PAPER

Quantum software development

Quantum error correction with the quantum Fourier transform algorithm

Benjamin Eder

Email: beder@hm.edu

Munich University of Applied Sciences

ABSTRACT

This work represents an attempt to apply quantum error correction with a quantum Fourier transform circuit. The circuit structure will be demonstrated and tested both with the Qiskit simulator and with real quantum backends from the IBM Quantum Experience platform. We notice that we are more or less able to correct bit flips with the circuit when we use the simulator. In addition, we can observe that real quantum devices are still very noisy and our error correction method therefore does not lead to good results.

CONTENTS

1 | INTRODUCTION

While quantum computing is still at an early stage, many quantum algorithms seem to provide drastic accelerations compared to their current classical counterparts. One of the best known of these is Shor's algorithm which is almost exponentially faster than the currently most efficient factoring algorithms and thus poses a threat to cryptosystems that rely on the factoring problem being hard to solve (Shor, 1994).

However building a quantum computer and making use of it is not an easy task. It is essential for such a quantum system to be isolated from unknown interactions with the external world which may mess up the internal state and thus the received results (Milburn, Sarovar, & Ahn, 2004, p. 34). Simultaneously those systems need to be manipulable to an extent in order to be able to implement quantum gates or encoding starting states for the individual qubits (Matuschak & Nielsen, 2019).

Qubits on their own, as well as in an entangled state are error prone in the presence of *noise* and *decoherence* (Milburn et al., 2004, p. 34). Both appear even with current quantum systems. To protect fragile quantum information and build fault-tolerant quantum computers, we use quantum error correction (*QEC*) (Li, 2020, p. 46998). QEC is achieved by using special codes to encode quantum information (de Brito, do Nascimento, & Ramos, 2008, p. 113). Most codes are based on redundancy, which means the introduction of multiple qubits, instead of just one, that carry the information (de Brito et al., 2008, p. 113).

1.1 | Background

This examination paper was written in the context of the lecture *Quantum Software Development*, which was held by *Prof. Dr. Sabine Tornow* in the summer semester of 2020 at the *Munich University of Applied Sciences*.

1.2 | Description

Li (2020, p. 46998) mentions that quantum error correction is already achievable on platforms such as *IBM Q Experience* (*IBM Quantum Experience*, 2020). Using that platform this paper describes an attempt to correct bit-flip errors in a quantum circuit performing a quantum Fourier transform on a real quantum device. Before testing on the actual computer provided by the Cloud service, the needed circuit has to be modelled and tested using a Simulator. IBM provides a library `Qiskit` which is usable with the programming language `Python` and offers the needed Simulator as well as an interface to use the IBM Q Experience platform (*Qiskit*, 2020).

1.3 | Structure

To give an overview over the structure of the paper, the following sections are quickly described.

The paper starts with a quick recapitulation about the **basic concepts**, namely *quantum error correction*, *quantum Fourier transform* and **related work**. Afterwards the **used methods and tools** are introduced with which the quantum circuit is modelled and tested, followed by the actual **implementation**. The resulting circuit is then tested and results from the simulator as well as the real quantum computer **analyzed**.

A **conclusion** about the resulting paper is drawn at the end with some *proposals for further studies*.

2 | BASIC CONCEPTS

Before delving into the details of implementing the quantum circuit with error correction, a brief recapitulation about the basic concepts is helpful. To keep it short, only the *immediately needed* concepts are described. It is assumed that the reader is already familiar with the principles of quantum computing.

2.1 | Quantum Error Correction

Error correction to fight noise has been around in the classical computing domain for a long time (Benenti, Casati, & Strini, 2004, p. 459). It has been discovered that one of the key ingredients is **redundancy** (Benenti et al., 2004, p. 459). Fortunately the same happens to be true for the quantum domain (Li, 2020, p. 46998). To apply a simple error correction one could use three instead of just one qubit carrying the same information just like in classical computing. Unfortunately when dealing with qubits we have to obey some additional rules.

1. There is the no-cloning theorem which means it is impossible to just copy quantum states (Benenti et al., 2004, p. 460). We are left with encoding the same starting states of individual qubits or using `CNOT` gates to mimic copying behavior (Benenti et al., 2004, p. 461). The approach using `CNOT` gates is shown in figure 1.

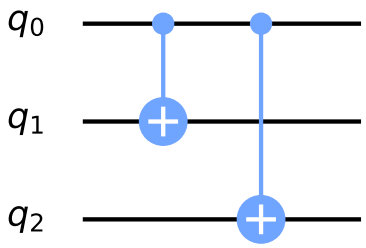

FIGURE 1 `CNOT` gates used to encode the information of qubit q_0 onto three.

2. Measuring a qubit causes its state to collapse (Benenti et al., 2004, p. 460). Thus, we cannot just measure the state and correct it as it will destroy any superposition we ideally want to continue working with. The introduction of ancillary qubits that measure the *error syndrome* solves that problem (Benenti et al., 2004, p. 463) as shown in figure 2.

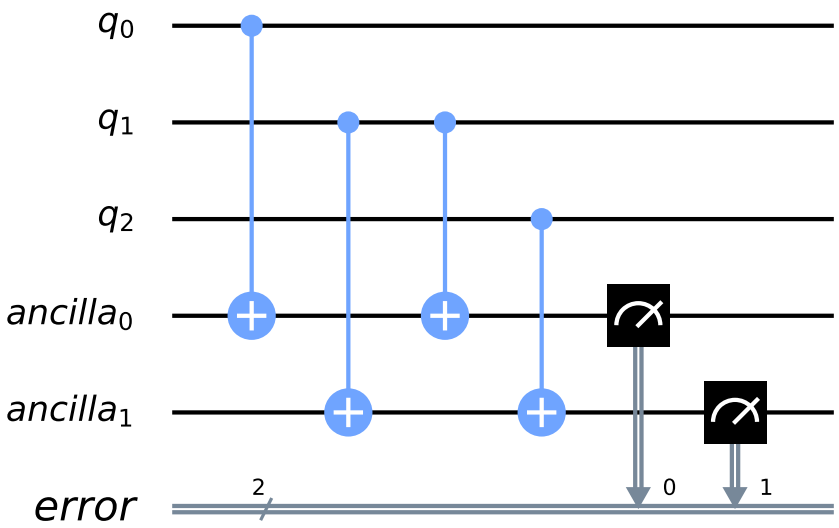

FIGURE 2 Measuring the *error syndrome* using ancillary qubits.

The ancillary qubits under utilization of `CNOT` gates allow the comparison of the quantum information that is present in the redundant qubits. Thus, the following table shows the possible outcomes of the error syndrome which shows whether there was an error and even where it happened.

`00`	$\Rightarrow$	No error happened
`01`	$\Rightarrow$	Error happened in q_0
`10`	$\Rightarrow$	Error happened in q_2
`11`	$\Rightarrow$	Error happened in q_1

Based on the results we are able to correct the errors.

3. Compared to classical computing where only a bit-flip error is possible, qubits are additionally prone to phase-flips and even worse to a number of those errors. In the process of a quantum computation noise may apply multiple small rotations on the state (Benenti et al., 2004, p. 460). Benenti et al. (2004, p. 465) show that the same method for correcting bit-flips can be used to detect phase-flips when we transform from the $\{|0\rangle, |1\rangle\}$ computational basis to $\{|+\rangle, |-\rangle\}$ as a bit-flip becomes a phase-flip and vice versa.

Since in this paper we will attempt to correct bit-flip errors, there is nothing more than this to understand how it works.

2.2 | Quantum Fourier Transform

In the course of this paper we try to correct errors in a quantum Fourier transform (*QFT*) circuit that may happen due to noise and decoherence in current quantum computers. Therefore, it is certainly advantageous to repeat the underlying concepts.

The QFT maps a quantum state $|\psi\rangle$ on another quantum state $|\alpha\rangle$ with the following definition (Weinstein, Pravia, Fortunato, Lloyd, & Cory, 2001, p. 1).

$$QFT_M|\psi\rangle \rightarrow \frac{1}{\sqrt{M}} \sum_{k=0}^{M-1} e^{2\pi i a k/M} |\alpha\rangle$$

This can be expressed as a unitary matrix which can be decomposed into several quantum gates (Weinstein et al., 2001, p. 2). For example the one-qubit QFT is just the Hadamard gate (The Qiskit Team, 2020b).

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

Another example is the two-qubit QFT_4 taken from Weinstein et al. (2001, p. 2).

$$QFT_4 = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

That matrix can be decomposed into the circuit shown in figure 3 (The Qiskit Team, 2020b).

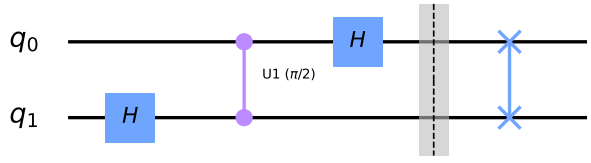

FIGURE 3 Circuit for the two-qubit QFT QFT_4.

All those matrices and circuits follow the same pattern (Yin, 2020):

$$QFT_M = \frac{1}{\sqrt{M}} \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \dots & \omega^{M-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \dots & \omega^{2M-2} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \dots & \omega^{3M-3} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & \omega^{M-1} & \omega^{2M-2} & \omega^{3M-3} & \dots & \omega^{(M-1)(M-1)} \end{bmatrix}$$

where $\omega = e^{\frac{2\pi i}{M}}$. Even a matrix of any size can be broken down into a circuit as shown in Figure 4. Note that the shown controlled `R` gate is really just a rotation around the Z-axis with angle $\frac{\pi}{2^{k-1}}$.

