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: 黃琮暐教授、張元翔教授

中華民國 113 年 2 月

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。

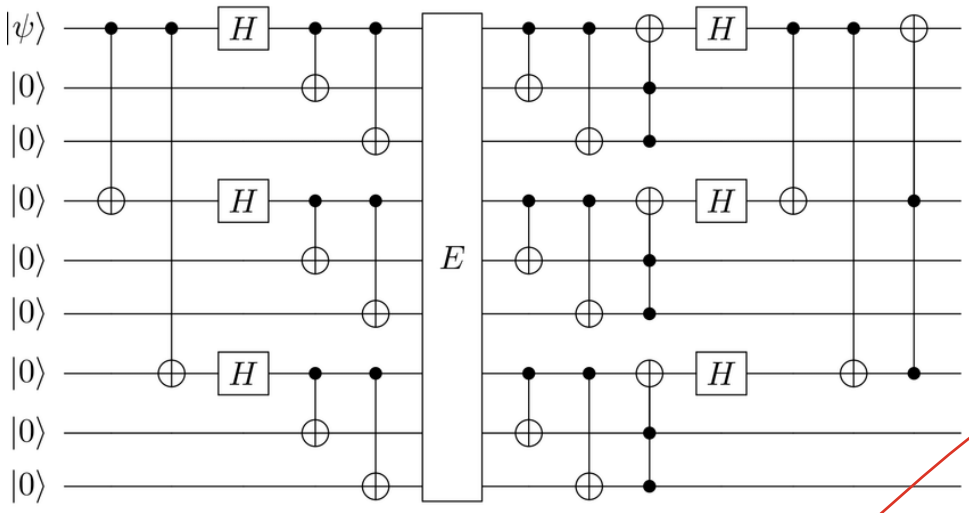
位元翻轉錯誤（Bit-flip error）：

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

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

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。

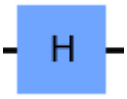
量子邏輯閘使用么正矩陣表示，而基本狀態$|0\rangle$以$\begin{bmatrix}1\\0\end{bmatrix}$表示，$|1\rangle$以$\begin{bmatrix}0\\1\end{bmatrix}$表示。常見的量子閘有 CNOT gate、Hadamard gate、Pauli-X gate、Pauli-Y gate、Pauli-Z gate、Toffoli gate：

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。

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

Hadamard gate



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

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

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

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

Pauli-X gate(NOT) 

繞 Bloch 球體的 x 軸旋轉 180 度。

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

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

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

Pauli-Y gate 

繞 Bloch 球體的 y 軸旋轉 180 度。

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

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

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

Pauli-Z gate 

繞 Bloch 球體的 z 軸旋轉 180 度。

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

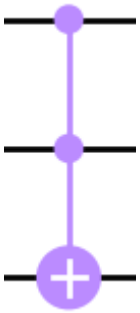
$$Z|0\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$$

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

Toffoli gate (CCX gate)

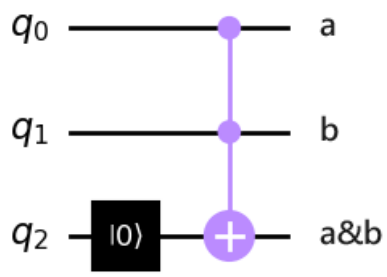
操作三個量子位元的閘,有 2-qubit 輸入與 1-qubit 輸出。若前兩個 qubit 皆為$|0\rangle$則對第三個 qubit 做 Pauli-X gate 運算，否則不動作。

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

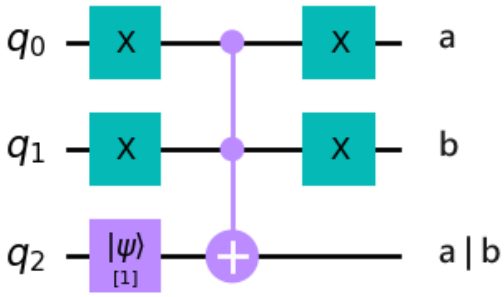


用 CNOT gate 實作量子位元的 AND 閘和 OR 閘：

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。

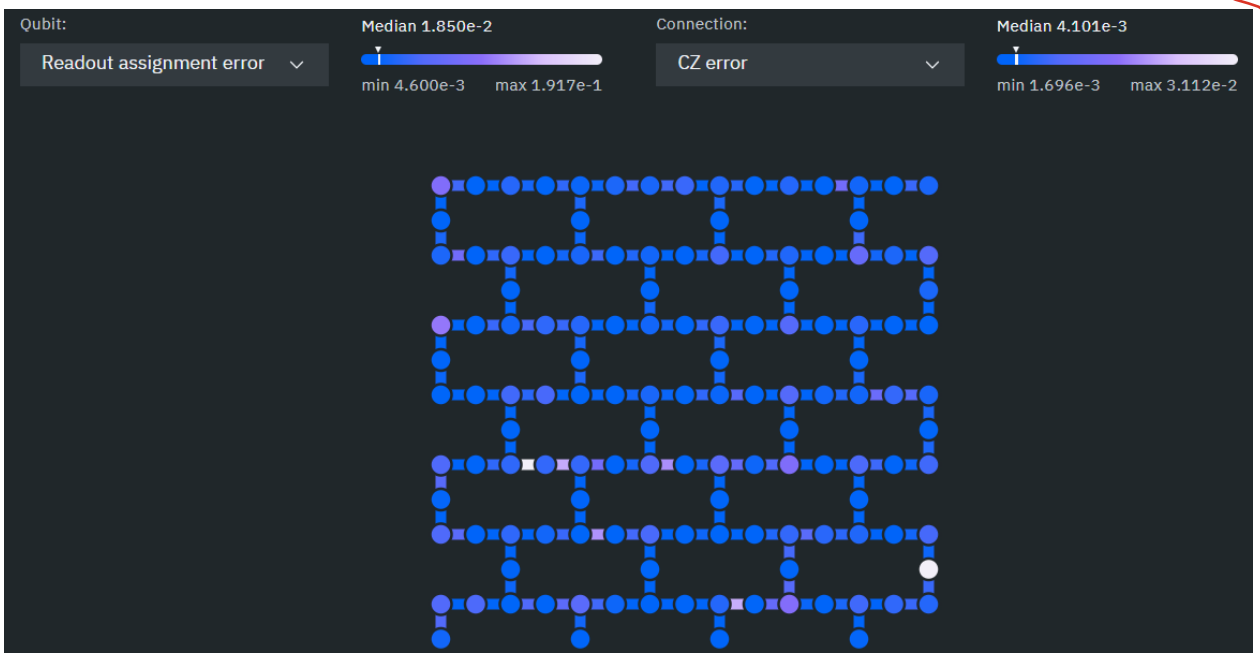
相位糾正錯誤（Phase Correction Errors）： 這種錯誤發生在嘗試糾正相位錯誤時出現的錯誤，可能導致進一步的計算錯誤。

