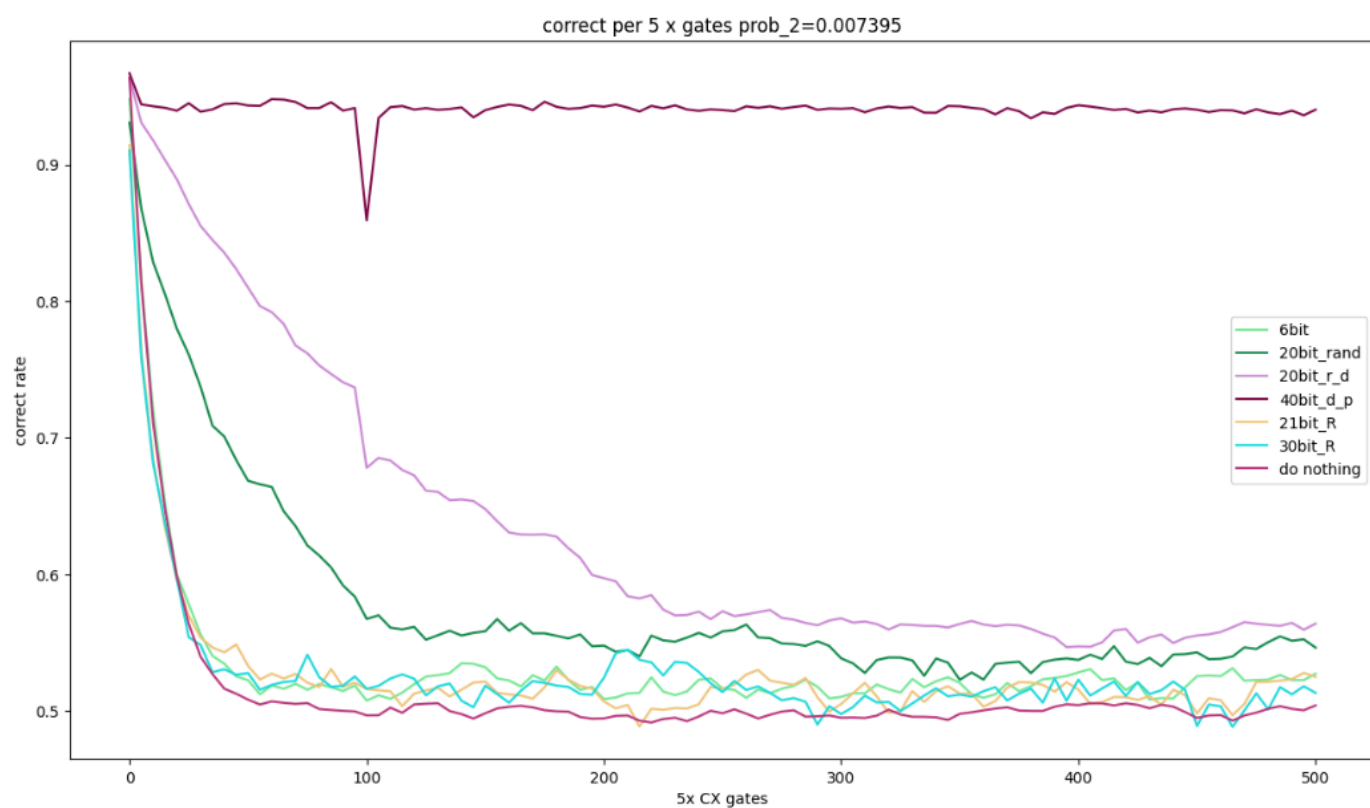
圖三 ↑

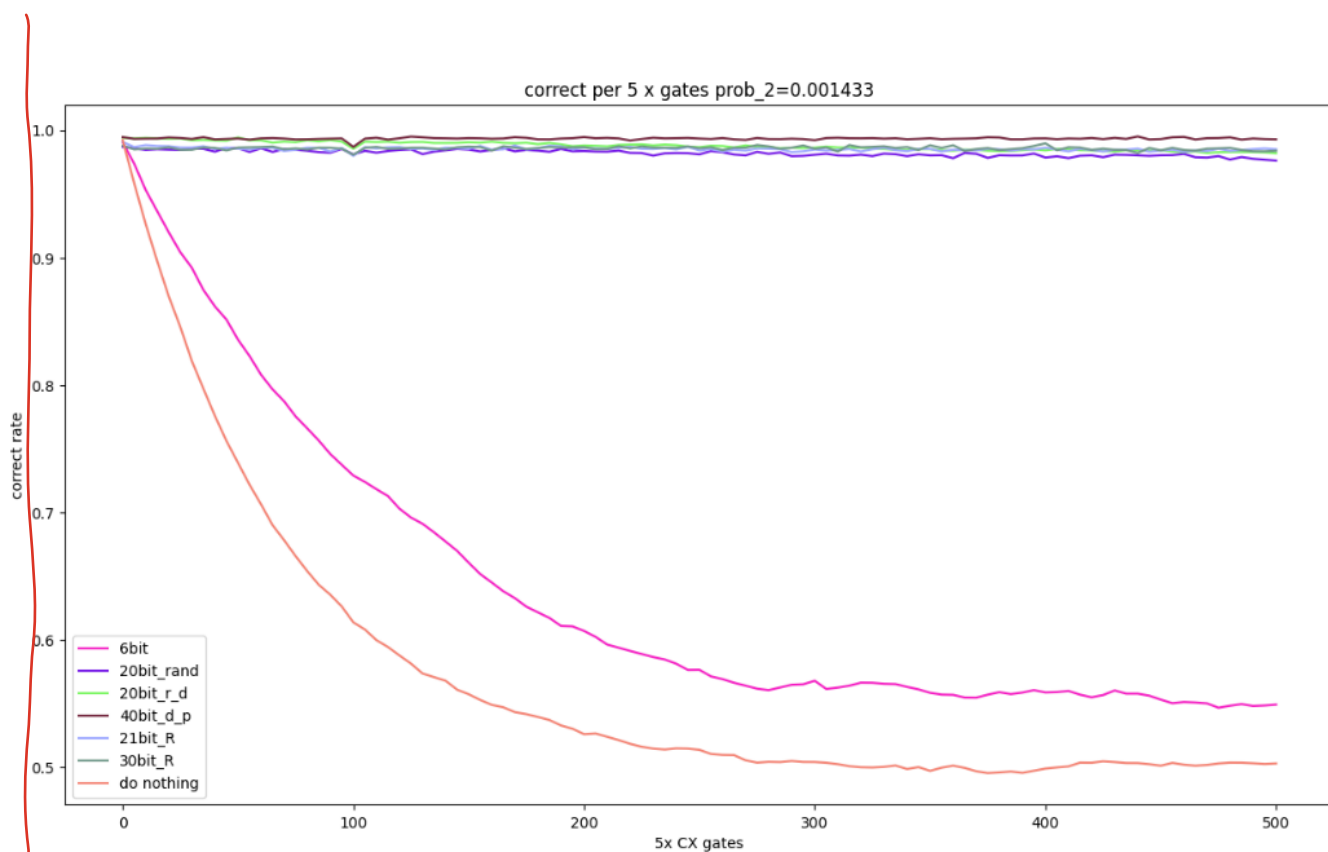
圖二、圖三：編號 53、54 已存放糾錯後的結果後，將待糾錯資料的 qubit 兩兩位置交換，交換方式如圖二的藍色框線所示，完成以後再如圖三的橘線所示兩兩互換位置，即完成一次打亂。打亂後得到新的編號 40、41、42 的 qubit，取此三個 qubit 進行糾錯得結果存入編號 60 的 qubit。以次類推，每次打亂後取編號 40、41、42 的 qubit 糾錯，將結果依序填入下方儲存空間，直至空間填滿，即獲得所有糾錯結果。