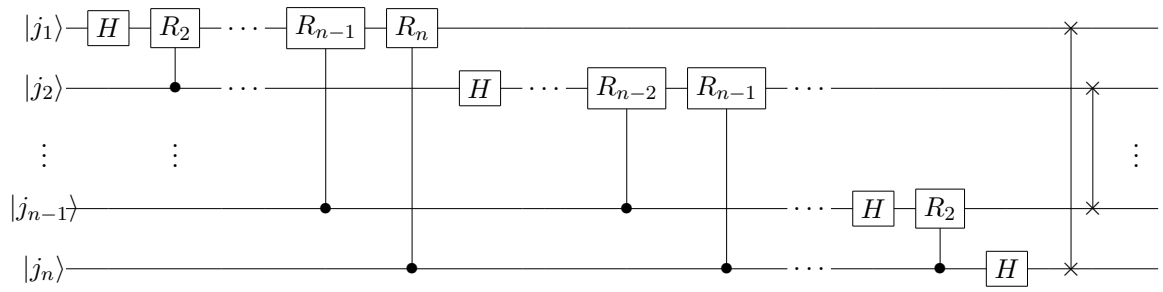

FIGURE 4 QFT_M circuit taken from Van Gael (2005)

Now we have enough information at hand to realize such a circuit with the tools and methods introduced and used in the following sections.

3 | RELATED WORK

In the last section some basic concepts about quantum error correction were introduced. For this paper we do not need more. But there is certainly more about that comparatively new topic in the quantum domain. This section will provide an overview of past and current developments in QEC.

It all began the last 90s where the noise has been discretized into discrete error models, namely bit- and phase-flip errors (Li, 2020, p. 46999). Shor has developed a quantum error correction code (QECC) that successfully applies the previously reviewed bit-flip and phase-flip error circuits to protect against any type of error with 9 qubits (Li, 2020, p. 46999). That event was followed by Calderbank, Shor and Steane each constructing the so-called CSS codes which were adopted from the theory of classical linear codes (Preskill, 2019, Chapter: Quantum Error Correction, p. 24). It was discovered that the minimum amount of qubits needed to protect a single qubits information against any error is five (Li, 2020, p. 46999).

Since there seems to be no improvement to protect against any error, scientists moved on to protect against specific errors instead, thereby reducing the amount of qubits even further (Li, 2020, p. 46999). That is especially useful when the error-source is known and thus can be excluded.

Discrete error models are not like the real world and rely on idealized scenarios. For example they simplify that errors occur on qubits independently, thus there is currently increasing effort in implementing continuous error models. (Li, 2020, p. 46999).

4 | METHODS AND TOOLS

As a reminder, we will try to correct bit-flip errors of a quantum Fourier transform circuit. Therefore, some tools and methods are selected and presented in this section.

4.1 | Qiskit

Since we are already familiar with the quantum computing software development framework `Qiskit`, it is an obvious choice. Its tight coupling with IBM Quantum Experience and thus accessible quantum devices allow testing quantum circuits in a real environment (*IBM Q Account*, 2020). Besides that the framework offers a simulator capable of testing the created code as well as adding a noise model to imitate real quantum computer behavior, which we want to fight using our quantum error correction solution (*Qiskit - Noise Models*, 2020a).

The specific `Qiskit` version used in this paper is listed in the appendix A.2.

4.2 | Python

Qiskit is a library for the programming language Python. Thus we have to use it. Additionally, Pythons popularity soared in the last years especially in scientific areas, making it one of the most popular programming languages (*Stack Overflow Developer Survey*, 2020). Thus, most people reading this paper should easily be capable of understanding the included code snippets, improving accessibility.

The precise version of the language is `Python 3.8.2` which is used throughout this paper. There is no other reason for choosing that exact version other than `Qiskit` is supporting it and it is an up-to-date version of the language at the time of writing.

4.3 | JupyterLab

`JupyterLab` is a web-based user interface for Project Jupyter (*JupyterLab Documentation*, 2020). It is an open-source project to support interactive data science and scientific computing (*Project Jupyter*, 2020) One of its components called *Jupyter Notebooks* will be used to quickly write and test Python code with additional support for LaTeX and Markdown to document that code more clearly (*JupyterLab Overview*, 2020).

In addition, the code is also included in the annex to this document in order to ease the reproduction of the work of this paper.

4.4 | QFT circuit

So far we have not discussed what specific circuit is planned to be implemented. The QFT is transforming from the computational basis in the Fourier basis where we store numbers using various rotations around the Z-axis of the individual qubits (The Qiskit Team, 2020b). When measuring the result we would just receive 0 or 1 with probability $\frac{1}{2}$ (neglecting noise errors). There is the quantum phase estimation algorithm that would estimate that phase, but for the sake of just trying to run error correction we will just transform in the computational basis again using the *inverse* QFT circuit. Afterwards we should be able to measure the input again. Thus, we can easily validate whether errors happened.

5 | IMPLEMENTATION OF A QECC

In this section we are finally using the tools and methods presented in the last section to apply the quantum error correction (bit-flips) for the quantum Fourier transform.

5.1 | Implementing the quantum Fourier transform circuit

First and foremost it would make certainly sense to build the proposed QFT circuit and see if it actually works on a simulator and a real quantum device. Recalling the basic concepts chapter we have already seen a generic circuit in figure 4 that can be rebuilt using Qiskit. The controlled `R`-gate is a rotation around the Z-axis which can be realized in `Qiskit` using a controlled `U1` gate (*IBM Quantum Experience - Docs and Resources*, 2020).

Before using *Qiskit* and building the circuit we need to set up the code by listing the needed imports in appendix A.1 for *Python*. Afterwards we can follow the circuit in figure 4 and implement it iteratively for any size in Qiskit as seen in appendix A.3. That allows us creating an arbitrarily big circuit. One example circuit graph for 4 input qubits is created and shown in figure 5. Note that the circuit is mirrored vertically compared to the reference circuit, as Qiskit is ordering the qubits differently compared to most resources (*Getting Started with Qiskit*, 2020).

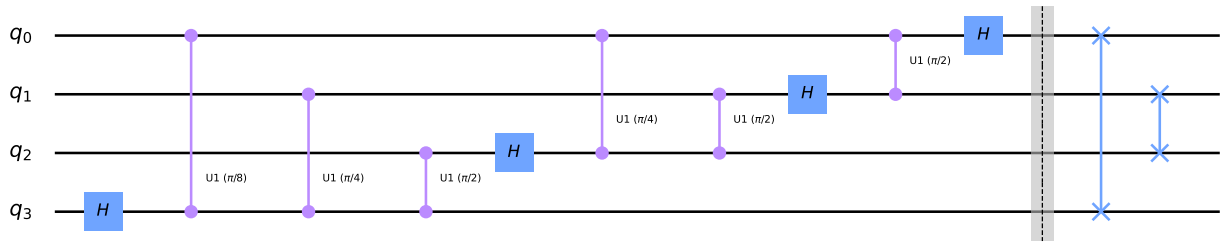

FIGURE 5 Qiskit 4-qubit QFT circuit graph

To get the inverse of that circuit *Qiskit* offers the method `inverse()` which can be applied on any `QuantumCircuit`. It is applied in the function displayed in appendix A.4 and delivers together with the non-inversed QFT the circuit depicted in figure 6.

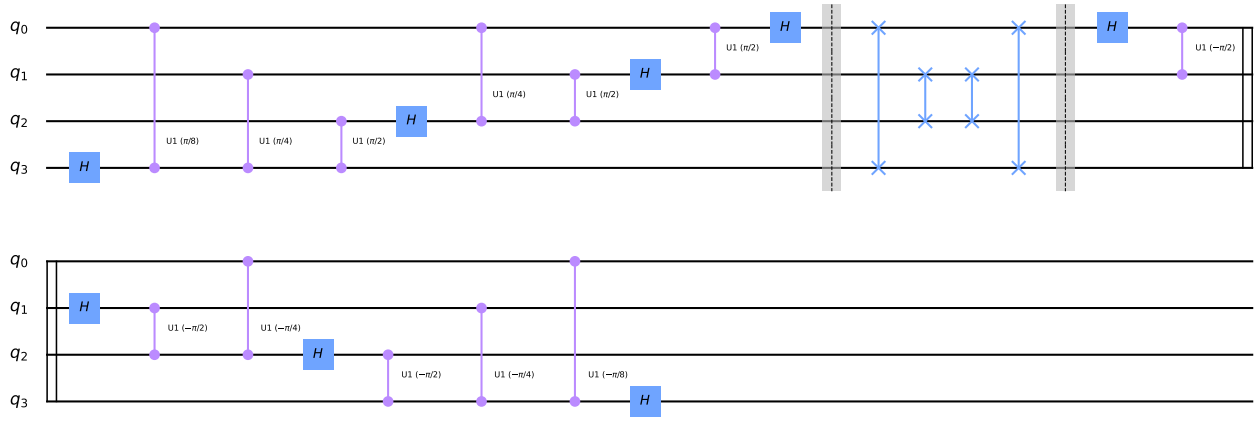

FIGURE 6 Qiskit 4-qubit QFT + inverse QFT circuit

The last step to have a fully functional circuit, is to encode a number to the qubits that serve as the input. A method in appendix A.5 is written that encodes a number to a binary string and later to another quantum circuit that is scaled to fit the amount of needed bits to represent the passed number. For example the decimal integer number 12 is encoded to the binary string `1100` and therefore needs the circuit to have 4 input qubits. Additionally, the same code in the appendix shows a function to measure the qubits to classical bits at the end of the circuit to get the circuits result as binary string again. That result should ideally match the input when no error occurs.

The full circuit for the encoded number 6 (`110`) is shown in figure 7.

