Delft, University of Technology

Quantum Information Project

On Quantum Process Tomography

Group 4

Authors:
B. Apak - 4552342
J.C. van der Zalm - 4440943
G. Sotiropoulos - 5231175

Supervisor:
Prof. Dr. ir. Leo DiCarlo

22nd January 2021

Abstract

Quantum Process Tomography (QPT) was executed using the Pauli transfer matrix, where readout error correction was applied to minimise the readout error that is present in state tomography measurements. To get a good sense of the error in the fidelity, with which we quantify how good the realised quantum process matches the theoretical process, the bootstrapping technique was applied. After having done QPT on two qubits we went on to write a more general algorithm which could handle any dimensionality you want.

After the groundwork had been done, the results of these efforts were presented. We found that when we compared the 3 backends IBM Yorktown, IBM Ourense and Starmon-5 the fidelities were usually abiding the order: $F_{ourense} > F_{yorktown} > F_{Starmon-5}$. Besides unitary processes the Pauli transfer matrix could also describe non-unitary processes. We explored both the symmetrical mixing in a depolarising channel as well as the asymmetrical mixing in an amplitude damping channel. From this we conclude that the asymmetrical behaviour in a process seems to be mostly captured in the first column of the transfer matrix.

Contents

1 Introduction **3**

1.1 Proposal 3

1.2 Statements after the report was written 4

1.3 Introduction to process tomography 4

2 Theory **4**

2.1 State tomography 4

2.2 Process tomography 4

2.3 Pauli state vectors and Pauli sets 5

2.4 The Pauli Transfer Matrix 5

2.4.1 Definition and properties 5

2.4.2 Experimental Calculation 6

2.4.3 Measuring efficiency - Average Gate fidelity 7

2.5 Extracting statistical error - Bootstrapping 8

2.6 Read-out error correction 9

3 The code **10**

3.1 General Implementation 11

3.2 Modelling Non-Unitary Processes 12

3.3 Modelling higher dimensional Processes 13

4 Results - Discussion **15**

4.1 Assumptions for the experimental implementation 15

4.2 Two qubit Process Tomography 16

4.2.1 CNOT gate on different backends 17

4.2.2 CZ gate on different backends 18

4.2.3 $H^{\otimes 2}$ on different backends 19

4.3 Statistical error - Number of shots 21

4.4 Non-Unitary Single Qubit Processes 22

4.4.1 Symmetrical Mixing - The depolarising channel 22

4.4.2 Asymmetrical Processes - Amplitude damping channel 24

5 Conclusion **27**

1 Introduction

This report is part of the course AP3421-PR Fundamentals of quantum information project. Before we started this course we wrote a project proposal, for completeness it is given in the following section.

1.1 Proposal

Title: Investigating Quantum process tomography on multiple backends

Description: Our goal is to implement quantum process tomography for 2 qubits. We would like to start from simple quantum state tomography for 1 and 2 qubits. Then we shall proceed to deal with quantum process tomography for 1 qubit and figure out what the results are for simple unitary rotations and generalise into single qubit processes. After fully understanding the single qubit case, we aim to tackle the 2 qubit case with the final aim of comprehending how the 240 different parameters describing such a process can be interpreted. Slowly building up in complexity so that we can keep a good understanding of what is happening.

The end goal is that we delve into what actually is a quantum process and how it can be described by the chi-matrix mentioned in Nielsen & Chuang. We want to compare the differences between the Spin-2 and Starmon-5 qubits, as well as from simulations. We will try to quantify our results by measuring the time the algorithm takes on all platforms, but also the accuracy of the algorithm on different platforms. This will be done by looking at how well the algorithm could predict a certain predetermined set of operations, for example by measuring the variance. Later on we could look with more detail into how many times we repeat a process to apply tomography successfully. More towards the end of the project we could aim to optimize the tomography algorithm and see how many runs are necessary and what accuracy we could obtain.

Platform: We will work with the quantum inspire platforms from QuTech, where at the beginning we will mainly focus on writing code for the QX quantum simulator. After we are content with the results on that platform we also want to use the Spin-2 hardware backend. For us it would be nice to compare the results we get from the actual on chip calculations and see if we can explain any differences. We would also like to work with the Starmon-5 hardware: we expect the systems to have restrictions and it would be a nice opportunity for us to work with the hardware backends, learn what the restrictions are and see how it impacts the measurements.

Motivations:

Boran: I think this is the perfect opportunity to not only apply what we learned in the fundamentals of quantum information course, but also gain a deeper theoretical understanding of what we've learned, because by really applying something you always find out that things turn out to be slightly different than you thought they were.

Yorgos: I am really excited at the possibility of learning more about this fascinating field through this project and eventually really grasp the elusive concepts of quantum processes and their correlations to density matrices.

Joost: This would be a good opportunity for me to get more experience with the experimental part of QI. Which is also nice to prepare me for my MSc thesis later on. Looking forward to the moment we can press run and the code finally works and gives the results we were aiming for!

1.2 Statements after the report was written

In our proposal we wanted to use the chi-matrix from Nielsen & Chuang [1]. After further research we wanted to apply the Pauli Transfer matrix [2]. Another aspect that we have strayed away from was using the Spin-2 backend. In the time that we were gathering our results, the Spin-2 system was not available so we ended up not being able to reliably use it to compare our different processes with other backends.

1.3 Introduction to process tomography

To carry out process tomography entails that you find the process that a quantum system has undergone. This is useful because after carrying out process tomography you can essentially estimate the process that your system (previously comparable to a black box) has undergone. This can be very useful in practice if you, for example, want to know if a quantum computer is calibrated correctly.

Furthermore, it can help in calibrating the components in your quantum system by executing process tomography every time a change in the quantum system has been made to evaluate the effect of this change in the quantum system. To carry out process tomography, first we will discuss some theory related to process tomography, after which we will explain how our code, that can be seen in the Appendix, has implemented process tomography. After explaining this theory and the experimental method, the results of this paper will be presented. Finally a conclusion will be given, briefly summarising the contents of this paper and containing some final comments.

2 Theory

2.1 State tomography

A quantum state that is a linear combination of basis states collapses to 1 basis state once it is measured, and gives the corresponding outcome of the measurement for that basis state. This means that the state can (in general) not be characterised by only one measurement. We can however, due to the probabilistic properties of a quantum state, estimate what a quantum state is after measuring that same state many times and finding the mean value of a measurement in for example the X direction. After doing this for every direction X, Y and Z we can estimate the real quantum state of the system. This concept is called quantum state tomography. A more extensive description of this concept could be found in for example *Quantum Computation and Quantum Information* by Nielsen & Chuang [1].

2.2 Process tomography

Quantum process tomography is very closely related to quantum state tomography. At the start of process tomography, one wants to initialise the system in a certain state and let that system undergo a process. Then at the end of the process, we apply state tomography to find out what the final state of the system was after the process. However, if we initialise the system in the $|0\rangle$ state and our process consists of a single Z gate, we would not get an answer that is different from having no process at all! This is why for process tomography we need to apply state tomography for a multitude of different initial states.