每次打亂後取的 qubit 情形舉例：

[(1, 3, 25), (1, 5, 7), (1, 23, 27), (2, 4, 29), (3, 5, 27), (3, 7, 29), (4, 6, 7), (5, 6, 29), (8, 9, 13), (8, 10, 11), (8, 12, 14), (8, 20, 28), (9, 10, 12), (9, 18, 20), (10, 12, 17), (10, 14, 16), (10, 21, 28), (11, 16, 18), (12, 14, 15), (12, 16, 18), (12, 19, 21), (13, 14, 16), (14, 17, 19), (14, 18, 20), (15, 16, 17), (16, 20, 28), (18, 21, 28), (19, 20, 21), (22, 25, 27), (23, 24, 25)]

不同錯誤率下的糾錯率





代號解釋：

6 bit：最基礎的 6bit 環狀糾錯

20bit_rand：取樣 index 為 $[-1, 0, 2]$ 的方法

20bit_r_d：取樣 index 為 $[-1, 0, 2]$ ，且將結果複製出來再寫回，而不是使用糾錯結果對後續 qubit 糾錯

40bit_d_p：算完隨機打亂再寫回

21bit_R：使用我們提出的方法在 IBM 量子電路上模擬

30bit_R：使用我們提出的方法在 IBM 量子電路上模擬

如何處理相位的問題：

可以將要修正的相位先旋轉映射到 x 軸上，糾錯算法完成後再返向旋轉回去，以 $+H$ 、 $-H$ 為例， 0 、 1 和 $+H$ 、 $-H$ 互為經過 Hadamard gate 的結果，若要做 $+H$ 、 $-H$ 的修正可以先用 Hadamard gate 轉變為 0 、 1 的狀態，修正完成後再利用 Hadamard gate 轉回為 $+H$ 、 $-H$ 。

取錯誤率的理由：

我們使用三個錯誤率做實驗，分別為 0.007395 、 0.002795 、 0.001433 ，分別為 `ibm_Eagle r3` 的 Median ECR error、`ibm_Egret` 的 Median ECR error 和 `ibm_Egret` 的 Minima ECR error，第一個是我們用來參考現在(2024/2)能用到的量子電腦平均錯誤率，第二個跟第三個是 IBM 開發中的量子電腦錯誤率，我們想要用此推估這個糾錯演算法在未來的淺力。

模擬器 CODE

optional

~\Documents\學校\Project_QEC\adder\booltest.py

```
1  # %%
2  import numpy as np
3  from statistics import median
4  import matplotlib.pyplot as plt
5  from tqdm import tqdm
6  from multiprocessing import Pool
7
8
9  class testresult:
10     def __init__(self, testname, measurex, result, yerr):
11         self.testname = testname # "6bit"
12         self.measurex = measurex # [0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
13         self.result = result # [0.9,0.8,0.7,0.6,0.5,0.4,0.3,0.2,0.1,0.0]
14         self.yerr = yerr # [0.01,0.01,0.01,0.01,0.01,0.01,0.01,0.01,0.01,0.01]
15
16     def addplt(self):
17         # random color
18         col = (np.random.random(), np.random.random(), np.random.random())
19         """
20         plt.errorbar(
21             self.measurex,
22             self.result,
23             yerr=[i * 1 for i in self.yerr],
24             color=col,
25             label=self.testname,
26             fmt="k",
27         )
28         """
29
30         plt.plot(
31             self.measurex,
32             self.result,
33             color=col,
34             label=self.testname,
35         )
36
37     def __str__(self):
38         return f"testname:{self.testname}\nresult:{self.result}\nyerr:{self.yerr}"
39
40
41  class boolcirc:
42     def __init__(self, qubit, errorrate):
43         # self.qubit = np.array([False] * qubit, dtype=np.bool_)
44         self.qubit = [False] * qubit
45         self.prob = errorrate
46
47     def checkerror(self, i, rate):
48         if np.random.rand() < self.prob * rate:
49             self.qubit[i] = not self.qubit[i]
50
51     def checkerror_both(self, i, j, rate):
52         if np.random.rand() < self.prob * rate:
53             self.qubit[i] = not self.qubit[i]
54             self.qubit[j] = not self.qubit[j]
55
56     def checkerror_ecr(self, i, j):
57         self.checkerror(i, 0.25)
```



```

58         self.checkerror(j, 0.25)
59         self.checkerror_both(i, j, 0.25)
60
61     def reset(self, a):
62         self.qubit[a] = False
63
64     def x(self, a):
65         self.qubit[a] = not self.qubit[a]
66
67     def cx(self, a, b):
68         if self.qubit[a]:
69             self.qubit[b] = not self.qubit[b]
70         self.checkerror_ecr(a, b)
71
72     def ccx(self, c1, c2, t):
73         beginstate = self.qubit[c1] ^ self.qubit[c2]
74
75         self.checkerror_ecr(c1, t)
76         self.checkerror_ecr(c2, t)
77         self.checkerror_ecr(c1, t)
78         self.checkerror_ecr(c2, t)
79
80         if self.qubit[c1] and self.qubit[c2]:
81             self.qubit[t] = not self.qubit[t]
82
83         self.qubit[t] ^= beginstate ^ self.qubit[c1] ^ self.qubit[c2]
84
85     def swap(self, a, b):
86         self.cx(a, b)
87         self.cx(b, a)
88         self.cx(a, b)
89
90     def swap_direct(self, a, b):
91         self.qubit[a], self.qubit[b] = self.qubit[b], self.qubit[a]
92
93     def permutation(self, a, b):
94         self.qubit[a:b] = np.random.permutation(self.qubit[a:b])
95
96     def count(self, f, e, value):
97         count = 0
98         for i in range(f, e):
99             if self.qubit[i] == value:
100                 count += 1
101         return count
102
103     def __str__(self):
104         s = ""
105         for i in self.qubit:
106             if i:
107                 s += "1"
108             else:
109                 s += "0"
110         return s
111
112
113 # %% Make a circuit
114 def fulladder(circuit, in1, in2, in3, carry):
115     """in1, in2, in3 are the input qubits, carry is the output qubit,!!no sum"""
116     circuit.ccx(in1, in2, carry)
117     circuit.cx(in1, in2)