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 processer 的 error rate map

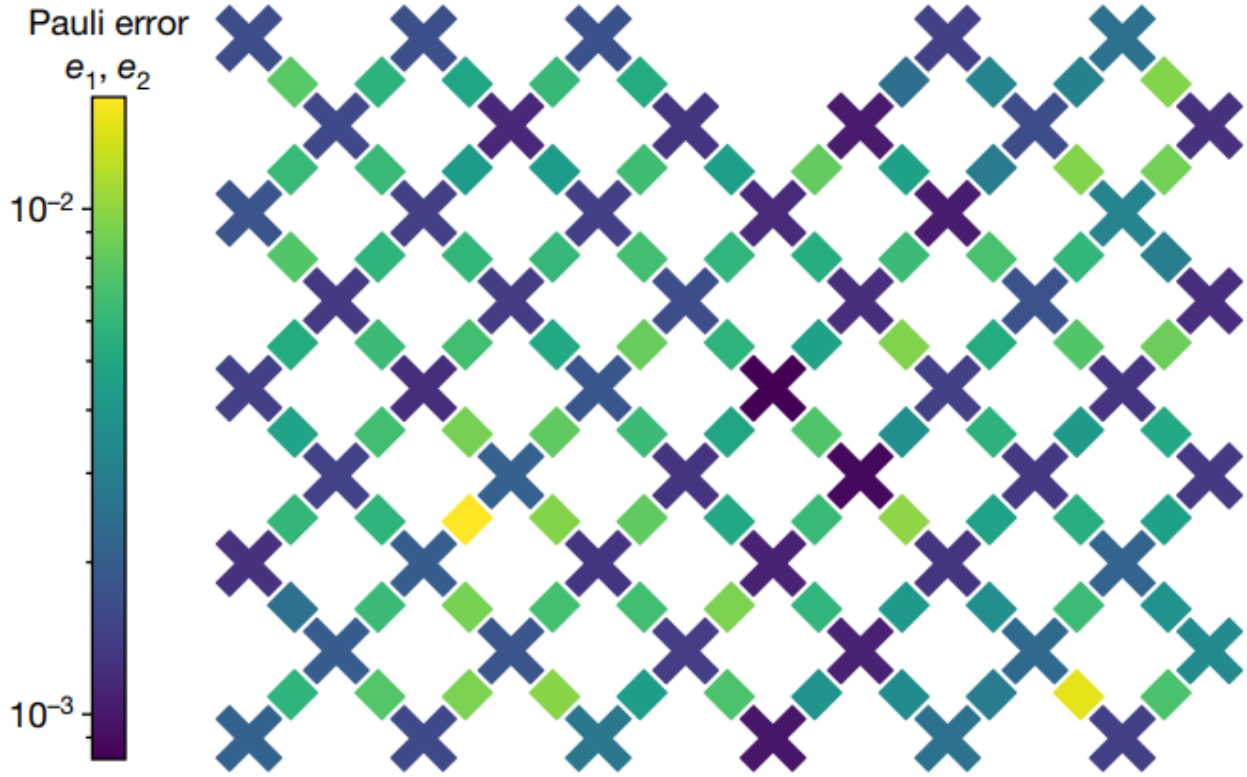


From:
https://quantum.ibm.com/services/resources?tab=systems&system=ibm_torino

CZgate 是這個 quantum processer 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：

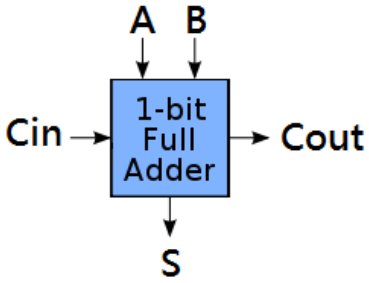
1 -> 111

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

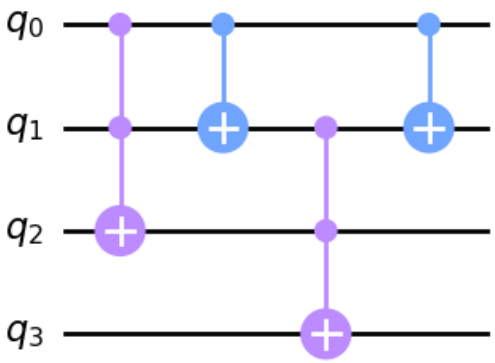


Input			Output	
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

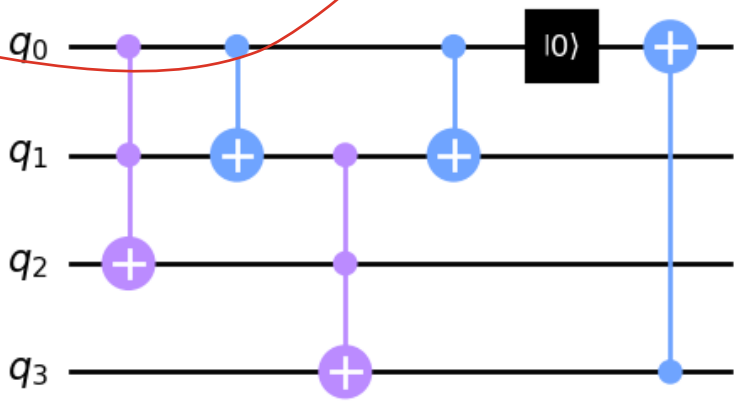
量子電路實作全加器

input：q0,q1,q2

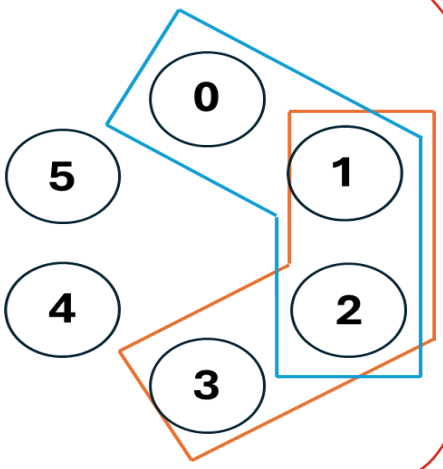
output：q3(carry)