2.3 Pauli state vectors and Pauli sets

Our notions of the Pauli state vectors and Pauli sets described in this section are based on the *Universal Quantum Gate Set Approaching Fault-Tolerant Thresholds with Superconducting Qubits* paper [2]. One way of characterising a quantum state is by specifying the expected value of a measurement in a certain direction.

In particular, for a 1 qubit system to be uniquely described, the expected values <X>, <Y> and <Z> need to be calculated. For example, the states of the computational basis correspond to the set: $\big($ <X>,<Y>,<Z> $\big) = \big(0, 0, \pm 1\big)$, with $+1$ corresponding to $|0\rangle$ and -1 corresponding to $|1\rangle$. These can be also identified as the Bloch coordinates of the state.

Unfortunately, the neat visualisation of the Bloch vector does not scale to more than 1 qubit. This is why we will be using expectation values/correlations to identify a 2 qubit state. We will be referring to these expectation values that characterise a state as its Pauli set and we will construct a vector comprising of these expectation values which we will call *Pauli state vector.* Dropping the angles <> for simplicity, for a 2 qubit state this corresponds to the 16-vector:

$$\vec{p} = (\text{II, XI, YI, ZI, IX, IY, IZ, XX, XY, XZ, YX, YY, YZ, ZX, ZY, ZZ}) \tag{1}$$

For example: the state $|11\rangle$ corresponds to the Pauli vector:

$$\vec{p} = (1, 0, 0, -1, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 1)$$

This notation has the advantage that all of our parameters correspond to measurable physical quantities and can be given by 16 real numbers restricted to $[-1, 1]$. Notation-wise this comes at the cost of briefness. Also the transformation back to the ket notation is not trivial.

Note that:

- As in the case of 1 qubit, the expectation values are also bounded by extra conditions. Recall the 1 qubit case: $|< X >|^2 + |< Y >|^2 + |< Z >|^2 \leq 1$
- The inclusion of the expectation value $< II >$ is trivial and equal to 1 for any state but this way we preserve a mapping from n qubits to a 2^{2n} vector.

2.4 The Pauli Transfer Matrix

Again, the notions we introduce here are based on another paper [2].

2.4.1 Definition and properties

Having this experimentally powerful notation described previously in our arsenal, we can formulate a way to describe any quantum process in an intuitive way.

We will define the *Pauli Transfer Matrix* $\mathcal{R}$ as the linear operator that transforms a Pauli State Vector into another one.

$$\vec{p}_{out} = \mathcal{R}\,\vec{p}_{in} \tag{2}$$

A quantum process thus can be described by specifying this operator $\mathcal{R}$ that transforms an input state $\vec{p}_{in}$ into an output state $\vec{p}_{out}$.

For a 2-qubit system this operator will be a 16x16 matrix while for a 1-qubit system it will be a 4x4 matrix. Due to $< I^{\otimes n} >= 1$ the first entry of the matrix will always be 1.

Recalling the restriction $[-1, 1]$ of the entries of a Pauli state vector, we can easily conclude that this matrix will also be real valued and each entry will also be restricted to $[-1, 1]$.

Note that the transfer matrix is not bounded by unitarity of operators acting on the system. It can capture any mixing of the initial state as this can be encoded as a decrease of the absolute value of some correlations/expectation values.

For instance, $\mathcal{R} \approx \frac{1}{3}\mathbb{1}$ captures symmetrical mixing of a state by a factor of 3, which is non-unitary quantum process. The $\approx$ is used since as we said, the first entry is always fixed for a transfer matrix.

2.4.2 Experimental Calculation

The definition of the Transfer matrix (2), can guide us in order to calculate it experimentally.

Ideally, we would input the states with the vectors:

$$\begin{aligned} \vec{p}_{in\,1} &= (1, 0, ..., 0, 0) \\ \vec{p}_{in\,2} &= (0, 1, ..., 0, 0) \\ &\quad ... \end{aligned}$$

and we would measure the transformed vectors:

$$\vec{p}_{out\,1} \qquad \vec{p}_{out\,2} \qquad ...$$

The transfer matrix would then be easily constructed by the column vectors $\vec{p}_{out\,i}$.

However, this is not possible for quantum states due to the restrictions to the expectation values. To overcome this, we will input all the cardinal states so as to make sure we capture the behaviour of the transfer matrix on all possible inputs. Recall that the cardinal states for a single qubit are the six states:

$$\left\{ |0\rangle, |1\rangle, |+\rangle, |-\rangle, |+i\rangle, |-i\rangle \right\} \tag{3}$$

For 2 qubits they will be the tensor product of all possible combinations, totalling 36 different cardinal state mixtures.

It is trivial to see that this set of vectors forms an overcomplete basis for the Hilbert space of the system. But this is needed since as we saw earlier we can only use real valued entries for the transfer matrix and with this overcomplete basis we can express any state vector with real values. Also, any potential asymmetricalities will be encapsulated by this method.

We notate the total input as a 16x36 matrix so that every column corresponds to a cardinal Pauli state vector.

$$\boldsymbol{M}_{in} = \begin{pmatrix} \vec{p}_{in\,1} & \vec{p}_{in\,2} & ... & \vec{p}_{in\,k} \end{pmatrix} \tag{4}$$

Similarly, the total output matrix of these inputs will be:

$$\boldsymbol{M}_{out} = \begin{pmatrix} \vec{p}_{out\,1} & \vec{p}_{out\,2} & \cdots & \vec{p}_{out\,k} \end{pmatrix} \tag{5}$$

Recalling the definition of the transfer matrix (2), we can conclude that:

$$\boldsymbol{M}_{out} = \mathcal{R}\,\boldsymbol{M}_{in} \tag{6}$$

Dimension-wise we can confirm that the above equation is valid since: $dim(\mathcal{R}) = 16 \times 16$

As $\boldsymbol{M}_{in}\,, \boldsymbol{M}_{out}$ are non-square matrices, to obtain the transfer matrix we will right-multiply with the pseudo-inverse of $\boldsymbol{M}_{in}$ which we shall denote with $\boldsymbol{M}_{in}^{+}$. Thus:

$$\mathcal{R} = \boldsymbol{M}_{out} \cdot \boldsymbol{M}_{in}^{+} \tag{7}$$

It is important to stress that the basis in which the transfer matrix is expressed will be the same as the one used for the Pauli state vectors.

Also, in order to facilitate our visual understanding of the transfer matrix, we opt for a pixel-map representation because the entries are bounded by ± 1 and most often it is a sparse matrix for simple processes. For reference, we present the trivial example of the identity matrix.