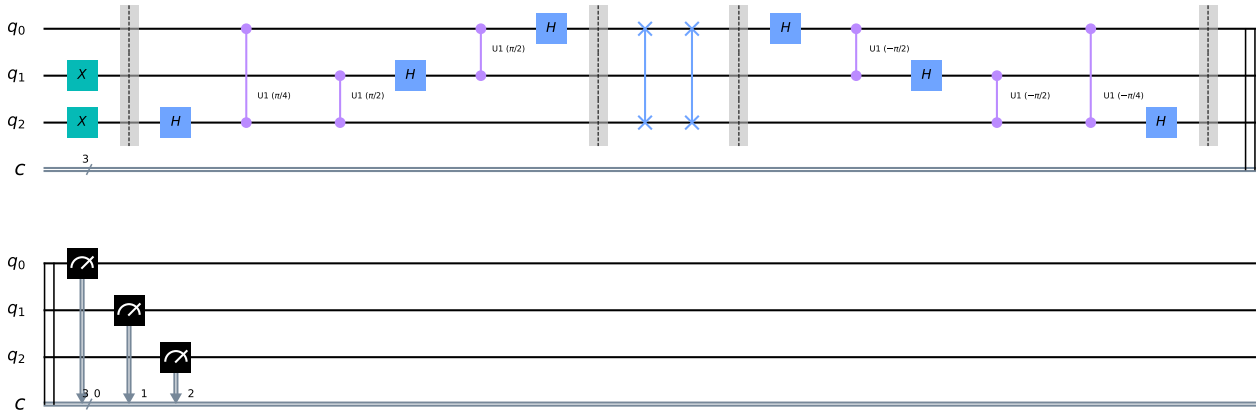

FIGURE 7 Qiskit 3-qubit circuit used in this paper

For the sake of completeness, the source code for generating the above image is listed in Annex A.6.

Testing

Now that we have the circuit we should be able to test it using the Qiskit simulator and a real quantum device. The simulator testing is done using the code from A.7 from which figure 8 is created. The histogram is showing 1024 simulated results from the circuit. Since we have not configured the simulator to simulate noise the result is the same (`110`) for all shots as expected.

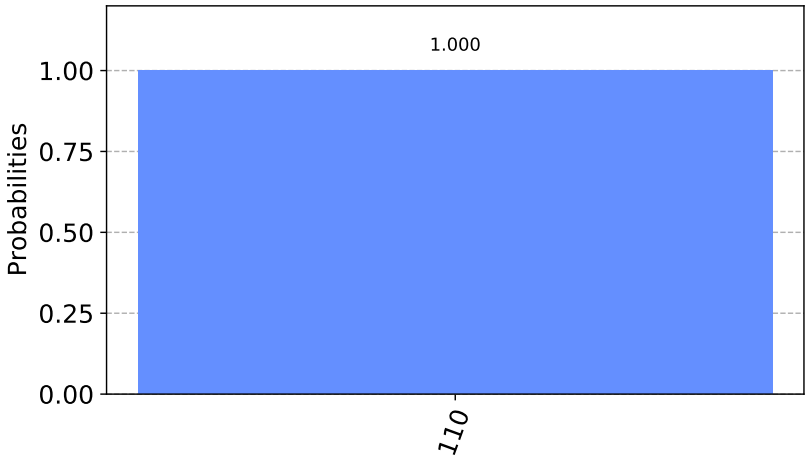

FIGURE 8 Simulator result without noise for the full proposed QFT circuit with 3 qubits

We are able to simulate noise using the `noise_model` parameter of Qiskits `execute` function. The Qiskit Team (2020a) showed a function to create a noise model from which we derived the one listed in appendix A.8. It anticipated two parameters:

1. The probability $p_{error_{measurement}}$ that the measurement leads to a bit-flip
2. The probability $p_{error_{gate}}$ to replace the state of a qubit with a random state

Another way of creating a noise model is by automatically generating one from the properties of a real quantum device using the `NoiseModel.from_backend()` method (*Qiskit - Noise Models*, 2020b). In the first test we'll settle with manually specifying the probabilities to both 5%. Later a noise model is generated from a real quantum backend followed by executing the circuit on the real device.

The code for generating the simulated results as shown in the histogram in figure 9 is listed in appendix A.9. It is evident that there is indeed noise as we do no more measure the correct result `110` with 100% probability.

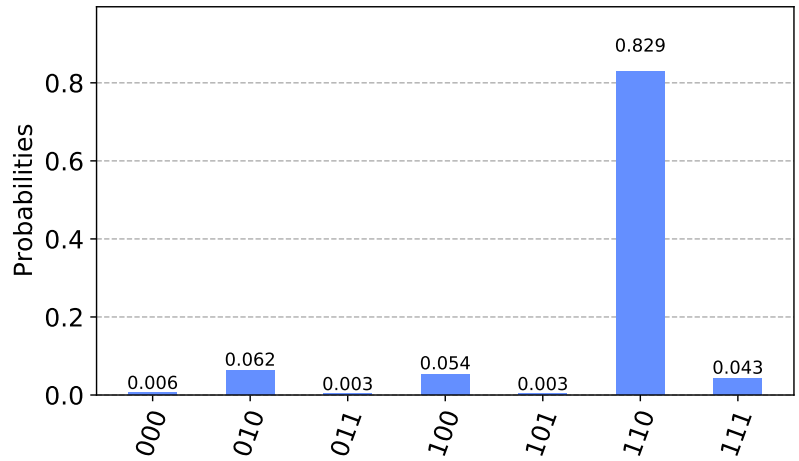

FIGURE 9 Simulator result **with** noise for the full proposed QFT circuit with 3 qubits

Now we try generating a noise model from a real quantum backend. The code to do that is as always placed in the appendix A.10. It is choosing the least busy quantum device. In the case of the resulting figure 10 it has chosen a backend called `ibmqx2`. If we compare it with the previous histogram, we find that it has produced a noise model that is not as noisy as the 5% gate and measurement errors we selected earlier.

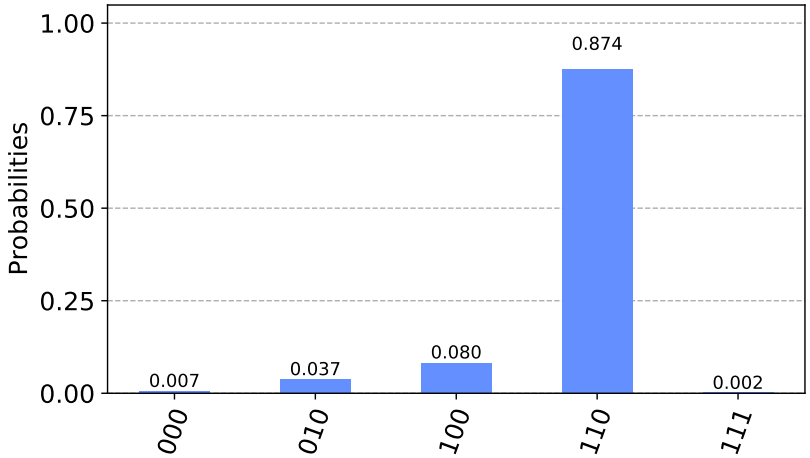

FIGURE 10 Real quantum device result for the full proposed QFT circuit with 3 qubits

The resulting histogram for the test with a real quantum device, which can be generated using the code from appendix A.11, is depicted in figure 11. Another result the code is outputting is the quantum backend name on which the circuit has been executed. In this case the name is `ibmqx2`. That happens to be the same that Qiskit has generated a noise model for. Comparing it to the actual results we see a huge difference as the actual noise seems to be greater.

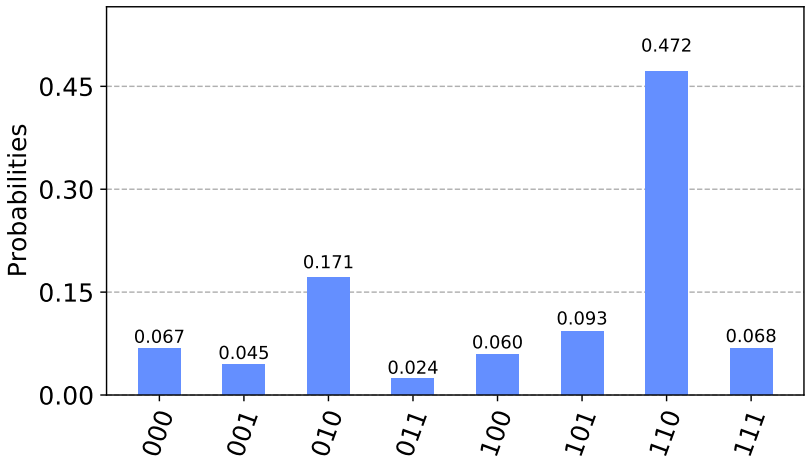

FIGURE 11 Real quantum device result for the full proposed QFT circuit with 3 qubits

5.2 | QFT circuit changes for error-correction

We now are applying some changes in order to correct bit-flip errors in the circuit. A good result would be a higher percentage of the anticipated result in the resulting histogram compared to the ones listed above.

To keep the circuit quantitatively small, we will stick with the 3-qubit QFT circuit and repeat it 3 times. That means we have to stack the previously shown circuit vertically and add some ancillary qubits. The functions in appendix B.1 are rewritten versions of the previously shown ones that allow passing a parameter specifying a quantum circuit object to apply the gates on as well a vertical `offset` to apply them in the passed circuit. This is done to have the code reusable, small and still comprehensible.

Additionally those functions are used in another function `prepare_rc_qec_qft_circuit` which can be seen in appendix B.2. It is building the whole circuit for a specified input number and repetition count. An example for 2 input qubits and 3 repetitions is displayed in figure 12 but it really can be any size. As one can see we need 4 ancillary qubits for correcting bit flips at the end. For 3 input qubits and 3 repetitions it would be 6 ancillary qubits. Thus we conclude a formula for determining the needed ancillary qubit count $c_{ancilla}$ for a given input qubit size c_{qubits} and repetitions c_{rep}:

$$c_{ancilla}(c_{qubits}, c_{rep}) = (c_{rep} - 1) * c_{qubits}$$

The ancilla qubits are needed per qubit in the QFT circuit. Since we repeat the QFT circuit multiple times we need more ancilla qubits. For 3 input qubits and 3 repetitions we need two ancilla qubits per qubit in a single QFT circuit suming up to 6 since they compare for example

all three first qubits of all repetitions as seen in figure 13.