再將結果寫回 q0：



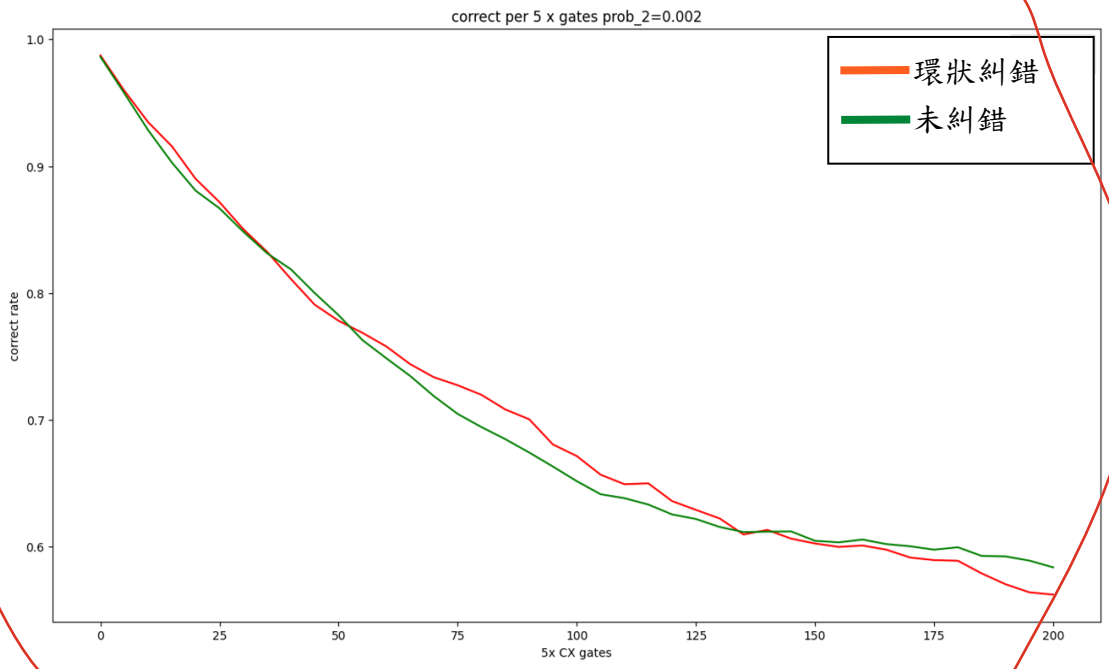
環狀分組糾錯



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

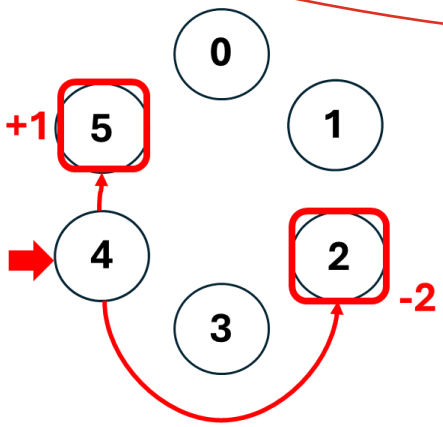
000101 -> 000000

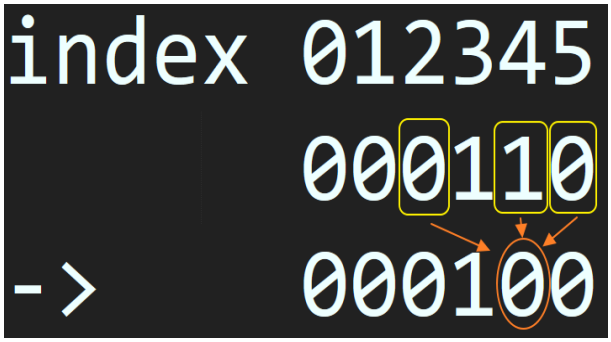
000110 -> 000110 (Error)

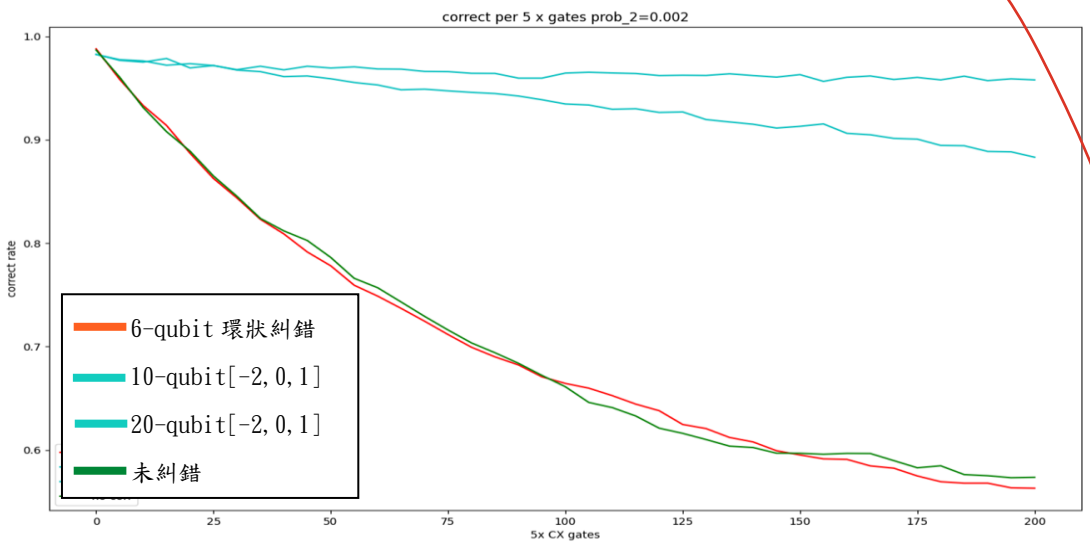
假設目的是將所有 qubit 糾正為 0，則當遇到連續兩個 1 時會發生死循環，無論糾錯幾次都無法修正為 0。意即，只要有連續的錯誤就是就會沒辦法修正，連續兩個 qubit 同時壞掉是很容易發生的，尤其 cx-gate(或其他雙 qubit gate)。為解開死循環，須將取樣順序打亂。

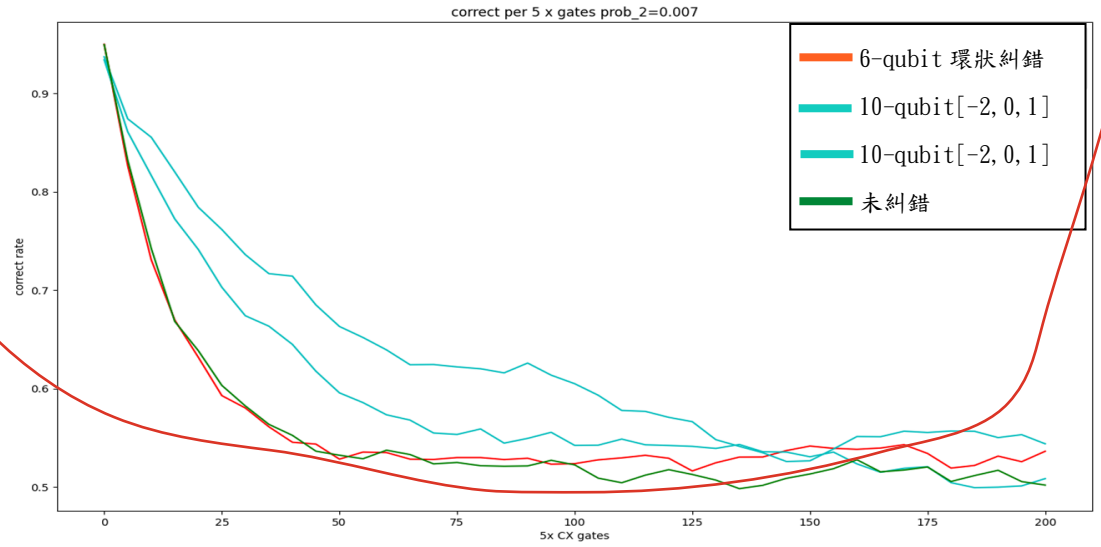
如何取樣

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。









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 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。