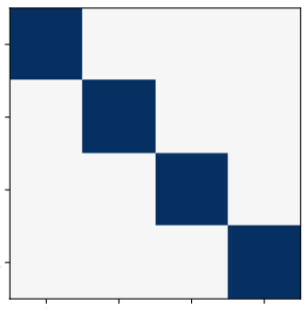


Figure 1: Identity Matrix expressed as pixel map with 4 measures

The coloring scale we will be using in this report is presented below:

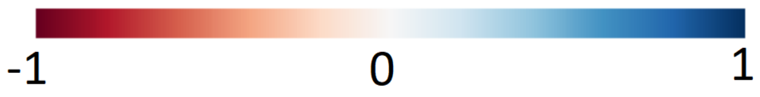


Figure 2: Scale of the pixel map

2.4.3 Measuring efficiency - Average Gate fidelity

Real quantum backends are known to be noisy. This applies also to the gates that are implemented by them. As a result, in our experimental attempt we will need a way to quantify the deviation of the experimentally calculated transfer matrix from the the theoretical one. We can construct such a measure. This we shall call the *average gate fidelity* and will be defined as:

$$F_{avg} = \frac{d\,F_{pr} + 1}{d+1} \tag{8}$$

where $d = 2^n$: is the dimensionality of the Hilbert space and F_{pr} the process fidelity [3]. For a unitary process, the latter can be calculated as:

$$F_{pr} = \frac{\mathrm{Tr}[\mathcal{R}_{ideal}^T \mathcal{R}_{exp}]}{d^2} \tag{9}$$

Note that the above trace represents the overlap between the theoretical and the experimental transfer matrix as:

$$\begin{aligned} \mathrm{Tr}[\mathcal{R}_{ideal}^T \mathcal{R}_{exp}] =& \mathrm{Tr}\Big[\sum_k (R_{ideal}^T)_{ik} (\mathcal{R}_{exp})_{kj}\Big] = \sum_i \sum_k (R_{ideal}^T)_{ik} (\mathcal{R}_{exp})_{ki} \\ =& \sum_{i,k} (R_{ideal})_{ik} (\mathcal{R}_{exp})_{ik} \end{aligned}$$

This measure that is theoretically bounded by $[0, 1]$ will allow us to compare the efficiency of different backends on the same gate or different gates of the same backend.

2.5 Extracting statistical error - Bootstrapping

As soon as we calculate experimentally the gate fidelity, we will need an error rate to accompany our result. One way would be for us to do the experimental calculation over and over again and extract the statistical error from these results. This is however a very inefficient way because for a single Process Tomography we need to realise a high number of circuits. For reference, to calculate a 2 qubit process transfer matrix we need to run $6^2 * 9 = 324$ circuits.

Instead, we shall use another highly efficient method called bootstrapping. We are going to generate many transfer matrices that could have been the result of our experiment and calculate the average gate fidelity for each of these matrices. The standard deviation of these fidelities will be our error rate.

In order to generate a new possible realisation of a transfer matrix, let's recall that a tranfer matrix is calculated by multiplying an input matrix and an output matrix which are both comprised of expectation values or correlations in general.

These correlations U have the following statistical properties:

$$\mu_U \in [-1, 1] \qquad\qquad s_U \in [0, 1] \tag{10}$$

where μ_U, s_U are the mean and standard deviation respectively.

The standard error of the mean when it is extracted by n number of shots is thus:

$$\sigma_U = \frac{s_U}{\sqrt{n}} \leq \frac{1}{\sqrt{n}} \tag{11}$$

Now we are going to generate a new possible expectation value of the correlation U. We will choose it randomly, assuming it obeys a normal distribution with the measured expectation value as a mean and a standard deviation of the upper bound of σ_U.

$$< U_{new} > \sim N(< U >, \frac{1}{\sqrt{n}}) \tag{12}$$

The procedure now is straightforward. We are going to resample all of the measured expectation values, extract a transfer matrix and then calculate its fidelity. Doing this for a sufficient amount[1] of times m we are going to collect m possible values for the average gate fidelity. The standard error of the fidelity will be the standard deviation of this population.

$$\Delta F = s_{F\,new} \tag{13}$$

2.6 Read-out error correction

The procedure explained in this chapter is based on the slides supplied to us by our supervisor Prof. dr. ir. Leo DiCarlo [4]. When working on real quantum hardware, we need to also take into consideration another issue: with some probability the result of a measurement may correspond to a different state than the actual state prior to measurement. This is the so called *read-out error*.

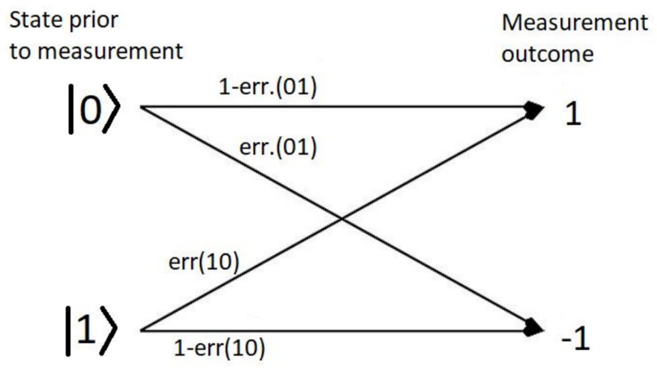


Figure 3: Diagram of the read-out error based on another figure [4, p. 8].

We, however, have as a goal to capture the gate errors with the concept of Average Gate Fidelity described in the previous section. As a result, we will need to eliminate the read-out error from our measurements by applying some correction to our raw data.

We will sketch here how this calibration for the read-out error works for 2 qubits, from which it is easily adjusted to a single qubit or it could be extended to more than 2 qubits.

Our goal is to create some linear correction to our raw data which is comprised of expectation values or correlations. For a 2 qubit system these correlations may be the expectation value of either qubit regardless of the other (e.g. $< IZ >, < ZI >$) and the correlation between their results ($< ZZ >$). In general, such a relation can be described by:

[1]As iterations m grow, the standard deviation becomes more and more precise. In our implementation we selected 1000 iterations and rounded it up.

$$\begin{pmatrix} < IZ > \\ < ZI > \\ < ZZ > \end{pmatrix}_{corrected} = \underbrace{\begin{pmatrix} b_{01} \\ b_{02} \\ b_{03} \end{pmatrix}}_{\vec{b}} + \underbrace{\begin{pmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{pmatrix}}_{B} \begin{pmatrix} < IZ > \\ < ZI > \\ < ZZ > \end{pmatrix}_{raw} \tag{14}$$

We will be assuming that the initialisation and single qubit unitary rotations are perfect, or equivalently that their error rate is significantly lower than the read out error which in general for quantum hardware is true [1]. Consequently, to capture the readout error we will be initialising our system in the 4 states:

$$|00\rangle \quad |01\rangle \quad |10\rangle \quad |11\rangle$$

For each of those states, we will just measure both qubits right after initialisation and extract the expectation values of these 3 correlations. These will correspond to the raw vectors. The corrected vectors will correspond to the theoretical vectors which can be trivially calculated. We present an example:

$$|10\rangle \quad \longrightarrow \quad \begin{pmatrix} < IZ > \\ < ZI > \\ < ZZ > \end{pmatrix}_{corrected} = \begin{pmatrix} 1 \\ -1 \\ -1 \end{pmatrix}$$

This results in a linear system of $4 * 3 = 12$ equations with 12 unknown coefficients: 3 for the offset vector $\vec{b}$ and 9 for the entries of the B matrix. As a result, the calculation of $\vec{b}$ and B is possible[2]. We will not elaborate on this calculation as it should be straightforward as long as we keep track of the theoretical values of the correlations.