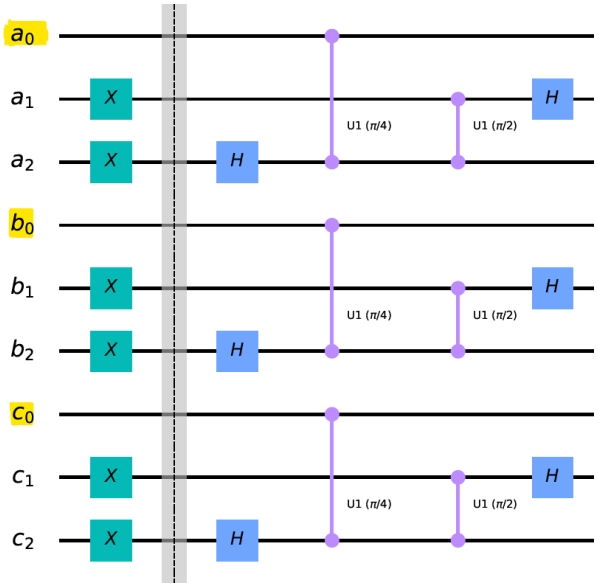

FIGURE 13 Snippet from the error correcting circuit with 3 input qubits and 3 repetitions. The first qubits of every repetition that are essentially duplicates are highlighted. For those we introduce ancillary qubits in order to compare them at the end.

The last code snipped can be looked up in appendix B.3 and is decoding a resulting bit string from the circuit. That means correcting the error when the ancillary bits indicate one. As an example we decode the possible result string `10 01 00 010 110 010`. The first three bit strings `10`, `01` and `00` are the measured results from the ancillary qubits. Those indicate whether an error happened. The concrete meaning is summarized in the following table.

`00`	$\Rightarrow$	No error happened
`01`	$\Rightarrow$	Error happened in the first QFT register (Qubit a_0, $a_1 or a_2$)
`10`	$\Rightarrow$	Error happened in the last QFT register (Qubit c_0, $c_1 or c_2$)
`11`	$\Rightarrow$	Error happened in the middle QFT register (Qubit b_0, $b_1 or b_2$)

Additionally to that, the position of the ancillary bit strings indicate the exact qubit the error occurred on. So in the example the error `10` occurred on the third, `01` on the second and `00` (No error) on the first qubit of the specified register.

`10`	$\Rightarrow$	Error at c_2
`01`	$\Rightarrow$	Error at a_1
`00`	$\Rightarrow$	No error at a_0, $b_0 or c_0$

Since we are trying to correct bit-flips we can just flip the resulting bits in the remaining 3 QFT result strings `010` (Register c), `110` (Register b) and `010` (Register a). According to the ancillary qubits we have to flip c_2 and a_1 which results in the three corrected strings `110` (Register c),

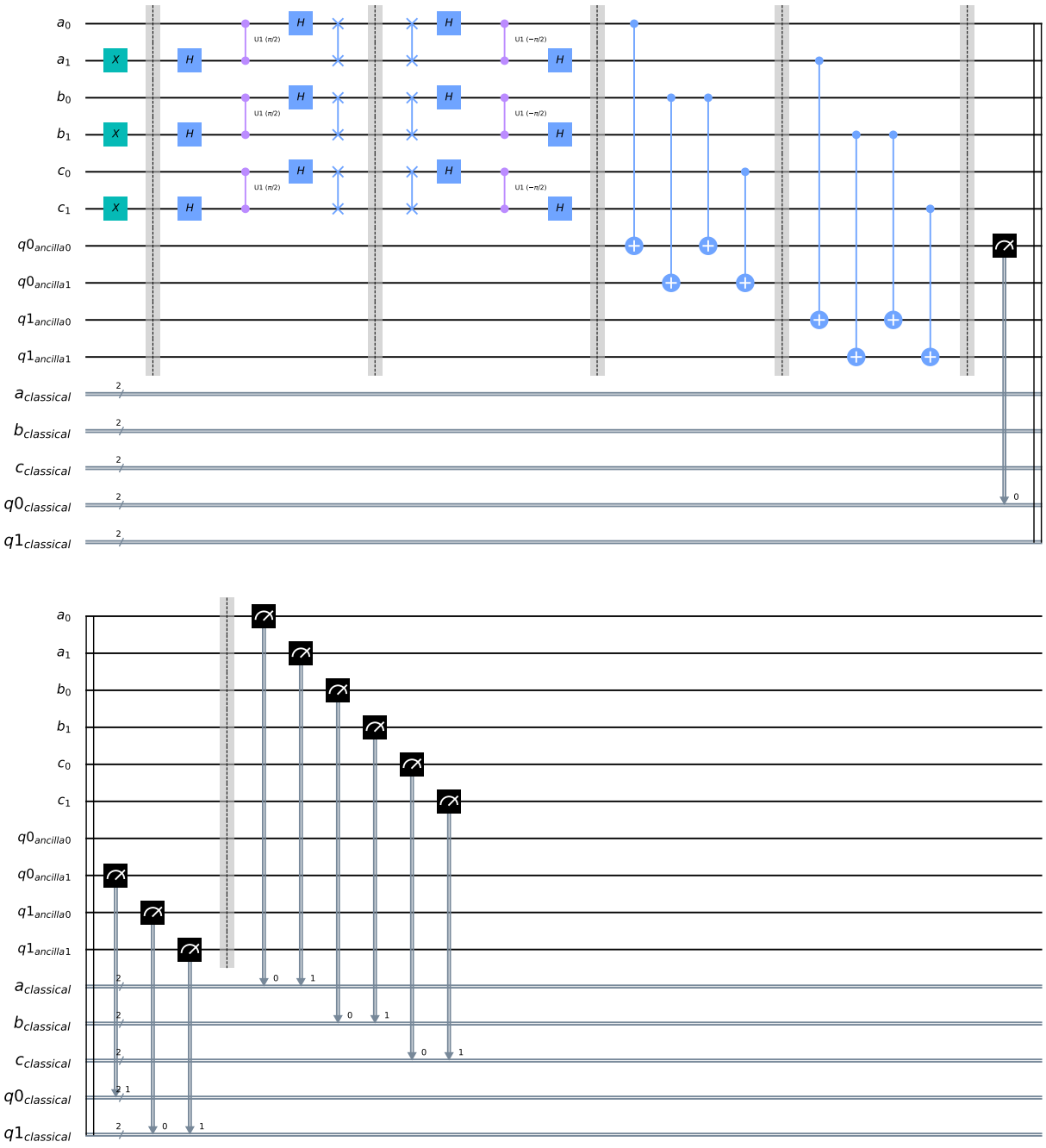

FIGURE 12 Error corrected QFT circuit with 2 qubits input and 3 repetitions

110 (Register *b*) and 000 (Register *a*). When merging that three strings we get 110 as **decoded** result. In summary that is exactly what the decoding function does.

Testing

As quick reminder: We now have functions to generate a error correcting circuit via repetition of the initial QFT circuit we proposed. Additionally, we have introduced the resulting bit string format which we will get from the Qiskit simulator as well as the real quantum backends.

This section aims at testing a 3 input qubit, 3 repetitions circuit with the encoded number 6 using the simulator as well as real quantum devices. In the first step we will use the Simulator with manually chosen and generated noise models, followed by some real quantum backends.

Utilizing the code displayed in appendix B.4 we are able to generate histograms showing the decoded results for multiple execution shots of the quantum circuit. The histograms are plotted on the same result set but with different error correcting techniques applied:

- **No error correction**: No error correction is applied on the result set and all the results from the individual QFT circuit repetitions are taken into account.
- **Only merging results**: The results from the individual QFT circuits are merged, but the ancillary bit result is not taken into account.
- **Error correction**: The results from the individual QFT circuits are first error corrected using the ancillary bit results and then merged.
- **Only merge results without error**: Only results from the individual QFT circuits are merged that have not been flagged as wrong by the ancillary bit results.

Figure 14 is visualizing the results when using the simulator with a noise model with both probability values set to 5%.