date qubit:　　100001

support qubit:000000　　calculate->

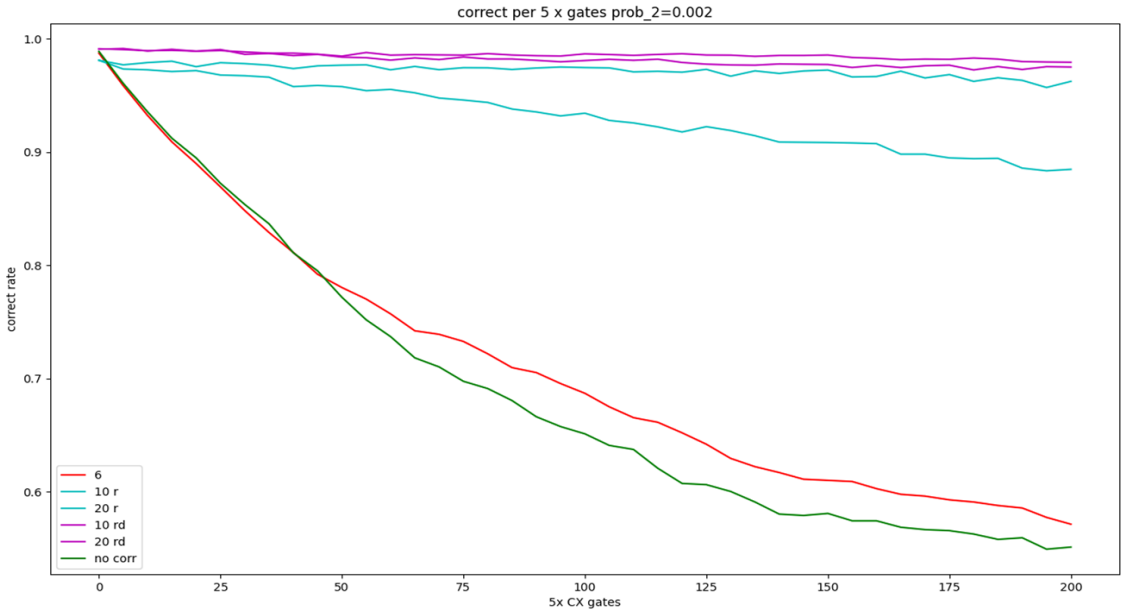
date qubit:　　100001

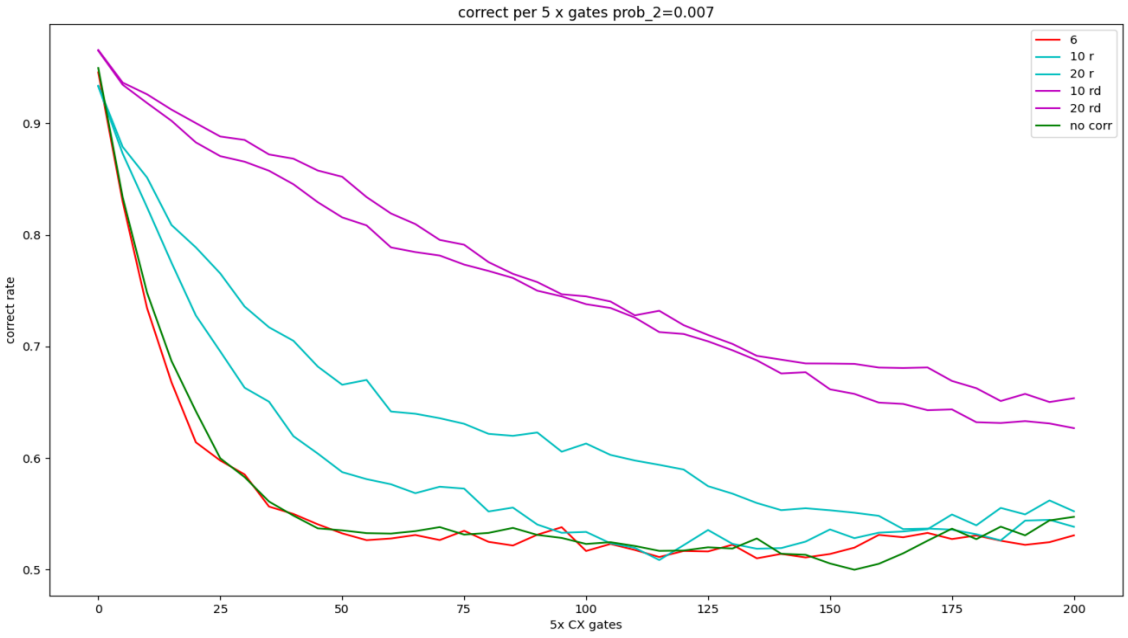
support qubit:000001　　copy(write back)