```

```

118     circuit.ccx(in2, in3, carry)
119     circuit.cx(in1, in2)
120
121
122 def testcircle(numofqubit, totaltime, coorrate, makeerror, measurerate, shots, prob_2):
123     counts = np.empty([(totaltime // measurerate + 1), shots // numofqubit])
124     for _ in tqdm(range(shots // numofqubit)):
125         circ = boolcirc(numofqubit + 1, prob_2)
126         for k in range(totaltime + 1):
127             if k == 100:
128                 for i in range(numofqubit):
129                     circ.checkerror(i, 20)
130             if makeerror > 0:
131                 for i in range(numofqubit):
132                     circ.checkerror(i, makeerror)
133             if k % coorrate == coorrate - 1:
134                 for i in range(numofqubit):
135                     fulladder(
136                         circ, (i - 1) % numofqubit, i, (i + 1) % numofqubit, numofqubit
137                     )
138                     circ.reset(i)
139                     circ.cx(numofqubit, i)
140                     circ.reset(numofqubit)
141             if k % measurerate == 0:
142                 counts[k // measurerate][_] = (
143                     circ.count(0, numofqubit, False) / numofqubit
144                 )
145     return testresult(
146         f"{numofqubit}bit" if coorrate < 10000 else "do nothing",
147         [i for i in range(0, totaltime + 1, measurerate)],
148         [np.average(i) for i in counts],
149         [np.std(i) for i in counts],
150     )
151
152
153 def testcircle_rand(
154     numofqubit, totaltime, coorrate, makeerror, measurerate, shots, prob_2
155 ):
156     counts = np.empty([(totaltime // measurerate + 1), shots // numofqubit])
157     for _ in tqdm(range(shots // numofqubit)):
158         circ = boolcirc(numofqubit + 1, prob_2)
159         count = 0
160         for k in range(totaltime + 1):
161             if k == 100:
162                 for i in range(numofqubit):
163                     circ.checkerror(i, 20)
164             if makeerror > 0:
165                 for i in range(numofqubit):
166                     circ.checkerror(i, makeerror)
167             if k % coorrate == coorrate - 1:
168                 for i in range(numofqubit):
169                     fulladder(
170                         circ,
171                         (i - 1 - count % 2) % numofqubit,
172                         i,
173                         (i + 1 + (count + 1) % 2) % numofqubit,
174                         numofqubit,
175                     )
176                     circ.reset(i)
177                     circ.cx(numofqubit, i)