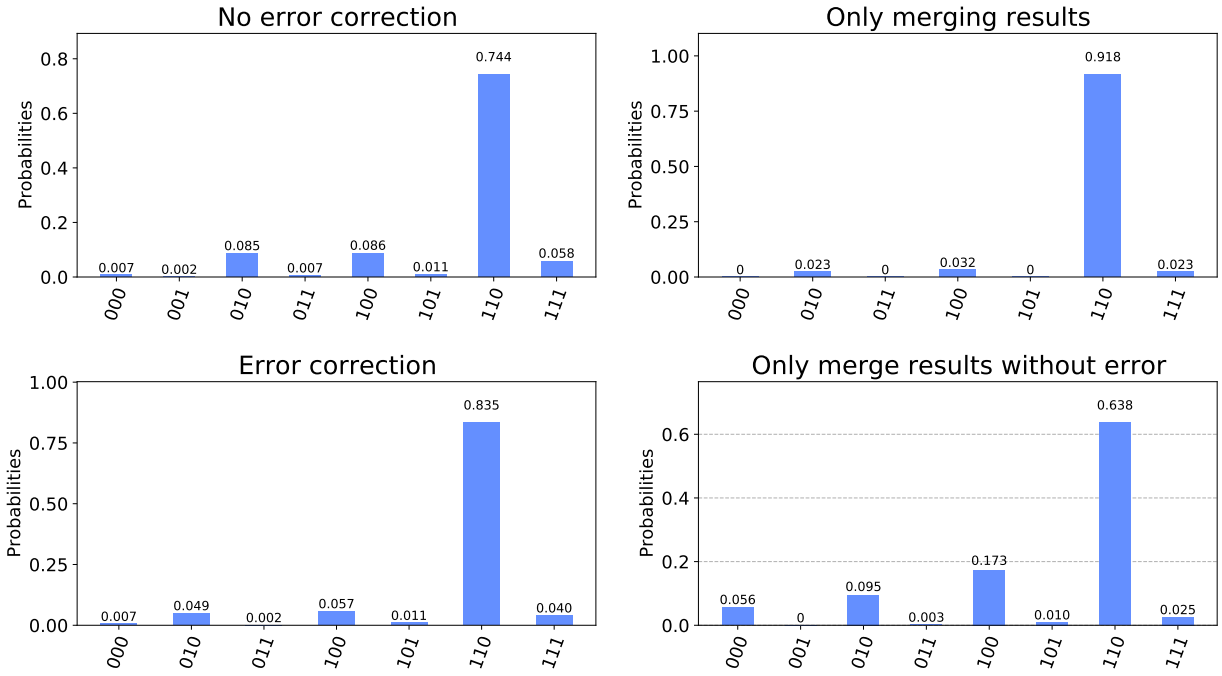

FIGURE 14 Result distributions of 1024 shots of the proposed error correcting circuit.

The histograms show that the error correction is indeed working by increasing the correct result ratio from 0.744% to 0.835%. Interestingly only merging the results without error correction seems to work even better.

When generating a noise model using Qiskit from the backend `ibmq_16_melbourne` we get the results from figure 15.

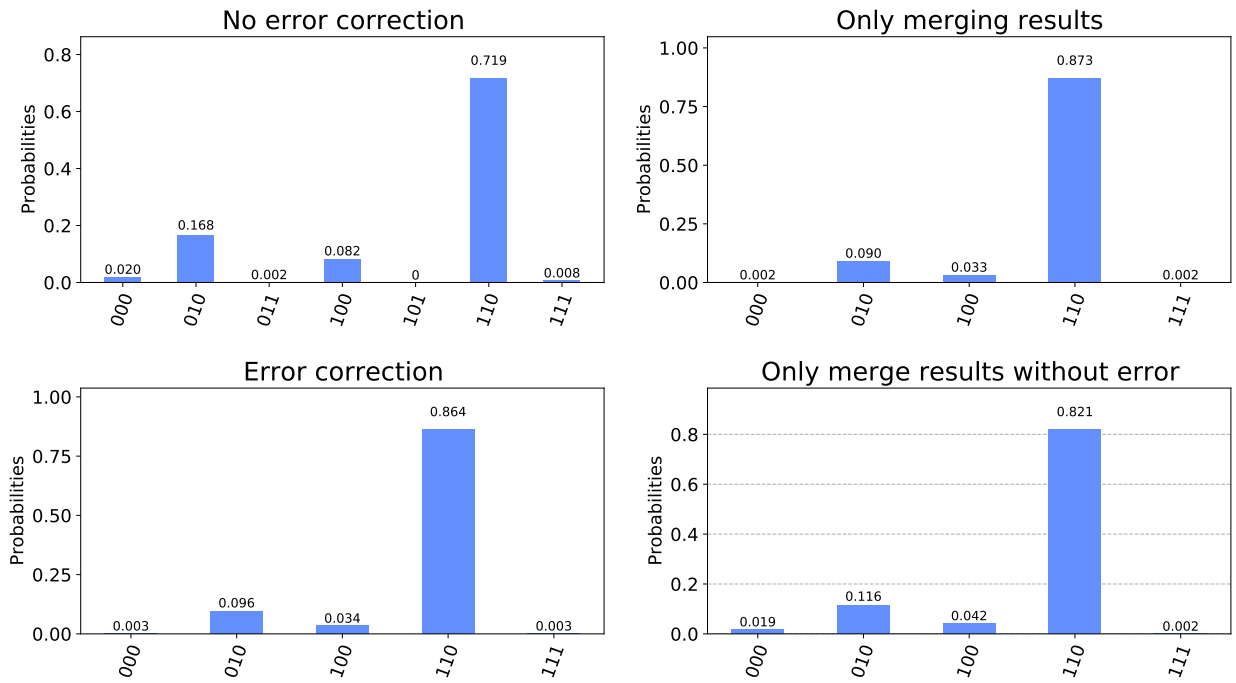

FIGURE 15 Result distributions of 1024 shots of the proposed error correcting circuit using a generated noise model from the backend `ibmq_16_melbourne`.

Again the results show an improvement when using error correction but with less difference between the "Error correction" and "Only merging results" plots compared to the manually configured noise model.

Running on the real quantum backend `ibmq_16_melbourne` we get the results displayed in the below plots of figure 16. Unfortunately that backend seems to be too noisy and results in mostly random results.

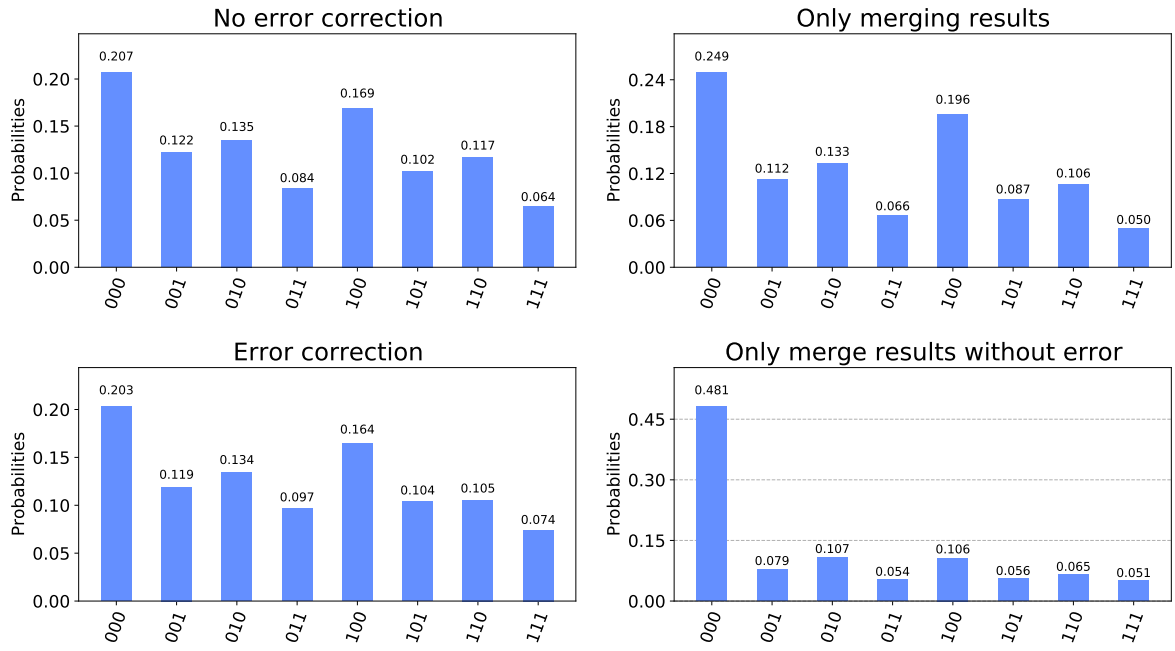

FIGURE 16 Result distributions of 1024 shots of the proposed error correcting circuit using the backend `ibmq_16_melbourne`.

Additionally the circuit has been executed on two more sophisticated quantum devices "Paris" (27 qubits) and "Johannesburg" (20 qubits) leading to the results displayed in figure 17 and 18 (*IBMQ - Available Systems*, 2020).

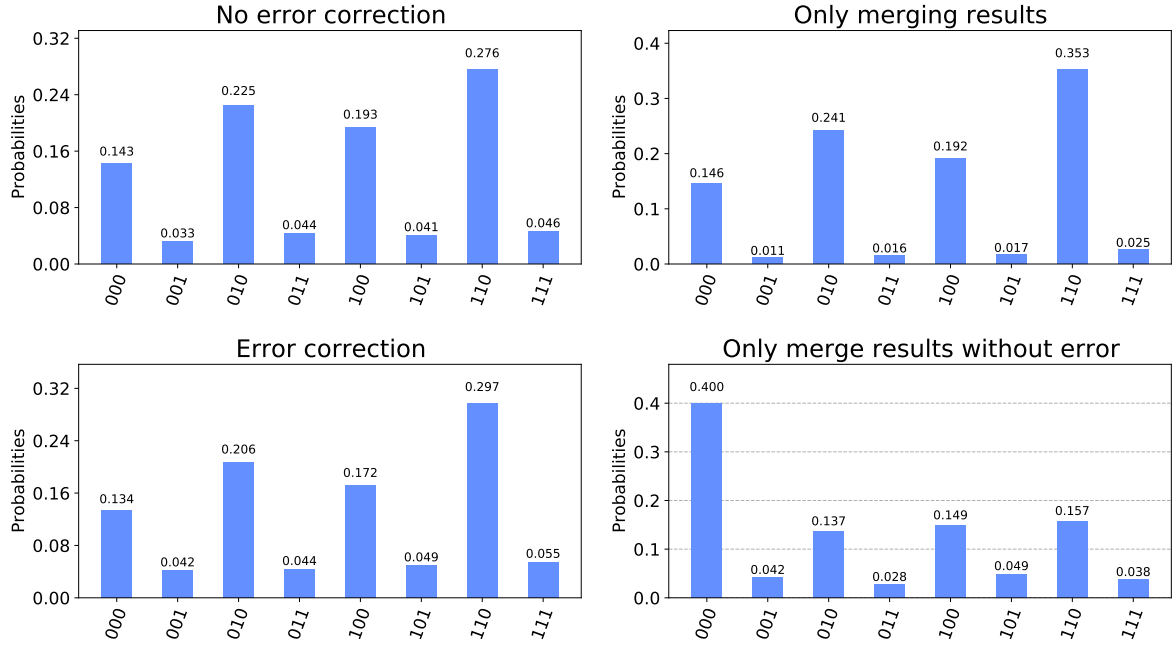

FIGURE 17 Result distributions of the proposed error correcting circuit using the backend "Paris" with 27 qubits.