date qubit:　　000001

support qubit:000001

以下兩圖分別為錯率 0.002 與 0.007 的糾錯結果：

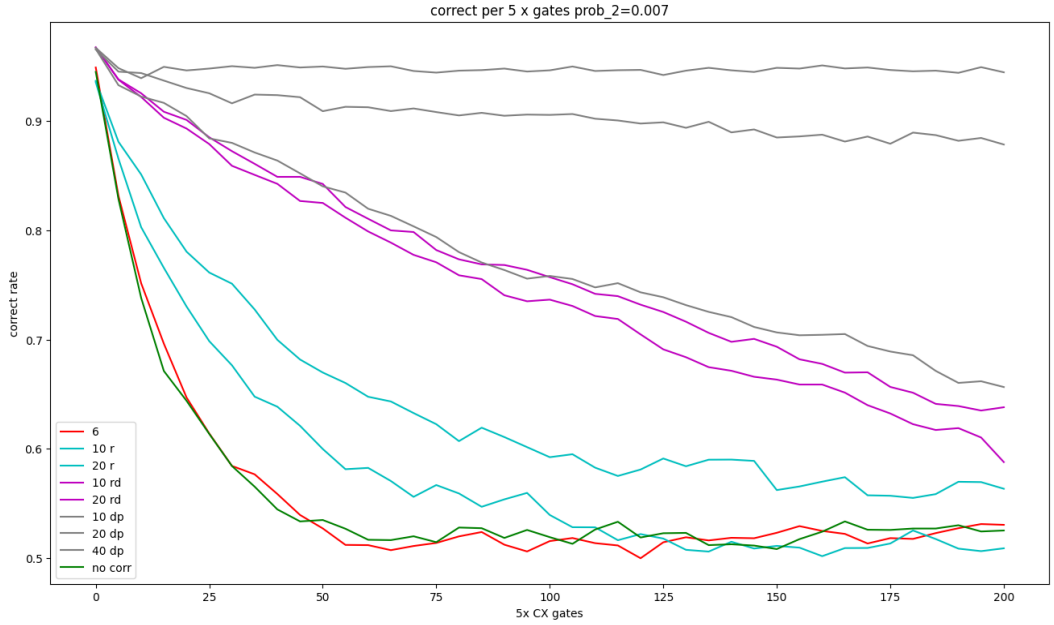


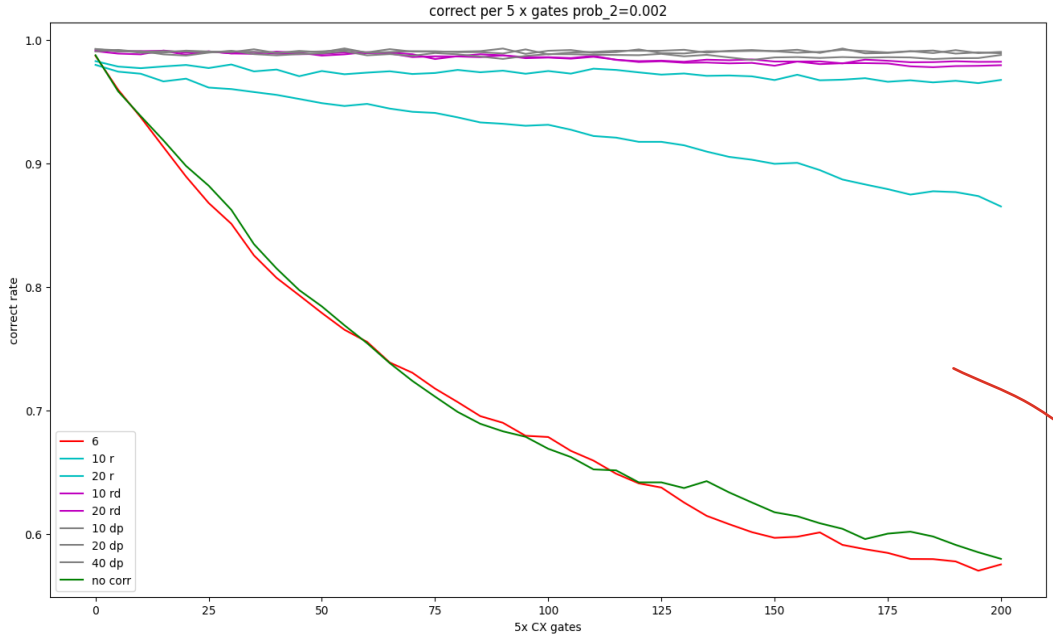


可以發現錯誤率稍微變大，糾錯率就大幅降低。

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)。



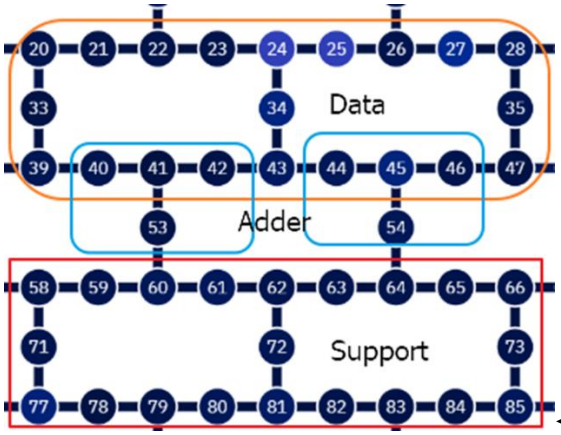


雖然現實中不存在這種隨機打亂的方法，但可以觀察「隨機打亂」是否有助於大幅提升糾錯能力，結果發現確實是有助於的。

->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。

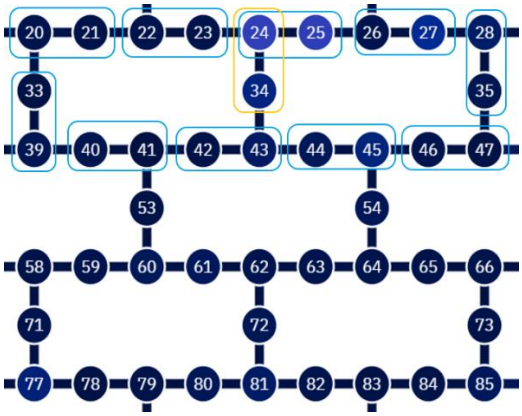
理想中的隨機打亂??