```

```

178         circ.reset(numofqubit)
179         for i in range(0, numofqubit, 2):
180             circ.swap(i, (i + 1) % numofqubit)
181         for i in range(1, numofqubit, 2):
182             circ.swap(i, (i + 1) % numofqubit)
183         if k % measurerate == 0:
184             counts[k // measurerate][_] = (
185                 circ.count(0, numofqubit, False) / numofqubit
186             )
187     return testresult(
188         f"{numofqubit}bit_rand",
189         [i for i in range(0, totaltime + 1, measurerate)],
190         [np.average(i) for i in counts],
191         [np.std(i) for i in counts],
192     )
193
194
195 def testcircle_r_d(
196     numofqubit, totaltime, coorrate, makeerror, measurerate, shots, prob_2
197 ):
198     counts = np.empty([(totaltime // measurerate + 1), shots // numofqubit])
199     for _ in tqdm(range(shots // numofqubit)):
200         circ = boolcirc(numofqubit * 2, prob_2)
201         count = 0
202         for k in range(totaltime + 1):
203             if k == 100:
204                 for i in range(numofqubit):
205                     circ.checkerror(i, 20)
206             if makeerror > 0:
207                 for i in range(numofqubit):
208                     circ.checkerror(i, makeerror)
209             if k % coorrate == coorrate - 1:
210                 for i in range(numofqubit):
211                     circ.reset(i + numofqubit)
212                     fulladder(
213                         circ,
214                         (i - 1 - count % 2) % numofqubit,
215                         i,
216                         (i + 1 + (count + 1) % 2) % numofqubit,
217                         i + numofqubit,
218                     )
219                 for i in range(0, numofqubit, 2):
220                     circ.swap(i, (i + 1) % numofqubit)
221                 for i in range(1, numofqubit, 2):
222                     circ.swap(i, (i + 1) % numofqubit)
223                 for i in range(numofqubit):
224                     circ.swap_direct(i + numofqubit, i)
225             if k % measurerate == 0:
226                 counts[k // measurerate][_] = (
227                     circ.count(0, numofqubit, False) / numofqubit
228                 )
229     return testresult(
230         f"{numofqubit}bit_r_d",
231         [i for i in range(0, totaltime + 1, measurerate)],
232         [np.average(i) for i in counts],
233         [np.std(i) for i in counts],
234     )
235
236
237 def testcircle_d_p(

```

```

238     numofqubit, totaltime, coorrate, makeerror, measurerate, shots, prob_2
239 ):
240     counts = np.empty([(totaltime // measurerate + 1), shots // numofqubit])
241     for _ in tqdm(range(shots // numofqubit)):
242         circ = boolcirc(numofqubit * 2, prob_2)
243         for k in range(totaltime + 1):
244             if k == 100:
245                 for i in range(numofqubit):
246                     circ.checkerror(i, 20)
247             if makeerror > 0:
248                 for i in range(numofqubit):
249                     circ.checkerror(i, makeerror)
250             if k % coorrate == coorrate - 1:
251                 for i in range(numofqubit):
252                     circ.reset(i + numofqubit)
253                     fulladder(
254                         circ,
255                         (i - 1) % numofqubit,
256                         i,
257                         (i + 1) % numofqubit,
258                         i + numofqubit,
259                     )
260                 for i in range(numofqubit):
261                     circ.swap_direct(i + numofqubit, i)
262                 circ.permutation(0, numofqubit)
263             if k % measurerate == 0:
264                 counts[k // measurerate][_] = (
265                     circ.count(0, numofqubit, False) / numofqubit
266                 )
267     return testresult(
268         f"{numofqubit}bit_d_p",
269         [i for i in range(0, totaltime + 1, measurerate)],
270         [np.average(i) for i in counts],
271         [np.std(i) for i in counts],
272     )
273
274
275 def testcircle_R21(totaltime, makeerror, measurerate, shots, prob_2):
276     """
277     0 1 2 3 4 5 6 7 8
278     19          20      9
279     18 17 16 15 14 13 12 11 10
280     """
281     counts = np.empty([(totaltime // measurerate + 1), shots // 21])
282     for _ in tqdm(range(shots // 21)):
283         circ = boolcirc(21 * 2, prob_2)
284         for k in range(totaltime + 1):
285             if k == 100:
286                 for i in range(21):
287                     circ.checkerror(i, 20)
288             if makeerror > 0:
289                 for i in range(21):
290                     circ.checkerror(i, makeerror)
291             for i in range(10):
292                 circ.reset(i * 2 + 21)
293                 fulladder(
294                     circ,
295                     15,
296                     16,
297                     17,