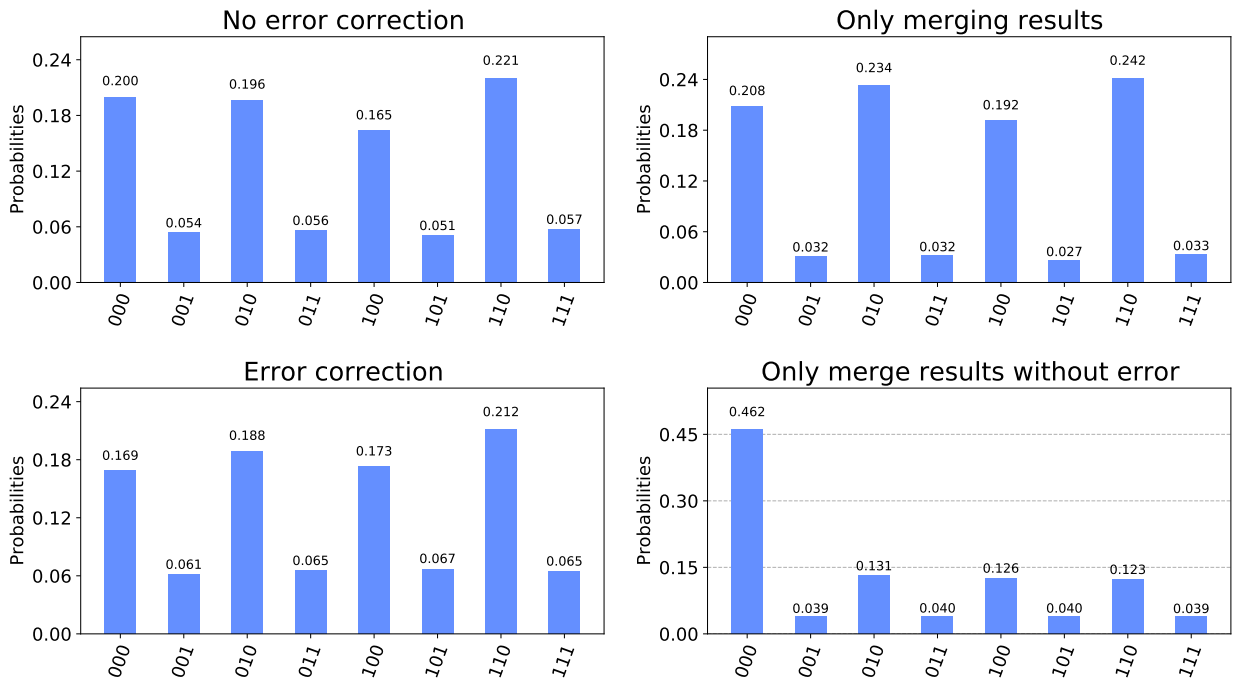

FIGURE 18 Result distributions of the proposed error correcting circuit using the backend "Johannesburg" with 20 qubits.

While there is still a lot of noise observable - at least for the proposed circuit - we can definitely see that error correction is working. Especially on "Paris" where "Only merging results" provides a jump from 0.276% to 0.353%. "Paris" is also the only one of the tested real quantum backends where "Error corrections" does show an improvement compared to "No error correction".

6 | SUMMARY

After having observed some interesting results we want to quickly sum up the contents of the paper followed by some proposals for further studies.

6.1 | Conclusion

We have been implementing a quantum circuit trying to error correct bit-flips over several repeated QFT circuits using the Framework Qiskit. While the Qiskit simulator showed good results for manual as well as automatically configured noise models, current IBMQ backends seem to be still too noisy for the proposed circuit. Additionally, we are able to say that the generated noise model from Qiskit is not estimating properly - at least for the proposed circuit.

Tannu and Qureshi (2018, p. 1) state that we are currently dealing with *Noisy Intermediate Scale Quantum computers* that do not have the capacity to utilize QEC due to the limited amount of qubits available. That said we could experience a little improvement with the "Paris" backend although we were only correcting bit-flip errors. They are correct as having proper QEC codes requires a tremendous bigger amount of qubits than we have available today.

Additionally, Tannu and Qureshi (2018, p. 3) write that the overall error rate is usually dominated by the gate errors from which we have a lot in the proposed circuit. Especially the `CNOT` gate seems to have high error rates which we utilize extensively in the circuit to "read" the current qubit state to the ancillary qubits (Tannu & Qureshi, 2018, p. 3). That might be another factor that leads to the observed strong noisy results.

6.2 | Proposals for further studies

In summary we find several proposals that might be worth studying in the future:

- Since we rely on `CNOT` gates it would be interesting to remove them from the circuit and afterwards observe what happens to the "Only merged results" histogram.
- Qiskit allows us to directly select and assign specific qubits of a real quantum backend that we want to use. That way we might be able to use only qubits that have low error rates.

REFERENCES

Benenti, G., Casati, G., & Strini, G. (2004). *Principles of quantum computation and information: Basic tools and special topics*. World Scientific.

de Brito, D. B., do Nascimento, J. C., & Ramos, R. V. (2008). Quantum communication with polarization-encoded qubit using quantum error correction. *IEEE Journal of Quantum Electronics*, *44*(2), 113-118.

Getting Started with Qiskit. (2020, Jun). Retrieved from https://qiskit.org/documentation/tutorials/circuits/1_getting_started_with_qiskit.html ([Online; accessed 14. Jun. 2020])

IBM Q Account. (2020, Jun). Retrieved from https://qiskit.org/ibmqaccount ([Online; accessed 14. Jun. 2020])

IBMQ - Available Systems. (2020, Jun). Retrieved from https://quantum-computing.ibm.com/docs/cloud/backends/systems/#system-backends-systems-available ([Online; accessed 16. Jun. 2020])

IBM Quantum Experience. (2020, Jun). Retrieved from https://quantum-computing.ibm.com ([Online; accessed 12. Jun. 2020])

IBM Quantum Experience - Docs and Resources. (2020, Jun). Retrieved from https://quantum-computing.ibm.com/docs/circ-comp/q-gates#u1-gate ([Online; accessed 14. Jun. 2020])

JupyterLab Documentation. (2020, Jun). Retrieved from https://jupyterlab.readthedocs.io/en/stable ([Online; accessed 14. Jun. 2020])

JupyterLab Overview. (2020, Jun). Retrieved from https://jupyterlab.readthedocs.io/en/stable/getting_started/overview.html ([Online; accessed 14. Jun. 2020])

Li, J. (2020). Some progress on quantum error correction for discrete and continuous error models. *IEEE Access*, *8*, 46998-47012.

Matuschak, A., & Nielsen, M. (2019). *Quantum computing for the very curious - Quantum wires*. Retrieved from https://quantum.country/qcvc#quantum-wires ([Online; accessed 12. Jun. 2020])

Milburn, G. J., Sarovar, M., & Ahn, C. (2004). Quantum control and quantum error correction. In *2004 5th asian control conference (ieee cat. no.04ex904)* (Vol. 1, p. 33-41 Vol.1).

Preskill, J. (2019, Sep). *Quantum Compution Course*. Retrieved from http://theory.caltech.edu/people/preskill/ph229 ([Online; accessed 13. Jun. 2020])

Project Jupyter. (2020, May). Retrieved from https://jupyter.org/about ([Online; accessed 14. Jun. 2020])

Qiskit. (2020, Jun). Retrieved from https://qiskit.org ([Online; accessed 12. Jun. 2020])

Qiskit - Noise Models. (2020a, Jun). Retrieved from https://qiskit.org/documentation/apidoc/aer_noise.html ([Online; accessed 14. Jun. 2020])

Qiskit - Noise Models. (2020b, Jun). Retrieved from https://qiskit.org/documentation/apidoc/aer_noise.html ([Online; accessed 14. Jun. 2020])

Shor, P. W. (1994). Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science* (p. 124-134).

Stack Overflow Developer Survey. (2020, Jun). Retrieved from https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages-all-respondents ([Online; accessed 14. Jun. 2020])

Tannu, S. S., & Qureshi, M. K. (2018). A case for variability-aware policies for nisq-era quantum computers. *arXiv preprint arXiv:1805.10224*.