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：

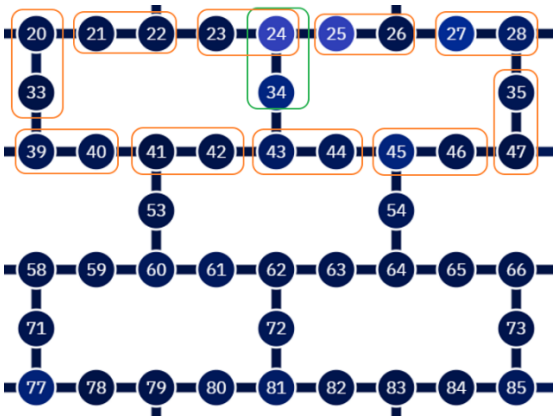


←圖一

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↑



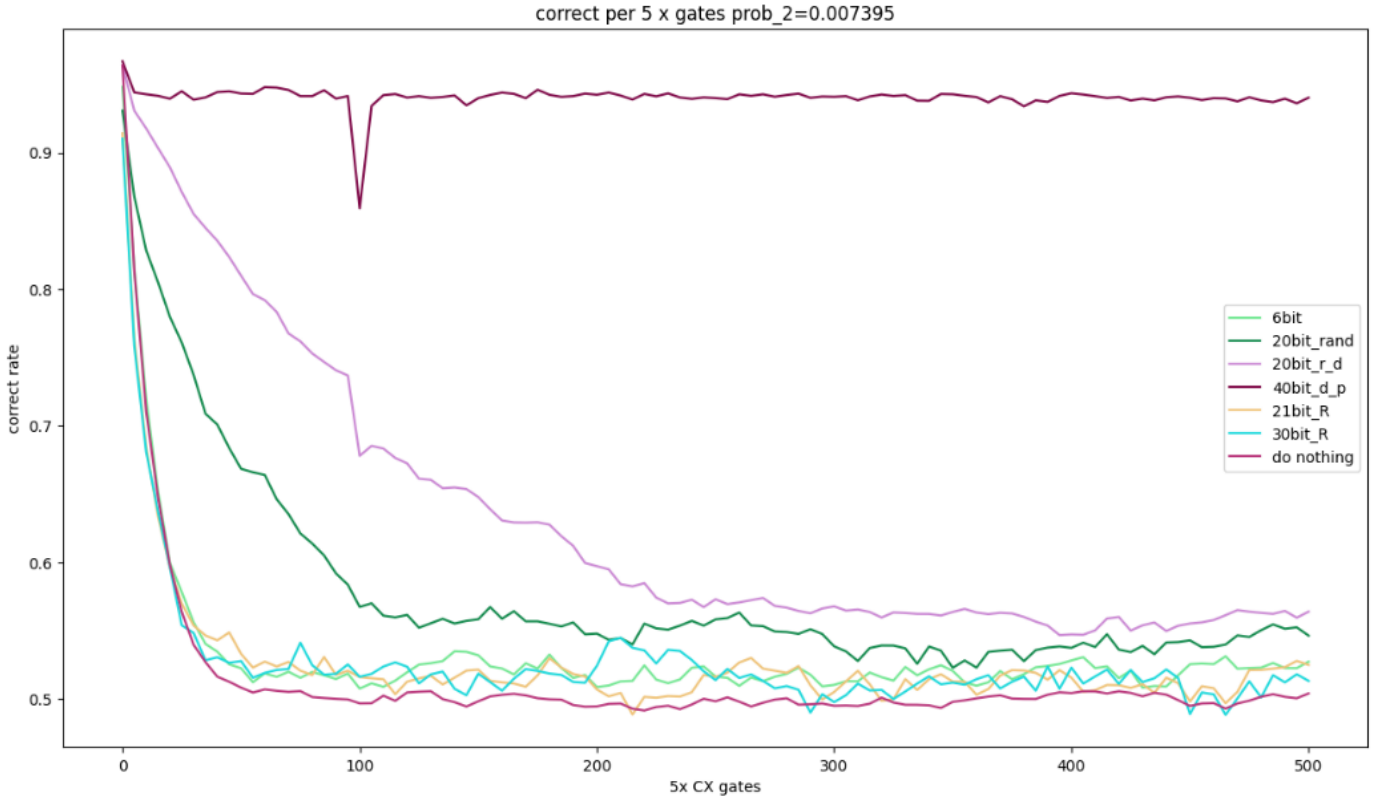
圖三↑

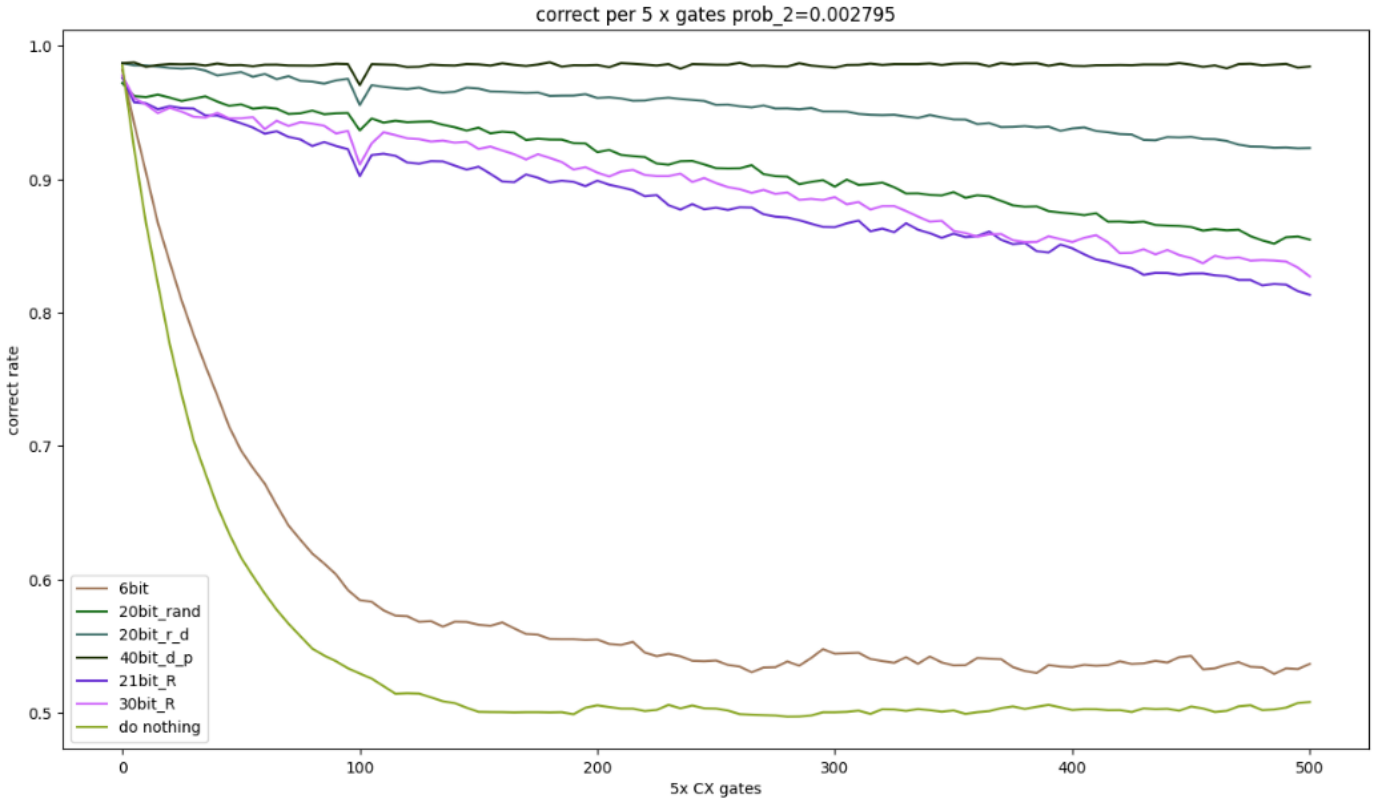
圖二、圖三：編號 53、54 已存放糾錯後的結果後，將待糾錯資料的 qubit 兩兩位置交換，交換方式如圖二的藍色框線所示，完成以後再如圖三的橘線所示兩兩互換位置，即完成一次打亂。打亂後得到新的編號 40、41、42 的 qubit，取此三個 qubit 進行糾錯得結果存入編號 60 的 qubit。以次類推，每次打亂後取編號 40、41、42 的 qubit 糾錯，將結果依序填入下方儲存空間，直至空間填滿，即獲得所有糾錯結果。

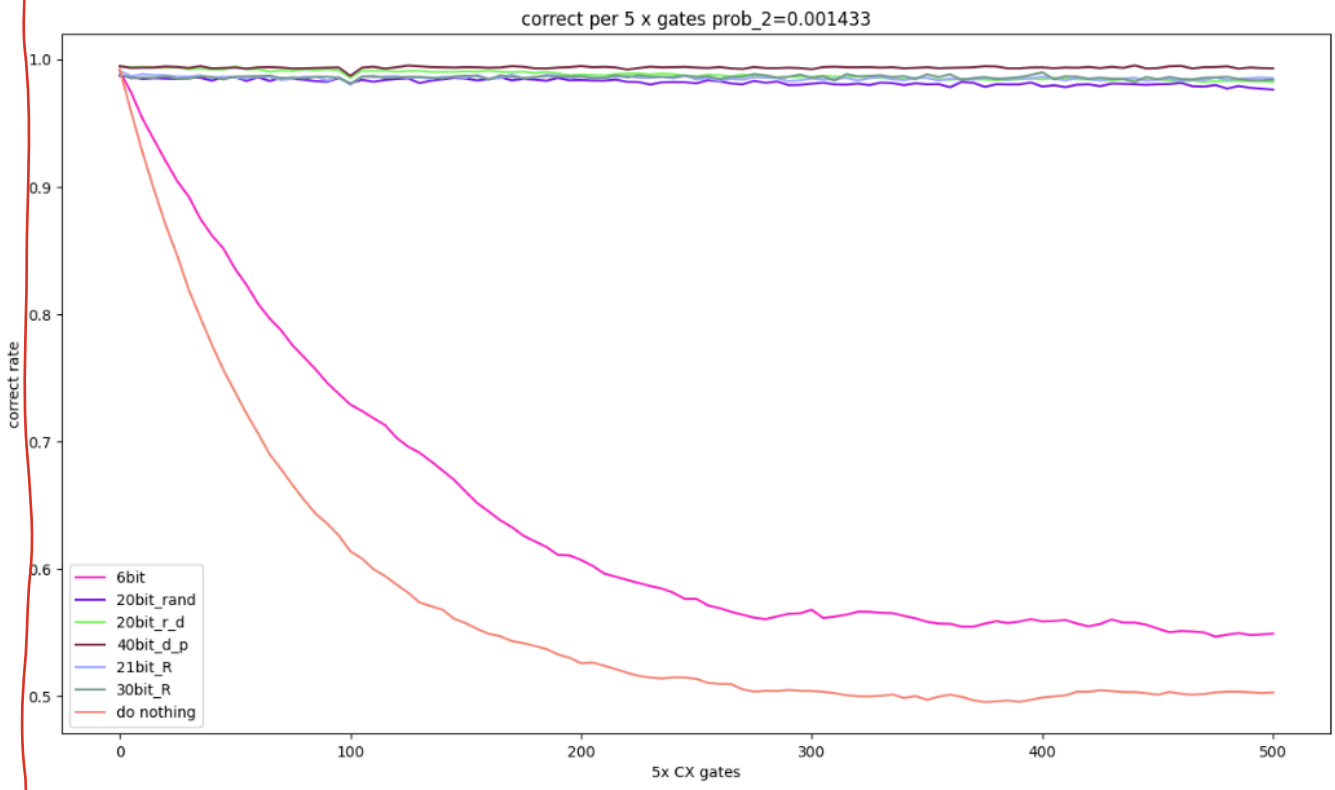
每次打亂後取的 qubit 情形舉例：

[(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


correct per 5 x gates prob_2=0.007395
correct rate
5x CX gates
6bit
20bit_rand
20bit_r_d
40bit_d_p
21bit_R
30bit_R
do nothing


correct per 5 x gates prob_2=0.002795
correct rate
5x CX gates
6bit
20bit_rand
20bit_r_d
40bit_d_p
21bit_R
30bit_R
do nothing