```

```

298         i * 2 + 21,
299     )
300     circ.reset(i * 2 + 1 + 21)
301     fulladder(
302         circ,
303         11,
304         12,
305         13,
306         i * 2 + 1 + 21,
307     )
308     for j in range(0, 20, 2):
309         circ.swap(j, j + 1)
310     circ.swap(20, 14)
311     for j in range(1, 20, 2):
312         circ.swap(j, (j + 1) % 20)
313     circ.swap(20, 14)
314     circ.reset(41)
315     fulladder(
316         circ,
317         15,
318         16,
319         17,
320         41,
321     )
322
323     for i in range(21):
324         circ.swap_direct(i + 21, i)
325         circ.checkerror(i + 21, prob_2 * 3)
326     # circ.permutation(0, 21)
327     if k % measurerate == 0:
328         counts[k // measurerate][_] = circ.count(0, 21, False) / 21
329     return testresult(
330         f"{21}bit_R",
331         [i for i in range(0, totaltime + 1, measurerate)],
332         [np.average(i) for i in counts],
333         [np.std(i) for i in counts],
334     )
335
336
337 def testcircle_R30(totaltime, makeerror, measurerate, shots, prob_2):
338     """
339     0  1  2  3  4  5  6  7  8  9 10 11 12
340    27      28      29      13
341    26 25 24 23 22 21 20 19 18 17 16 15 14
342     """
343     counts = np.empty([(totaltime // measurerate + 1), shots // 30])
344     for _ in tqdm(range(shots // 30)):
345         circ = boolcirc(30 * 2, prob_2)
346         for k in range(totaltime + 1):
347             if k == 100:
348                 for i in range(30):
349                     circ.checkerror(i, 20)
350             if makeerror > 0:
351                 for i in range(30):
352                     circ.checkerror(i, makeerror)
353             for i in range(10):
354                 circ.reset(i * 3 + 30)
355             fulladder(
356                 circ,
357                 15,

```

```

358         16,
359         17,
360         i * 3 + 30,
361     )
362     circ.reset(i * 3 + 1 + 30)
363     fulladder(
364         circ,
365         19,
366         20,
367         21,
368         i * 3 + 1 + 30,
369     )
370     circ.reset(i * 3 + 2 + 30)
371     fulladder(
372         circ,
373         23,
374         24,
375         25,
376         i * 3 + 2 + 30,
377     )
378     for j in range(0, 28, 2):
379         circ.swap(j, j + 1)
380         circ.swap(28, 22)
381         circ.swap(29, 8)
382     for j in range(1, 28, 2):
383         circ.swap(j, (j + 1) % 28)
384         circ.swap(28, 22)
385         circ.swap(29, 8)
386     for i in range(30):
387         circ.swap_direct(i + 30, i)
388         circ.checkerror(i + 30, prob_2 * 3)
389     # circ.permutation(0, 30)
390     if k % measurerate == 0:
391         counts[k // measurerate][_] = circ.count(0, 30, False) / 30
392     return testresult(
393         f"{30}bit_R",
394         [i for i in range(0, totaltime + 1, measurerate)],
395         [np.average(i) for i in counts],
396         [np.std(i) for i in counts],
397     )
398
399
400 # %%
401 if __name__ == "__main__":
402     prob_2 = 0.007395 # 2-qubit gate 0.007395 0.002795 0.001433
403     shots = 20000
404     corrrate = 1 # 200
405     measurerate = 5 # 200
406     makeerror = 5
407     totaltime = 500 # 2000
408     x = range(0, totaltime + 1, measurerate)
409     allsim = {}
410
411     with Pool(19) as p:
412
413         allsim["6bit"] = p.apply_async(
414             testcircle, (6, totaltime, corrrate, makeerror, measurerate, shots, prob_2)
415         )
416         """
417         allsim["10bit_rand"] = p.apply_async(