The Qiskit Team. (2020a, Jun). *Introduction to Quantum Error Correction using Repetition Codes*. Retrieved from https://qiskit.org/textbook/ch-quantum-hardware/error-correction-repetition-code.html ([Online; accessed 14. Jun. 2020])

The Qiskit Team. (2020b, Jun). *Quantum Fourier Transform - Qiskit Textbook*. Retrieved from https://qiskit.org/textbook/ch-algorithms/quantum-fourier-transform.html ([Online; accessed 13. Jun. 2020])

Van Gael, J. (2005). *The Role of Interference and Entanglement in Quantum Computing*. Retrieved from http://pages.cs.wisc.edu/~dieter/Papers/vangael-thesis.pdf ([Online; accessed 13. Jun. 2020])

Weinstein, Y. S., Pravia, M. A., Fortunato, E. M., Lloyd, S., & Cory, D. G. (2001, Feb). Implementation of the quantum fourier transform. *Physical Review Letters*, *86*(9), 1889–1891. Retrieved from http://dx.doi.org/10.1103/PhysRevLett.86.1889 doi: 10.1103/physrevlett.86.1889

Yin, A. (2020, Jun). *Berkeley Quantum Computing*. Retrieved from https://www.academia.edu/16116703/Berkeley_Quantum_Computing ([Online; accessed 13. Jun. 2020])

A | QFT CIRCUIT WITH QISKIT

A.1 | Necessary imports

```
# Qiskit
from qiskit import __qiskit_version__
from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit
from qiskit import execute, Aer
from qiskit import IBMQ
from qiskit.visualization import plot_histogram
from qiskit.providers.ibmq import least_busy
from qiskit.providers.aer.noise import NoiseModel
from qiskit.providers.aer.noise.errors import pauli_error,
                                              depolarizing_error

# Plotting library
import matplotlib.pyplot as plt

# Math library
import numpy as np
```

A.2 | Qiskit version

The used Qiskit version can be received by calling the following.

```
print(__qiskit_version__)

>>> {'qiskit-terra': '0.14.1',
'qiskit-aer': '0.5.2',
'qiskit-ignis': '0.3.0',
'qiskit-ibmq-provider': '0.7.2',
'qiskit-aqua': '0.7.1',
'qiskit': '0.19.3'}
```

A.3 | QFT function

```
def qft(n):
    """
    Created a quantum Fourier transform on the passed quantum circuit.
    Pass [n] as the number of qubits the circuit should be able to deal with
                                                    .
    """

    qc = QuantumCircuit(n)
```

```
    # Create core QFT circuit
    for i in range(n - 1, -1, -1):
        # Apply Hadamard gate on the i'th qubit (most significant qubit)
        qc.h(i)

    # Apply controlled rotation around the Z-axis (CU1-Gate)
    for j in range(i):
        angle = np.pi / 2 ** (i - j)

        qc.cu1(theta=angle, control_qubit=j, target_qubit=i)

        qc.barrier() # Barrier for better readability of the resulting
                                                circuit graph

    # Add ending swaps
    for i in range(n // 2):
        qc.swap(i, n - 1 - i)

    return qc
```

A.4 | Inverse QFT function

```
def inverse_qft(n):
    """
    Create a inverse quantum Fourier transform on the passed quantum circuit
                                                .
    Pass [n] as the number of qubits the circuit should be able to deal with
                                                .
    """

    return qft(n).inverse()
```

A.5 | Encoding and decoding number in a circuit

```
def encode_num(num):
    """
    Encode the passed decimal integer number in an quantum circuit.
    """

    remainders = []
    while num > 0:
        remainder = num % 2
        num = num // 2

        remainders.append(remainder)
```

```
    qc = QuantumCircuit(len(remainders))

    for i in range(len(remainders)):
        if remainders[i] == 1:
            qc.x(i)

    return qc


def decode_num(n):
    qc = QuantumCircuit(n, n)

    qc.barrier()

    qc.measure(range(n), range(n))

    return qc
```

A.6 | Code to generate the full QFT circuit used in the paper

```
num_to_encode = 6

# Encode number to send through circuit
enc_qc = encode_num(num_to_encode)
enc_qc.barrier()

n = enc_qc.num_qubits

# Create QFT circuit
qft_qc = qft(n)

# Create inverse QFT
inv_qft_qc = inverse_qft(n)

# Decode number sent through circuit
dec_qc = decode_num(n)


qc = (enc_qc + qft_qc + inv_qft_qc + dec_qc)

fig, axes = plt.subplots(1)

qc.draw('mpl', ax = axes)

fig.savefig('out/full-qft-3-qubit-circuit.pdf', transparent=True)
```

A.7 | Testing circuit on a simulator without noise

```
qasm_simulator = Aer.get_backend('qasm_simulator')

job = execute(qc, backend=qasm_simulator, shots=1024)

result = job.result()
counts = result.get_counts()

fig, axes = plt.subplots(1)

plot_histogram(counts, ax = axes)

fig.savefig('out/test-histogram.pdf', transparent=True)
```

A.8 | Noise model function

```
def get_noise(p_measure_error, p_gate_error):
    error_measure = pauli_error([('X', p_measure_error), ('I', 1 -
                                        p_measure_error)])
    error_gate1 = depolarizing_error(p_gate_error, 1)
    error_gate2 = error_gate1.tensor(error_gate1)

    noise_model = NoiseModel()
    noise_model.add_all_qubit_quantum_error(error_measure, "measure") #
                                        measurement error is applied to
                                        measurements
    noise_model.add_all_qubit_quantum_error(error_gate1, ["x"]) # single
                                        qubit gate error is applied to x
                                        gates
    noise_model.add_all_qubit_quantum_error(error_gate2, ["cx"]) # two qubit
                                         gate error is applied to cx gates

    return noise_model
```

A.9 | Testing circuit on a simulator with noise

```
job = execute(qc, backend=qasm_simulator, shots=1024, noise_model=get_noise(
                                    0.05, 0.05))

result = job.result()
counts = result.get_counts()

fig, axes = plt.subplots(1, figsize=(7,4))

plot_histogram(counts, ax = axes)
```

```
fig.savefig('out/test-histogram-with-noise.pdf', transparent=True)
```

A.10 | Testing circuit with generated noise model

```
# Get backend
IBMQ.load_account()

# Get least busy quantum computer
provider = IBMQ.get_provider(hub='ibm-q')
real_backend = least_busy(
    provider.backends(
        filters = lambda b: b.configuration().n_qubits >= qc.num_qubits
            and not b.configuration().simulator and b.status().operational==
                                                    True
    )
)

print(real_backend.name())

noise_model = NoiseModel.from_backend(real_backend)

job = execute(qc, backend=qasm_simulator, shots=1024, noise_model=
                                        noise_model)

result = job.result()
counts = result.get_counts()

fig, axes = plt.subplots(1, figsize=(7,4))

plot_histogram(counts, ax = axes)

fig.savefig('out/test-histogram-generated-noise-model.pdf', transparent=True
                                        )
```

A.11 | Testing circuit on a real quantum device

```
IBMQ.load_account()

# Get least busy quantum computer
provider = IBMQ.get_provider(hub='ibm-q')
real_backend = least_busy(
    provider.backends(
        filters = lambda b: b.configuration().n_qubits >= qc.num_qubits
            and not b.configuration().simulator and b.status().operational==
                                                    True
```

```
    )
)

print(real_backend.name())

job = execute(qc, backend=real_backend, shots=1024)

result = job.result()
counts = result.get_counts()

fig, axes = plt.subplots(1, figsize=(7,4))

plot_histogram(counts, ax = axes)

fig.savefig('out/test-histogram-real-quantum-device.pdf', transparent=True)
```

B | ERROR CORRECTED CIRCUIT SOURCE CODE

B.1 | Modified QFT circuit functions

```
def to_bit_representation(num):
    """
    Convert the passed decimal integer number to a bit representation.
    For example 6 will be transformed to a list: [1, 1, 0].

    Parameters:
    [num]: A decimal integer number to encode to a bit representation

    Returns:
    The passed decimal integer number as a bit list
    """

    remainders = []
    while num > 0:
        remainder = num % 2
        num = num // 2

        remainders.append(remainder)

    return remainders


def encode_bits(bits, qc, offset=0):
    """
    Encode the passed decimal integer number given as bits list in the
                                            passed quantum circuit.

    Parameters:
    [bits]: Bit list (e. g. [1, 1, 0] for the decimal number 6) to encode
    [qc]: Quantum Circuit to encode the bits in
    [offset]: The offset to apply in the passed circuit
    """

    bit_count = len(bits)

    # Encode number in circuit
    for i in range(bit_count):
    if bits[i] == 1:
    qc.x(offset + i)


    def qft(size, circuit, offset=0):
    """
```

```
    Add quantum Fourier transform gates for the passed [circuit] at the
                                                passed [offset]
    with the given [size].

    Parameters:
    [size]: How many qubits the QFT should be able to deal with
    [circuit]: Quantum circuit to apply QFT gates to
    [offset]: The offset to apply the gates in the passed circuit to
    """

    # Create core QFT circuit
    for i in range(size - 1, -1, -1):
        # Apply Hadamard gate on the i'th qubit (most significant qubit)
        circuit.h(offset + i)

    # Apply controlled rotation around the Z-axis (CU1-Gate)
    for j in range(i):
        angle = np.pi / 2 ** (i - j)

        circuit.cu1(theta=angle, control_qubit=offset + j, target_qubit=
                                                offset + i)

    # Add ending swaps
    for i in range(size // 2):
        circuit.swap(offset + i, offset + size - 1 - i)


def inverse_qft(size, circuit, offset=0):
    """
    Add the inverse quantum Fourier transform gates for the passed [circuit]
    at the given [offset] with the specified [size].

    Parameters:
    [size]: How many qubits the inverse QFT should be able to deal with
    [circuit]: Quantum circuit to apply inverse QFT gates to
    [offset]: The offset to apply the gates in the passed circuit to
    """

    # Add starting swaps
    for i in range(size // 2):
        circuit.swap(offset + i, offset + size - 1 - i)

    # Create core QFT circuit
    for i in range(size):
        # Apply controlled rotation around the Z-axis (CU1-Gate)
        for j in range(i - 1, -1, -1):
            angle = - np.pi / 2 ** (i - j)
```

```
            circuit.cu1(theta=angle, control_qubit=offset + j, target_qubit=
                                                    offset + i)

    # Apply Hadamard gate on the i'th qubit (most significant qubit)
    circuit.h(offset + i)
```