At this point one may wonder: what about expectation values in other bases? First of all we should note that real quantum backends can only measure in the Z basis. To measure in other bases a suitable rotation is applied first. Recalling the assumption that rotation errors are insignificant, we can conclude that we can correct the readout error for measurements in any basis with the elements that we have calculated up until now.

For example, measuring in X and Y, we may use B and $\vec{b}$ to calibrate the read-out:

$$\begin{pmatrix} < IX > \\ < YI > \\ < YX > \end{pmatrix}_{corrected} = \vec{b} + B \begin{pmatrix} < IX > \\ < YI > \\ < YX > \end{pmatrix}_{raw} \tag{15}$$

3 The code

The GitHub repository containing our code can be found at: `https://github.com/joostvdzalm/Internation-QI-team`.

[2]For a single qubit it turns out that it is a system of 2 equations and for 3 qubits a system of 58 equations. They seem to be following the rule: $2^n(2^n - 1)$ required coefficients where n : number of qubits.

3.1 General Implementation

We have come to understand that the quest to describe a quantum process can be reduced to the experimental calculation of the transfer matrix. We wrote the code to build the transfer matrix for a 2 qubit system in python using the Qiskit SDK [5, 6]. The full code, together with some functions, can be seen in the Appendix, but in this section we will highlight some parts of the code since the code is a big part of our project.

After importing the necessary libraries and functions, we define what backend we are using and the name we want to give to this specific run of the code and we define the amount of shots. The first notable definition is the statevector backend sv_backend. To calculate the theoretical transfer matrix we want to apply some computations in Qiskit for which we use the built in statevector simulator.

```
sv_backend = Aer.get_backend('statevector_simulator') #statevector for theoretical
    transfer matrix
```

The second notable definition is the definition of B:

```
B=get_readout_err(experimental_backend, 8192)
```

From the function get_readout_err we find the matrix B with which we later need to calibrate the readout error.

In the code we use 4 nested for-loops to loop over all the different states for qubit 1 and qubit 2, and all different measurement directions for qubit 1 and qubit 2. We use the counter m for the (for 2 qubits) 6x6 $= 36$ different cardinal state combinations. And we use the counter n for the 9 different combinations of measurement directions $(XZ, YX, ZY...)$. This way we can also print those values to keep track of the progress when the experiment is running.

In the first part of the big for-loop, we set up what we need for the statevector simulation and we already set the II value to 1, since that will always hold true and doesn't require us to specify a circuit.

Next we build a circuit, for which we use Qiskit. We initialise the qubits in the states defined by the current iteration of the for-loop, this way we can ensure all cardinal states are cycled through. And lastly we define the process by the set of applied gates, but this is of course dependent on the process you want to examine. The last part of the circuit is to add a rotation to measure in the X or Y direction, but this is added later on in the code.

After we have defined the circuit we want to examine, we can use the statevector simulator to give us the expected values in a perfect world without any noise. With these we can later calculate the theoretical transfer matrix.

After doing the statevector simulation, we move on to the real deal: executing the job on a real quantum computer. Instead of measuring in the X direction, we add a -$\frac{\pi}{2}$ rotation around the Y axis and we measure in the Z direction like normal. We add a $\frac{\pi}{2}$ rotation around the X axis for every measurement of Y. This is in practice what happens either way, but the Quantum Inspire environment prefers this input (their platform can be found online [7]).

After executing the job we obtain a histogram with the amount of times the backend has found which measurement value. With the following code we then calculate the expected value:

```
176             expected_value=np.zeros(3)
177             for state, counts in histogram.items() :
178                 expected_value[0] += (-1)**(int(state[1]))*int(counts)     #
                        corresponds to IZ
179                 expected_value[1] += (-1)**(int(state[0]))*int(counts)     #
                        corresponds to ZI
180                 expected_value[2] += (-1)**(int(state[0])+int(state[1]))*int(counts)
                           #corresponds to ZZ
181
182             expected_value = expected_value / number_of_shots
```

This might be a bit mysterious at first. What is happening here is that the histogram has an element state (so for example 01) and the first element of expected value gets incremented by the amount of times the state of the first qubit is found to be 0, and gets decreased by the amount of times the state is found to be 1. The same holds for the 2nd qubit in line 179. And in line 180 it gets incremented by the counts if the states of qubit 1 and qubit 2 are the same, and gets decreased by the amount of counts the states of qubit 1 and qubit 2 are different. If we do this for all elements and then divide by the total number of shots we can calculate the expected value of $< IZ >$, $< ZI >$ and $< ZZ >$ respectively. When measuring in different directions we can then find the expectation values need to find the output Pauli set for every input cardinal state.

After finding the expected values we want to calibrate those expected values with the matrix B we have found earlier. We do this using the calibrate_readout_err function.

We save the data every time the for loop has run through one experiment to make sure that we can retrieve all data after a malfunction in the hardware backend. This way we can start the program again from the last working iteration instead of having to rerun the whole time consuming process again.

After the for loop all we need to do is calculate the transfer matrices (one theoretical, one experimental and we also calculate one experimental transfer matrix where we did not make up for the readout error).

Lastly we calculate the gate fidelity using the calculate_gate_fidelity function along with its error rate using the get_bootstrap_standard_error, and we plot the transfer matrices to give some insight.

3.2 Modelling Non-Unitary Processes

We have stated earlier that the Pauli Transfer matrix is able to capture the action of non-unitary processes. We have seen that in fact no actual gate can be unitary since the experimental implementation always introduces some finite error. In this section we shall implement some by definition non unitary processes in order to get a better feeling about the the Pauli Transfer representation. For simplicity, we will stick with single qubit processes and simulations since we are only interested about obtaining intuition and recognising non unitary elements in a transfer matrix.

Modelling non unitary processes can be done in various ways. One way is to run a simulation of doing no process at all i.e. the Identity operator but with a specified noise model added to the simulator. Another way to implement this is by having a circuit that is non-deterministic. What we mean by that is that some gate may or may not be applied on a single shot of the experiment. We will opt for the second method.

A hindrance we encountered was that many of these processes we wished to implement were not available for controlled action by some variable in Qiskit. For example, this variable could be the result of a measurement of a $|+\rangle$ state on the computational basis which gives a probabilistic bit. As a result, we apply the model on an even more elementary level.