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

```python
# %%
import numpy as np
from statistics import median
import matplotlib.pyplot as plt
from tqdm import tqdm
from multiprocessing import Pool


class testresult:
    def __init__(self, testname, measurex, result, yerr):
        self.testname = testname  # "6bit"
        self.measurex = measurex  # [0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
        self.result = result  # [0.9,0.8,0.7,0.6,0.5,0.4,0.3,0.2,0.1,0.0]
        self.yerr = yerr  # [0.01,0.01,0.01,0.01,0.01,0.01,0.01,0.01,0.01,0.01]

    def addplt(self):
        # random color
        col = (np.random.random(), np.random.random(), np.random.random())
        """
        plt.errorbar(
            self.measurex,
            self.result,
            yerr=[i * 1 for i in self.yerr],
            color=col,
            label=self.testname,
            fmt="k",
        )
        """

        plt.plot(
            self.measurex,
            self.result,
            color=col,
            label=self.testname,
        )

    def __str__(self):
        return f"testname:{self.testname}\nresult:{self.result}\nyerr:{self.yerr}"


class boolcirc:
    def __init__(self, qubit, errorrate):
        # self.qubit = np.array([False] * qubit, dtype=np.bool_)
        self.qubit = [False] * qubit
        self.prob = errorrate

    def checkerror(self, i, rate):
        if np.random.rand() < self.prob * rate:
            self.qubit[i] = not self.qubit[i]

    def checkerror_both(self, i, j, rate):
        if np.random.rand() < self.prob * rate:
            self.qubit[i] = not self.qubit[i]
            self.qubit[j] = not self.qubit[j]

    def checkerror_ecr(self, i, j):
        self.checkerror(i, 0.25)
```

```
        self.checkerror(j, 0.25)
        self.checkerror_both(i, j, 0.25)

    def reset(self, a):
        self.qubit[a] = False

    def x(self, a):
        self.qubit[a] = not self.qubit[a]

    def cx(self, a, b):
        if self.qubit[a]:
            self.qubit[b] = not self.qubit[b]
        self.checkerror_ecr(a, b)

    def ccx(self, c1, c2, t):
        beginstate = self.qubit[c1] ^ self.qubit[c2]

        self.checkerror_ecr(c1, t)
        self.checkerror_ecr(c2, t)
        self.checkerror_ecr(c1, t)
        self.checkerror_ecr(c2, t)

        if self.qubit[c1] and self.qubit[c2]:
            self.qubit[t] = not self.qubit[t]

        self.qubit[t] ^= beginstate ^ self.qubit[c1] ^ self.qubit[c2]

    def swap(self, a, b):
        self.cx(a, b)
        self.cx(b, a)
        self.cx(a, b)

    def swap_direct(self, a, b):
        self.qubit[a], self.qubit[b] = self.qubit[b], self.qubit[a]

    def permutation(self, a, b):
        self.qubit[a:b] = np.random.permutation(self.qubit[a:b])

    def count(self, f, e, value):
        count = 0
        for i in range(f, e):
            if self.qubit[i] == value:
                count += 1
        return count

    def __str__(self):
        s = ""
        for i in self.qubit:
            if i:
                s += "1"
            else:
                s += "0"
        return s


# %% Make a circuit
def fulladder(circuit, in1, in2, in3, carry):
    """in1, in2, in3 are the input qubits, carry is the output qubit,!!no sum"""
    circuit.ccx(in1, in2, carry)
    circuit.cx(in1, in2)
```

```
    circuit.ccx(in2, in3, carry)
    circuit.cx(in1, in2)


def testcircle(numofqubit, totaltime, coorrate, makeerror, measurerate, shots, prob_2):
    counts = np.empty([(totaltime // measurerate + 1), shots // numofqubit])
    for _ in tqdm(range(shots // numofqubit)):
        circ = boolcirc(numofqubit + 1, prob_2)
        for k in range(totaltime + 1):
            if k == 100:
                for i in range(numofqubit):
                    circ.checkerror(i, 20)
            if makeerror > 0:
                for i in range(numofqubit):
                    circ.checkerror(i, makeerror)
            if k % coorrate == coorrate - 1:
                for i in range(numofqubit):
                    fulladder(
                        circ, (i - 1) % numofqubit, i, (i + 1) % numofqubit, numofqubit
                    )
                    circ.reset(i)
                    circ.cx(numofqubit, i)
                    circ.reset(numofqubit)
            if k % measurerate == 0:
                counts[k // measurerate][_] = (
                    circ.count(0, numofqubit, False) / numofqubit
                )
    return testresult(
        f"{numofqubit}bit" if coorrate < 10000 else "do nothing",
        [i for i in range(0, totaltime + 1, measurerate)],
        [np.average(i) for i in counts],
        [np.std(i) for i in counts],
    )


def testcircle_rand(
    numofqubit, totaltime, coorrate, makeerror, measurerate, shots, prob_2
):
    counts = np.empty([(totaltime // measurerate + 1), shots // numofqubit])
    for _ in tqdm(range(shots // numofqubit)):
        circ = boolcirc(numofqubit + 1, prob_2)
        count = 0
        for k in range(totaltime + 1):
            if k == 100:
                for i in range(numofqubit):
                    circ.checkerror(i, 20)
            if makeerror > 0:
                for i in range(numofqubit):
                    circ.checkerror(i, makeerror)
            if k % coorrate == coorrate - 1:
                for i in range(numofqubit):
                    fulladder(
                        circ,
                        (i - 1 - count % 2) % numofqubit,
                        i,
                        (i + 1 + (count + 1) % 2) % numofqubit,
                        numofqubit,
                    )
                    circ.reset(i)
                    circ.cx(numofqubit, i)
```

```
                    circ.reset(numofqubit)
                for i in range(0, numofqubit, 2):
                    circ.swap(i, (i + 1) % numofqubit)
                for i in range(1, numofqubit, 2):
                    circ.swap(i, (i + 1) % numofqubit)
            if k % measurerate == 0:
                counts[k // measurerate][_] = (
                    circ.count(0, numofqubit, False) / numofqubit
                )
    return testresult(
        f"{numofqubit}bit_rand",
        [i for i in range(0, totaltime + 1, measurerate)],
        [np.average(i) for i in counts],
        [np.std(i) for i in counts],
    )


def testcircle_r_d(
    numofqubit, totaltime, coorrate, makeerror, measurerate, shots, prob_2
):
    counts = np.empty([(totaltime // measurerate + 1), shots // numofqubit])
    for _ in tqdm(range(shots // numofqubit)):
        circ = boolcirc(numofqubit * 2, prob_2)
        count = 0
        for k in range(totaltime + 1):
            if k == 100:
                for i in range(numofqubit):
                    circ.checkerror(i, 20)
            if makeerror > 0:
                for i in range(numofqubit):
                    circ.checkerror(i, makeerror)
            if k % coorrate == coorrate - 1:
                for i in range(numofqubit):
                    circ.reset(i + numofqubit)
                    fulladder(
                        circ,
                        (i - 1 - count % 2) % numofqubit,
                        i,
                        (i + 1 + (count + 1) % 2) % numofqubit,
                        i + numofqubit,
                    )
                for i in range(0, numofqubit, 2):
                    circ.swap(i, (i + 1) % numofqubit)
                for i in range(1, numofqubit, 2):
                    circ.swap(i, (i + 1) % numofqubit)
            for i in range(numofqubit):
                circ.swap_direct(i + numofqubit, i)
            if k % measurerate == 0:
                counts[k // measurerate][_] = (
                    circ.count(0, numofqubit, False) / numofqubit
                )
    return testresult(
        f"{numofqubit}bit_r_d",
        [i for i in range(0, totaltime + 1, measurerate)],
        [np.average(i) for i in counts],
        [np.std(i) for i in counts],
    )


def testcircle_d_p(
```

```
    numofqubit, totaltime, coorrate, makeerror, measurerate, shots, prob_2
):
    counts = np.empty([(totaltime // measurerate + 1), shots // numofqubit])
    for _ in tqdm(range(shots // numofqubit)):
        circ = boolcirc(numofqubit * 2, prob_2)
        for k in range(totaltime + 1):
            if k == 100:
                for i in range(numofqubit):