B.2 | Circuit building code

```
def prepare_circuit(qubits_per_circuit, repetitions=1):
    """
    Prepare the quantum circuit needed for QFT with repetition code QEC.

    Parameters:
    [qubits_per_circuit]: Number of qubits per repetition
    [repetitions]: Number of repetitions to make for QEC

    Returns:
    - The prepared quantum circuit
    - A list of quantum registers
    - A list of ancilla quantum registers
    - A list of classical registers
    - A list of ancilla classical registers
    """

    # Add QFT registers for each repetition
    qrs = [] # Quantum registers
    crs = [] # Classical registers

    for r in range(repetitions):
        label = chr(ord('a') + r)

        qr = QuantumRegister(qubits_per_circuit, label)
        cr = ClassicalRegister(qubits_per_circuit, f'{label}_{{classical}}')

        qrs.append(qr)
        crs.append(cr)


    # Add ancilla registers for each repetition
    qars = [] # Quantum ancilla registers
    cars = [] # Classical ancilla registers

    ancilla_qubit_count_per_qubit = (repetitions - 1)

    for q in range(qubits_per_circuit):
        qr = QuantumRegister(ancilla_qubit_count_per_qubit, f'q{q}_{{ancilla
                                                }}')
```

```
        cr = ClassicalRegister(ancilla_qubit_count_per_qubit, f'q{q}_{{
                                                classical}}')

        qars.append(qr)
        cars.append(cr)

    return QuantumCircuit(*qrs, *crs, *qars, *cars), qrs, crs, qars, cars


def prepare_rc_qec_qft_circuit(num_to_encode, repetitions=1, with_barriers=
                                        True):
    """
    Prepare the full QEC QFT circuit.

    Parameters:
    [num_to_encode]: Decimal integer number to encode in the circuit
    [repetitions]: Number of repetitions for the repetition QEC
    """

    # Step 1: Encode the decimal integer number to a bit list
    enc_num_bits = to_bit_representation(NUM_TO_ENCODE)

    # Step 2: Prepare the quantum circuit including ancilla qubits
    qc, qrs, crs, qars, cars = prepare_circuit(len(enc_num_bits),
                                           repetitions=repetitions)

    # Step 3: Encode the bits in the quantum circuit
    for r in range(repetitions):
        encode_bits(enc_num_bits, qc, offset=r * len(enc_num_bits))

    if with_barriers:
        qc.barrier()

    # Step 4: Add the QFT gates to the circuit
    for r in range(repetitions):
        qft(len(enc_num_bits), qc, offset=r * len(enc_num_bits))

    if with_barriers:
        qc.barrier()

    # Step 5: Add the inverse QFT gates to the circuit
    for r in range(repetitions):
        inverse_qft(len(enc_num_bits), qc, offset=r * len(enc_num_bits))

    if with_barriers:
        qc.barrier()

    # Step 6: Add CNOTs to apply the QFT result qubits on our ancilla qubits
    for a in range(len(qars)): # For each ancilla qubit register
```

```
        qar = qars[a]

        for offset in range(2):
            for i in range(qar.size):
                ancilla_qubit = qar[i]
                qubit = qrs[i + offset][a]

                qc.cx(control_qubit=qubit, target_qubit=ancilla_qubit)

        if with_barriers:
            qc.barrier()

    # Step 7: Add measurements for the ancilla qubits
    for i in range(len(qars)):
        qar = qars[i]
        car = cars[i]

        qc.measure(qar, car)

    if with_barriers:
        qc.barrier()

    # Step 8: Add measurements for the qft results
    for i in range(len(qrs)):
        qr = qrs[i]
        cr = crs[i]

        qc.measure(qr, cr)

    return qc
```

B.3 | Decoding result from circuit

```
def decode_result(result_str, error_correction=True,
                                        exclude_error_results_during_merge=
                                        False):
    """
    ### Description ###
    Decode the passed result string ([result_str]) from the quantum
                                            computation in the following form:
    'ZZ YY XX CCC BBB AAA' where either letter of A, B, C, X, Y and Z can be
                                             either 0 or 1.

    ### String parts meanings ###
    AAA: Measurement result of the a-register
    BBB: Measurement result of the b-register
    CCC: Measurement result of the c-register
```

```
    XX: Measurement result of the ancilla bits for the first qubit of the a
                                            -, b- and c-registers
    YY: Measurement result of the ancilla bits for the second qubit of the a
                                            -, b- and c-registers
    ZZ: Measurement result of the ancilla bits for the third qubit of the a
                                            -, b-, and c-registers

    ### Ancilla bits meaning (ZZ, YY and XX) ###
    00: No bit-flip happened
    01: Bit-flip in register a
    10: Bit-flip in register c
    11: Bit-flip in register b

    ### Arguments ###
    - result_str: The result string to decode
    - error_correction: Whether to perform error correction
    - exclude_error_results_during_merge: Whether to exclude results where
                                            errors occurred on during merging

    ### Returns ###
    The decoded bit representation of the initial number fed into the
                                            quantum circuit
    with error correction using the ancilla bits and repetition already
                                            applied.

    ### Example ###
    Input: '10 01 00 010 110 010'

    ZZ = 10 -> means that the third qubit of register c has suffered from a
                                            bit-flip
    XX = 01 -> means that the second qubit of register a has suffered from a
                                             bit-flip
    YY = 00 -> means not bit flip happened in the first qubit

    Corrected CCC = 110 (previously 010)
    Corrected BBB = 110 (previously 110)
    Corrected AAA = 000 (previously 010)

    Result = 110 (Merged CCC, BBB and AAA)
    """

    # Step 1: Parse result string
    parts = result_str.split()

    if len(parts) != 6:
        raise Exception(f'Wrong result string format: Expected 6 strings
                                                separated by white spaces.
                                                Instead got "{result_str}"')
```

```
ZZ = parts[0]
XX = parts[1]
YY = parts[2]
CCC = parts[3]
BBB = parts[4]
AAA = parts[5]

if len(ZZ) != 2 or len(XX) != 2 or len(YY) != 2 or len(CCC) != 3 or len(
                                        BBB) != 3 or len(AAA) != 3:
    raise Exception(f'Wrong result string format: Expected string to
                                            have the form "ZZ XX YY CCC
                                            BBB AAA" where each letter
                                            represents either 0 or 1')

def to_bit_array(s):
    l = list(s)

    for i in range(len(l)):
        ch = l[i]

        if ch != '0' and ch != '1':
            raise Exception(f'Wrong result string format: Expected
                                                    string to contain only
                                                     0s and 1s (separated
                                                    by white spaces)')

        l[i] = int(ch)

    return l


ZZ = to_bit_array(ZZ)
XX = to_bit_array(XX)
YY = to_bit_array(YY)
CCC = to_bit_array(CCC)
BBB = to_bit_array(BBB)
AAA = to_bit_array(AAA)

# Step 2: Correct bit-flips indicated by the error syndrome (ancilla
                                        bits)
def correct_bit_flip(err_syndrome, idx, regs):
    reg_idx = None
    if err_syndrome[0] == 0:
        if err_syndrome[1] == 0: # No Bit-flip
            return None
        else: # Bit-flip in register a
            reg_idx = 2
    else:
        if err_syndrome[1] == 0: # Bit-flip in register c
```

```
                reg_idx = 0
            else: # Bit-flip in register b
                reg_idx = 1

        reg = regs[reg_idx]
        reg[idx] = 0 if reg[idx] == 1 else 1

        return idx

    regs = [CCC, BBB, AAA]
    corrected_reg_indices = []
    if error_correction:
        reg_idx = correct_bit_flip(ZZ, 0, regs)
        if reg_idx is not None:
            corrected_reg_indices.append(reg_idx)

        reg_idx = correct_bit_flip(YY, 1, regs)
        if reg_idx is not None:
            corrected_reg_indices.append(reg_idx)

        reg_idx = correct_bit_flip(XX, 2, regs)
        if reg_idx is not None:
            corrected_reg_indices.append(reg_idx)

    # Step 3: Merge results
    result = [0] * 3

    for i in range(len(result)):
        zeros = 0
        ones = 0
        for a in range(len(regs)):
            reg = regs[a]
            if not exclude_error_results_during_merge or not a in
                                                    corrected_reg_indices:
                if reg[i] == 0:
                    zeros += 1
                else:
                    ones += 1

        result[i] = 1 if ones > zeros else 0

    return result
```

B.4 | Source code used to generate a histogram of the results

```
def plot_histograms(counts):
    fig, axes = plt.subplots(2, 2, figsize=(14,8))
```

```
# Plot without performing any error correction
result_map = {}
for key, value in counts.items():
    parts = key.split()

    for i in range(3, 6, 1):
        r = parts[i]

    if r not in result_map:
        result_map[r] = value
    else:
        result_map[r] += value

plot_histogram(result_map, ax=axes[0][0])
axes[0][0].set_title("No error correction", fontsize=20)

# Plot without performing bit-flip correction but allow merging
result_map = {}
for key, value in counts.items():
    r = "".join(map(lambda x: str(x), decode_result(key,
                                                    error_correction=False)))

    if r not in result_map:
        result_map[r] = value
    else:
        result_map[r] += value

plot_histogram(result_map, ax=axes[0][1])
axes[0][1].set_title("Only merging results", fontsize=20)

# Plot with error-correction
result_map = {}
for key, value in counts.items():
    r = "".join(map(lambda x: str(x), decode_result(key)))

    if r not in result_map:
        result_map[r] = value
    else:
        result_map[r] += value

plot_histogram(result_map, ax=axes[1][0])
axes[1][0].set_title("Error correction", fontsize=20)

# Plot without allowing bit-flip results to be merged
result_map = {}
for key, value in counts.items():
    r = "".join(map(lambda x: str(x), decode_result(key,
                                                    exclude_error_results_during_merge
```

```
                                                =True)))

        if r not in result_map:
            result_map[r] = value
        else:
            result_map[r] += value

    plot_histogram(result_map, ax=axes[1][1])
    axes[1][1].set_title("Only merge results without error", fontsize=20)

    fig.tight_layout(pad=2.5)

    fig.savefig('out/test-histogram-qec-results.pdf', transparent=True)
```