Instead of doing single experiments with the number of shots as an argument, we will be doing single-shot experiments which will be done number-of-shots times. This way we can implement in a set frequency but randomly any procedure we wish from the ones that are available on Qiskit. This can be achieved using a random generation function. Let us sketch below how we implemented the scheme:

```
import random as my_random

expected_value = 0
for i in range(number_of_shots):
    #initialisation ...

    #random application of a process
    rand_number=my_random.randint(0,2)
    if (rand_number==0 or rand_number==1):
        #apply operation ...

    #rotations to measure ...
    qiskit.execute(circuit, backend=backend, shots=1)
    #get contribution to expected_value ...
expected_value = expected_value/number_of_shots
```

The above example applies some operation with a frequency of $\frac{2}{3}$.

3.3 Modelling higher dimensional Processes

After doing two dimensional quantum process tomography, we got the idea to extend our code so that it will also works for higher dimensional processes. To do this in a structured way the code was rewritten a bit. First off the code was put in a Quantum Process Tomography class:

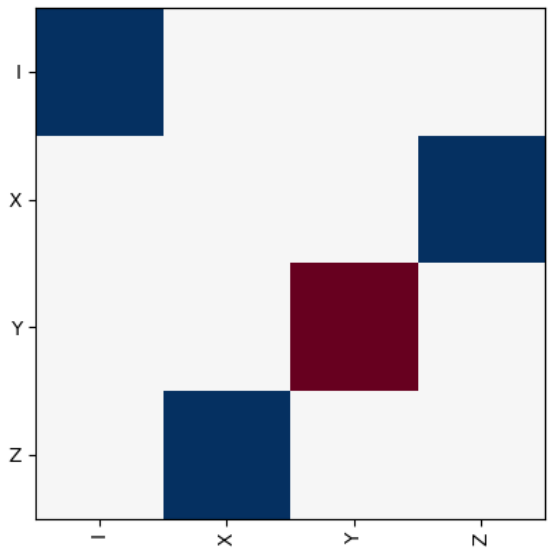
```
class QuantumProcessTomography:
```

Most of the sub routines look quite similar to what was done in the two dimensional case. But to generalize we needed to make a function which could generate a list of all possible inputs, assuming that we only input cardinal states. This is done in compute_all_qubit_inputs(self):

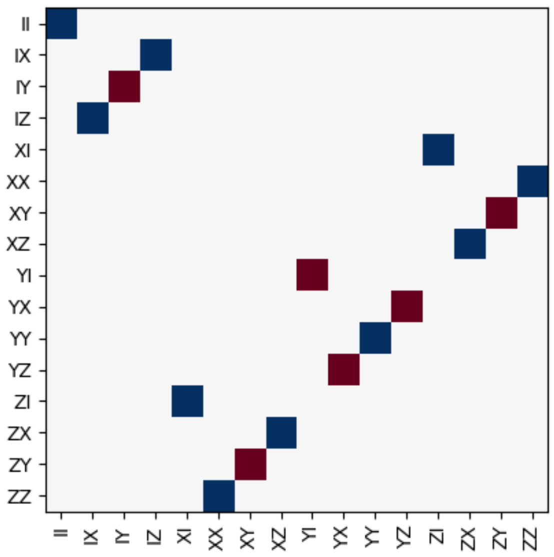
```
    def compute_all_qubit_inputs(self):
```

This function computes an array containing all possible qubit inputs. Given that we have n qubits and that we want all possible combinations of cardinal states. The second for loop works as follows, for every index it tries to find a cardinal state at a position which it can use, if it cannot find any a '0 state' will be used (notice that the start_input is first initialized with these states in the first for loop. By looping through all indices all combinations of cardinal states are made. This is done similarly to a number system (binary, decimal, hexadecimal). Here instead of numbers we have states which replace the numbers in the metaphor. So the states have been assigned an artificial value just as with the number systems. For example, the letter 'F' in hexadecimal has the decimal value 15. Then depending on the position of the 'F' a certain numeric value is represented. In the code the values 0, 1, 2, 3, 4 and 5 are assigned to the states $|0\rangle$, $|1\rangle$, $|+\rangle$, $|-\rangle$, $|+i\rangle$ and $|-i\rangle$ respectively. These values are raised to the power of the position and subtracted from the current loop index, subtracting the biggest possible values first. This ensures that at the end of the loop the leftover index should be zero, so that we successfully formed a new possible input it just being slightly different from the previous one. Once we have this list of inputs it isn't too difficult to extend the rest of the code to make it handle n-dimensional inputs.

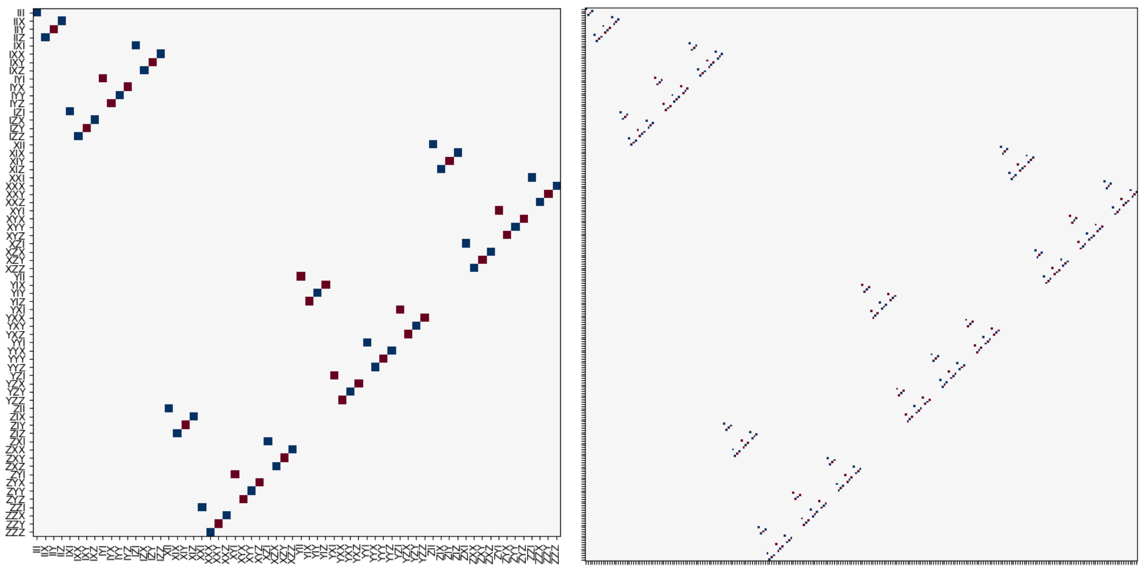
To give a proof of concept we performed a process that can be extended to any amount of qubits. We simply apply a Hadamard gate on every qubit. To showcase this, in figure 4 you can find the graphical illustrations of the Pauli transfer matrices up to four dimensions.