```

```

418         testcircle_rand,
419         (10, totaltime, corrrate, makeerror, measurerate, shots, prob_2),
420     )
421     """
422     allsim["20bit_rand"] = p.apply_async(
423         testcircle_rand,
424         (20, totaltime, corrrate, makeerror, measurerate, shots, prob_2),
425     )
426
427     """
428     allsim["10bit_rand_d"] = p.apply_async(
429         testcircle_r_d,
430         (10, totaltime, corrrate, makeerror, measurerate, shots, prob_2),
431     )
432     """
433     allsim["20bit_rand_d"] = p.apply_async(
434         testcircle_r_d,
435         (20, totaltime, corrrate, makeerror, measurerate, shots, prob_2),
436     )
437     """
438     allsim["10bit_d_p"] = p.apply_async(
439         testcircle_d_p,
440         (10, totaltime, corrrate, makeerror, measurerate, shots, prob_2),
441     )
442     allsim["20bit_d_p"] = p.apply_async(
443         testcircle_d_p,
444         (20, totaltime, corrrate, makeerror, measurerate, shots, prob_2),
445     )
446     """
447     allsim["40bit_d_p"] = p.apply_async(
448         testcircle_d_p,
449         (40, totaltime, corrrate, makeerror, measurerate, shots, prob_2),
450     )
451
452     allsim["21bit_R"] = p.apply_async(
453         testcircle_R21, (totaltime, makeerror, measurerate, shots, prob_2)
454     )
455
456     allsim["30bit_R"] = p.apply_async(
457         testcircle_R30, (totaltime, makeerror, measurerate, shots, prob_2)
458     )
459
460     allsim["base"] = p.apply_async(
461         testcircle, (6, totaltime, 9999999, makeerror, measurerate, shots, prob_2)
462     )
463
464     p.close()
465     p.join()
466
467     fig = plt.figure()
468     fig.set_size_inches(16, 9)
469     for i in allsim:
470         allsim[i] = allsim[i].get()
471         allsim[i].addplot()
472
473     """
474     plt.plot(x, allsim["6bit"], color="r", label="6")
475     plt.plot(x, allsim["10bit_rand"], color="c", label="10 r")
476     plt.plot(x, allsim["20bit_rand"], color="c", label="20 r")
477     plt.plot(x, allsim["10bit_rand_d"], color="m", label="10 rd")
478     plt.plot(x, allsim["20bit_rand_d"], color="m", label="20 rd")

```



```

478 plt.plot(x, allsim["10bit_d_p"], color="gray", label="10 dp")
479 plt.plot(x, allsim["20bit_d_p"], color="gray", label="20 dp")
480 plt.plot(x, allsim["40bit_d_p"], color="gray", label="40 dp")
481 plt.plot(x, allsim["21bit_R"], color="gold", label="21 R")
482 plt.plot(x, allsim["30bit_R"], color="gold", label="30 R")
483
484 plt.plot(x, allsim["base"], color="g", label="no corr")
485 """
486
487 plt.title(f"correct per {makeerror} x gates prob_2={prob_2}") # title
488 plt.ylabel("correct rate") # y label
489 plt.xlabel(f"{makeerror}x CX gates") # x label
490 plt.legend() # show legend
491 plt.show()
492
493 print(allsim["30bit_R"])
494
495 # %%
496

```


參考文獻

<https://zh.wikipedia.org/zh-tw/%E9%87%8F%E5%AD%90%E9%96%98>

https://www.researchgate.net/figure/a-Toffoli-gate-b-Swap-gate-c-Quantum-gate-d-Quantum-gate-stored-in-a-superposition_fig13_257641933

圖目錄

表目錄