                    circ.checkerror(i, 20)
            if makeerror > 0:
                for i in range(numofqubit):
                    circ.checkerror(i, makeerror)
            if k % coorrate == coorrate - 1:
                for i in range(numofqubit):
                    circ.reset(i + numofqubit)
                    fulladder(
                        circ,
                        (i - 1) % numofqubit,
                        i,
                        (i + 1) % numofqubit,
                        i + numofqubit,
                    )
            for i in range(numofqubit):
                circ.swap_direct(i + numofqubit, i)
            circ.permutation(0, numofqubit)
            if k % measurerate == 0:
                counts[k // measurerate][_] = (
                    circ.count(0, numofqubit, False) / numofqubit
                )
    return testresult(
        f"{numofqubit}bit_d_p",
        [i for i in range(0, totaltime + 1, measurerate)],
        [np.average(i) for i in counts],
        [np.std(i) for i in counts],
    )


def testcircle_R21(totaltime, makeerror, measurerate, shots, prob_2):
    """
     0  1  2  3  4  5  6  7  8
    19          20          9
    18 17 16 15 14 13 12 11 10
    """
    counts = np.empty([(totaltime // measurerate + 1), shots // 21])
    for _ in tqdm(range(shots // 21)):
        circ = boolcirc(21 * 2, prob_2)
        for k in range(totaltime + 1):
            if k == 100:
                for i in range(21):
                    circ.checkerror(i, 20)
            if makeerror > 0:
                for i in range(21):
                    circ.checkerror(i, makeerror)
            for i in range(10):
                circ.reset(i * 2 + 21)
                fulladder(
                    circ,
                    15,
                    16,
                    17,
```

```
                    i * 2 + 21,
                )
                circ.reset(i * 2 + 1 + 21)
                fulladder(
                    circ,
                    11,
                    12,
                    13,
                    i * 2 + 1 + 21,
                )
                for j in range(0, 20, 2):
                    circ.swap(j, j + 1)
                circ.swap(20, 14)
                for j in range(1, 20, 2):
                    circ.swap(j, (j + 1) % 20)
                circ.swap(20, 14)
            circ.reset(41)
            fulladder(
                circ,
                15,
                16,
                17,
                41,
            )

            for i in range(21):
                circ.swap_direct(i + 21, i)
                circ.checkerror(i + 21, prob_2 * 3)
            # circ.permutation(0, 21)
            if k % measurerate == 0:
                counts[k // measurerate][_] = circ.count(0, 21, False) / 21
    return testresult(
        f"{21}bit_R",
        [i for i in range(0, totaltime + 1, measurerate)],
        [np.average(i) for i in counts],
        [np.std(i) for i in counts],
    )


def testcircle_R30(totaltime, makeerror, measurerate, shots, prob_2):
    """
     0  1  2  3  4  5  6  7  8  9 10 11 12
    27          28          29          13
    26 25 24 23 22 21 20 19 18 17 16 15 14
    """
    counts = np.empty([(totaltime // measurerate + 1), shots // 30])
    for _ in tqdm(range(shots // 30)):
        circ = boolcirc(30 * 2, prob_2)
        for k in range(totaltime + 1):
            if k == 100:
                for i in range(30):
                    circ.checkerror(i, 20)
            if makeerror > 0:
                for i in range(30):
                    circ.checkerror(i, makeerror)
            for i in range(10):
                circ.reset(i * 3 + 30)
                fulladder(
                    circ,
                    15,
```

```
                    16,
                    17,
                    i * 3 + 30,
                )
                circ.reset(i * 3 + 1 + 30)
                fulladder(
                    circ,
                    19,
                    20,
                    21,
                    i * 3 + 1 + 30,
                )
                circ.reset(i * 3 + 2 + 30)
                fulladder(
                    circ,
                    23,
                    24,
                    25,
                    i * 3 + 2 + 30,
                )
                for j in range(0, 28, 2):
                    circ.swap(j, j + 1)
                circ.swap(28, 22)
                circ.swap(29, 8)
                for j in range(1, 28, 2):
                    circ.swap(j, (j + 1) % 28)
                circ.swap(28, 22)
                circ.swap(29, 8)
            for i in range(30):
                circ.swap_direct(i + 30, i)
                circ.checkerror(i + 30, prob_2 * 3)
            # circ.permutation(0, 30)
            if k % measurerate == 0:
                counts[k // measurerate][_] = circ.count(0, 30, False) / 30
    return testresult(
        f"{30}bit_R",
        [i for i in range(0, totaltime + 1, measurerate)],
        [np.average(i) for i in counts],
        [np.std(i) for i in counts],
    )


# %%
if __name__ == "__main__":
    prob_2 = 0.007395  # 2-qubit gate 0.007395 0.002795 0.001433
    shots = 20000
    corrrate = 1  # 200
    measurerate = 5  # 200
    makeerror = 5
    totaltime = 500  # 2000
    x = range(0, totaltime + 1, measurerate)
    allsim = {}

    with Pool(19) as p:

        allsim["6bit"] = p.apply_async(
            testcircle, (6, totaltime, corrrate, makeerror, measurerate, shots, prob_2)
        )
        """
        allsim["10bit_rand"] = p.apply_async(
```

```
            testcircle_rand,
            (10, totaltime, corrrate, makeerror, measurerate, shots, prob_2),
        )
        """
        allsim["20bit_rand"] = p.apply_async(
            testcircle_rand,
            (20, totaltime, corrrate, makeerror, measurerate, shots, prob_2),
        )

        """
        allsim["10bit_rand_d"] = p.apply_async(
            testcircle_r_d,
            (10, totaltime, corrrate, makeerror, measurerate, shots, prob_2),
        )
        """
        allsim["20bit_rand_d"] = p.apply_async(
            testcircle_r_d,
            (20, totaltime, corrrate, makeerror, measurerate, shots, prob_2),
        )
        """
        allsim["10bit_d_p"] = p.apply_async(
            testcircle_d_p,
            (10, totaltime, corrrate, makeerror, measurerate, shots, prob_2),
        )
        allsim["20bit_d_p"] = p.apply_async(
            testcircle_d_p,
            (20, totaltime, corrrate, makeerror, measurerate, shots, prob_2),
        )
        """
        allsim["40bit_d_p"] = p.apply_async(
            testcircle_d_p,
            (40, totaltime, corrrate, makeerror, measurerate, shots, prob_2),
        )

        allsim["21bit_R"] = p.apply_async(
            testcircle_R21, (totaltime, makeerror, measurerate, shots, prob_2)
        )

        allsim["30bit_R"] = p.apply_async(
            testcircle_R30, (totaltime, makeerror, measurerate, shots, prob_2)
        )

        allsim["base"] = p.apply_async(
            testcircle, (6, totaltime, 9999999, makeerror, measurerate, shots, prob_2)
        )

        p.close()
        p.join()

    fig = plt.figure()
    fig.set_size_inches(16, 9)
    for i in allsim:
        allsim[i] = allsim[i].get()
        allsim[i].addplt()
    """
    plt.plot(x, allsim["6bit"], color="r", label="6")
    plt.plot(x, allsim["10bit_rand"], color="c", label="10 r")
    plt.plot(x, allsim["20bit_rand"], color="c", label="20 r")
    plt.plot(x, allsim["10bit_rand_d"], color="m", label="10 rd")
    plt.plot(x, allsim["20bit_rand_d"], color="m", label="20 rd")
```

```
    plt.plot(x, allsim["10bit_d_p"], color="gray", label="10 dp")
    plt.plot(x, allsim["20bit_d_p"], color="gray", label="20 dp")
    plt.plot(x, allsim["40bit_d_p"], color="gray", label="40 dp")
    plt.plot(x, allsim["21bit_R"], color="gold", label="21 R")
    plt.plot(x, allsim["30bit_R"], color="gold", label="30 R")

    plt.plot(x, allsim["base"], color="g", label="no corr")
    """

    plt.title(f"correct per {makeerror} x gates prob_2={prob_2}")  # title
    plt.ylabel("correct rate")  # y label
    plt.xlabel(f"{makeerror}x CX gates")  # x label
    plt.legend()  # show legend
    plt.show()

    print(allsim["30bit_R"])

    # %%

```

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