(a) One dimensional



(b) Two dimensional



(c) Three dimensional (d) Four dimensional

Figure 4: In these plots you can see a graphical depiction of the transfer matrices for n-dimensional Hadamard processes. The dark blue and red colours indicate the +1 and -1 values respectively. On the bottom and side you can find the directional labels for the input and output respectively.

It is interesting to see the fractal pattern appearing here. This is due to the fact that we apply the same gate on every qubit. There is a pattern to be found in how the directions repeat within the labels. Since there are four different labels (I, X, Y, Z) the pattern repeats every 4 rows/columns. The Hadamard gate rotates states in the Z-direction to the X-direction, this results in a blue square at the spot indicated by the (X, Z) label for the 1D case. But, going to higher dimensions the position of the Z label for that qubit will be shifted further to the right. This results in the 1D pattern to repeat going to higher dimensions. Similarly the larger 2D pattern is repeated four times in the 3D case, where these are located at spots according to the 1D pattern.

Note that the way this plot looks could drastically be changed by changing the ordering in the labels. However, the information the picture would contain of course stays the same. Another way of ordering the labels would be to start with the lower dimensional labels first. So for example for the 2D case the first 7 elements could contain the labels with an 'I'. In this way the top left would contain the lower dimensional correlations. In the results section we have chosen this approach as we also saw that this was done in literature [2, 3].

4 Results - Discussion

4.1 Assumptions for the experimental implementation

In our subsequent implementation we have made some silent assumptions that need to be addressed as they will be sources of systematical errors that cannot be contained by the fidelity error.

Read-out error correction time dependence assumption

In general, the read-out error of a quantum backend may vary in the duration of a single day. Our experiments on the other hand need at least 4 hours to complete and may take up to a single day depending on how busy a backend is. For each experiment we will be performing the read-out calibration only once in the beginning and we will be assuming that this stays constant throughout the experiment.

Experimental input correlations time dependence assumption

All of the experiments we run can be decomposed in 4 parts:

- Initialisation
- Unitary Transformation
- Analysis
- Measurement

As mentioned earlier, the transfer matrix should encapsulate how an input is transformed into an output. The input is not ideal and this is why we should also perform a "blank" process tomography for each process tomography we wish to study. The output of this blank process tomography will be the actual experimental input that we should use to eventually calculate the transfer matrix and then the fidelity.

However, this procedure takes a really long time and due to our limited resources we will also be assuming that this experimental input does not change for a particular backend. We allow ourselves to combine these experimental inputs with potential outputs that were taken days apart in order to estimate the transfer matrix.

4.2 Two qubit Process Tomography

Below we will present experimental results for the application of three different gates, each implemented on 3 different backends.

Gates:

- Controlled-Z gate also known as CZ gate
- Controlled-X gate also known as CNOT gate
- Hadamard on each of the 2 qubits ($H^{\otimes 2}$)

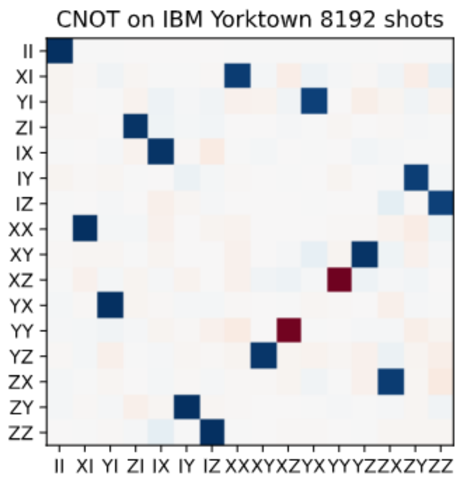
Experimental backends and number of shots:

- IBM Yorktown [8], 8192 shots
- IBM Ourense [9], 8192 shots
- Quantum Inspire Starmon-5 [10], 8192 shots

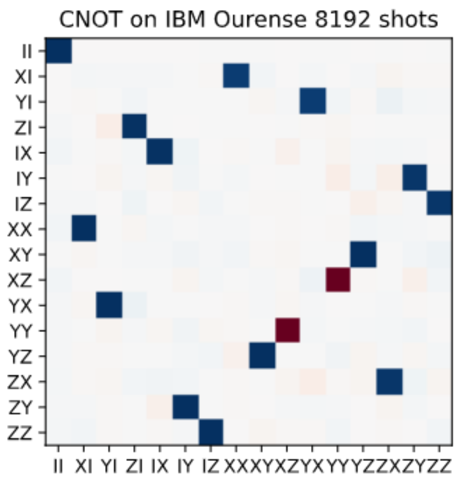
Note that we will be using qubits `q0` and `q1` for Yorktown and Ourense while for Starmon-5 we use a modified version of our code to access qubits `q1` and `q2`. This is because on Starmon-5 `q0` and `q1` are not directly connected.

4.2.1 CNOT gate on different backends

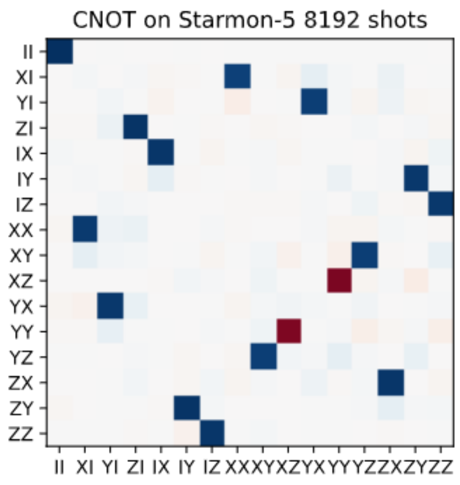
Below you can see the experimental results for the CNOT gate:



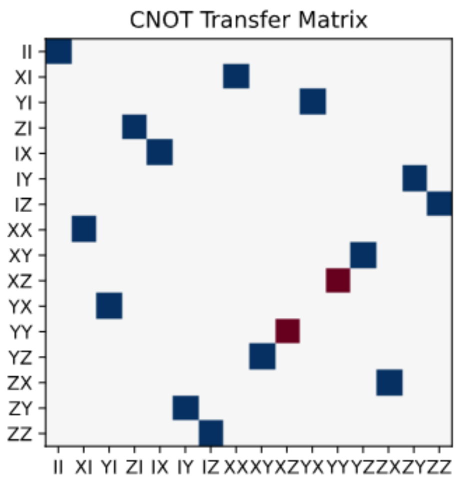
(a) $F = (98.16 \pm 0.13)\%$



(b) $F = (99.57 \pm 0.13)\%$



(c) $F = (96.77 \pm 0.13)\%$



(d) Theoretical CNOT

Figure 5: Transfer matrices for CNOT

Below we also present the calibration data from these backends

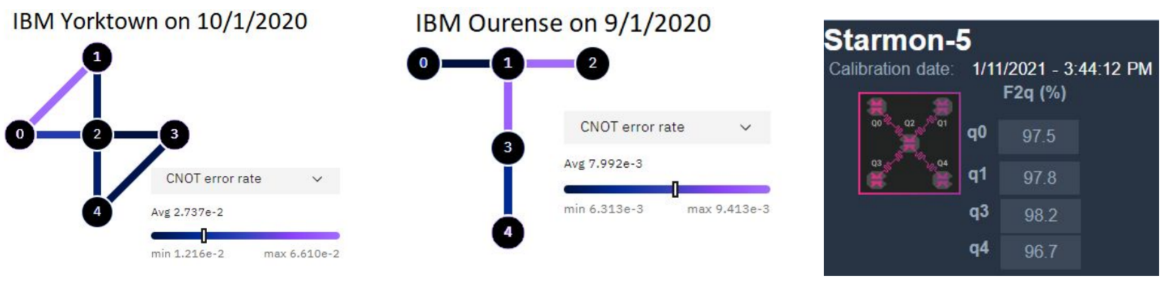


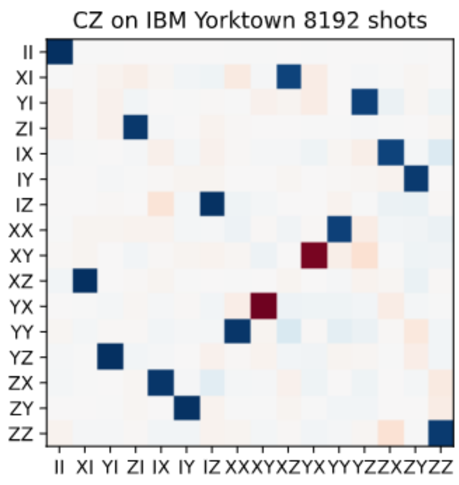
Figure 6: Calibration data from the quantum hardware used. [8, 9, 10]

At this point we should note that these calibration data fluctuate from day to day but on average they stay about the same. Also note that the data of Starmon-5 correspond to a CZ gate [10] which is the native 2-qubit gate of Starmon-5 but it is presented here for completeness.

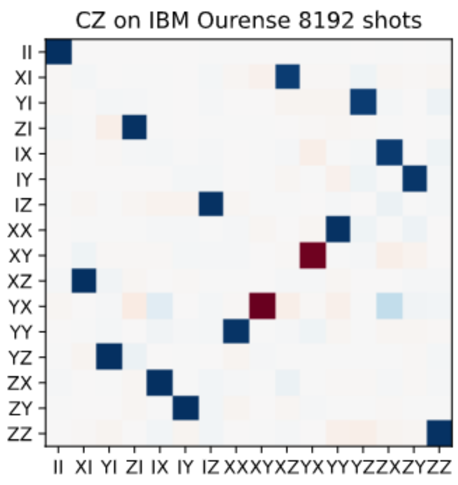
We observe that that the results we obtained are reasonable in terms of the documented values. Although, we should note that the greatest inconsistency seems to regard the error rate of IBM Yorktown backend. Upon closer inspection we noticed that the calibration data that are presented here seem to be on a really "bad" day of this particular backend.

4.2.2 CZ gate on different backends

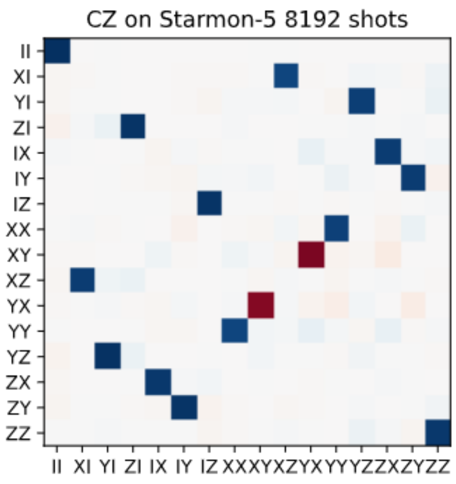
The experimental results for the CZ gate are as follows:



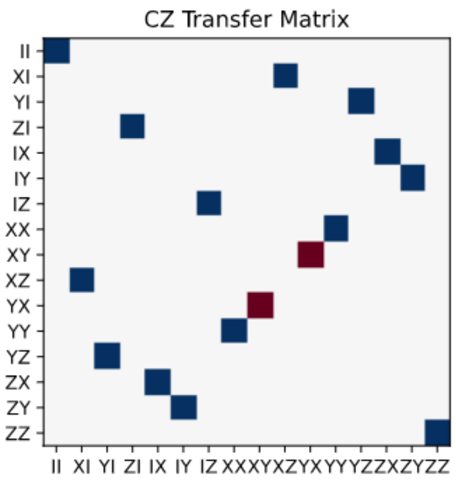
(a) $F = (97.17 \pm 0.13)\%$



(b) $F = (98.86 \pm 0.13)\%$



(c) $F = (96.38 \pm 0.13)\%$



(d) Theoretical CZ

Figure 7: Transfer matrices for CZ

We again notice the same tendency. Namely: $F_{ourense} > F_{yorktown} > F_{Starmon-5}$. This is because a CZ is decomposed by IBM's backends as CNOT and two single qubit rotations.

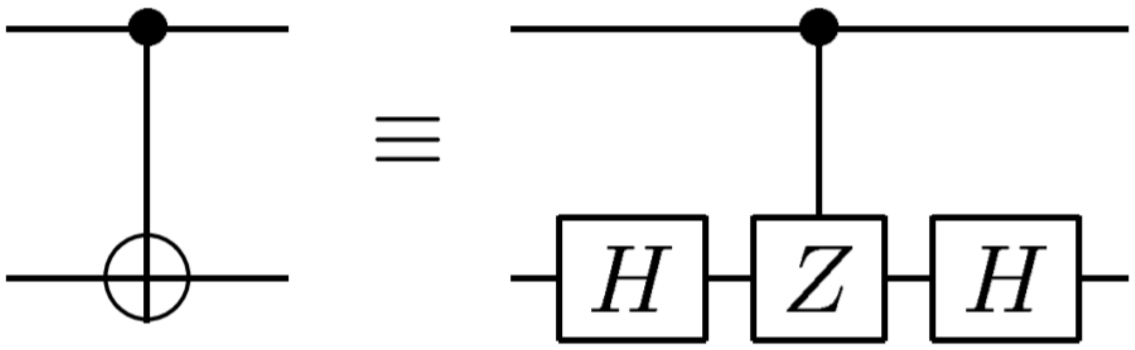


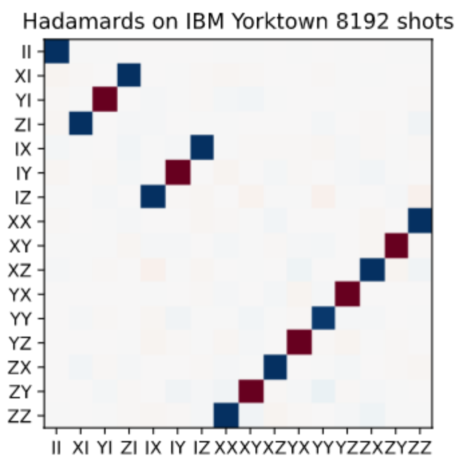
Figure 8: CNOT-CZ equivalence with single qubit gates [11, p. 5].

As we will find out later these rotations have really high fidelity errors and this is why we obtain lower fidelities than the CNOT but not too low.

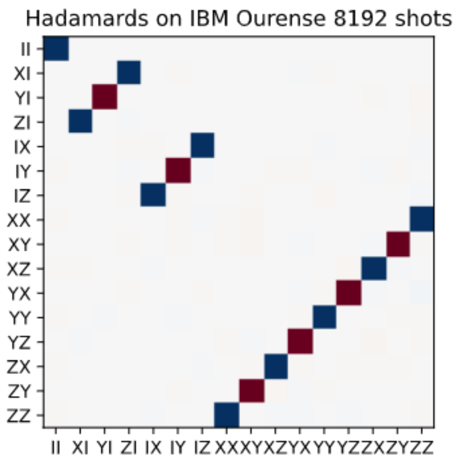
The alarming part is that Starmon-5 should have actually performed better than previously, since CZ does not have to be decomposed, while CNOT had to. We may attribute this to our experimental input data having been a out of date in comparison with the CNOT case or Starmon-5 having a "bad" day. We should also mention that Starmon-5 required the most time to complete an experiment, thus rendering the read-out calibration more inaccurate.

4.2.3 $\mathbf{H}^{\otimes 2}$ on different backends

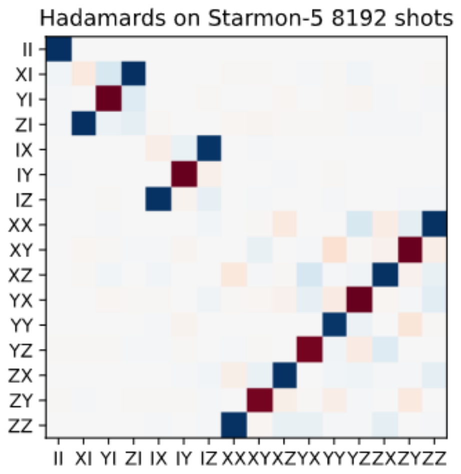
Below we present the experimental results of applying a Hadamard gate on each qubit:



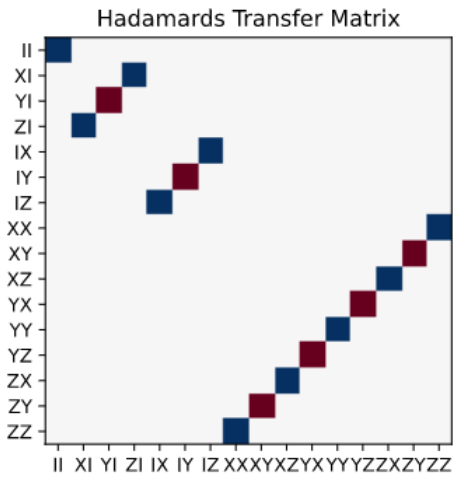
(a) $F = (100.01 \pm 0.13)\%$



(b) $F = (99.94 \pm 0.13)\%$



(c) $F = (98.96 \pm 0.13)\%$



(d) Theoretical $\mathrm{H}^{\otimes 2}$

Figure 9: Transfer matrices for $\mathrm{H}^{\otimes 2}$

We can see right off the bat that we have an issue with IBM backends. Not only our fidelity error allows for fidelities higher than 1 but also the calculated fidelity for Yorktown exceeds the unit value.

Upon closer inspection, this comes as no surprise, though. These backends have a documented single-qubit-gate error rate lower than 5×10^{-4} [8, 9] while the fidelity error rate is $\approx 10^{-3}$. As a result, we cannot distinguish the gate error rate!

The results of Starmon-5 are again alarming. While the documented single-qubit gate fidelities are above 99.8 [10] we seem to be getting results that are significantly below that. This reinforces the previously stated idea that we had to get our experimental input closer to the date of the the obtained output or to recalibrate the read-out mid experiment.

4.3 Statistical error - Number of shots

In the previous paragraphs there are two issues raised regarding the statistical error of the fidelity. The first being that this error seemed to be the same for all combinations of 2-qubit gates. The second second issue was that the statistical error was not sufficiently low in some cases so as to distinguish the error rate of the hardware.

We notice that one thing was constant in all our experiments: the number of shots. And in fact if we recall paragraph 2.5, the number of shots is used to calculate the standard deviation of the normal distribution.

$$< U_{new} > \sim N(< U >, \frac{1}{\sqrt{n}})$$

The higher the number of shots is, the sharper the distribution will be, less fluctuations will be assumed and in the end the higher the precision will be.

Below we present a simulation of the calculated statistical error, varying the number of shots.

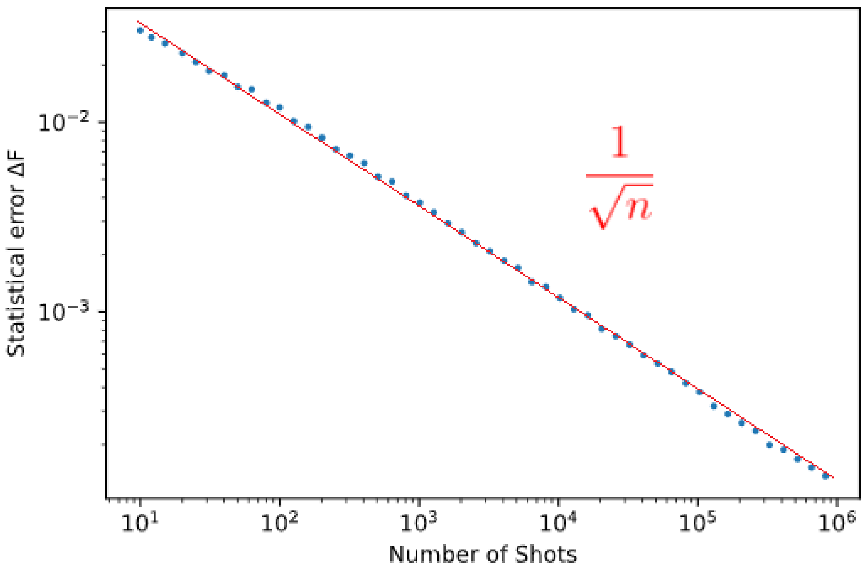


Figure 10: Fidelity statistical error as a function of the number of shots for 2-qubits.

We see clearly that:

$$\Delta F \propto \frac{1}{\sqrt{n}} \tag{16}$$

which is reasonable if we recall that the fidelity is calculated linearly from the transfer matrix which in turn is calculated linearly from our raw data.[Sec 2.4] Our error rate in the raw data was governed by $1/\sqrt{n}$, so it is expected that the error after propagating will still have this tendency.

Some points need to be addressed, however. There was a comment in our final presentation that it is quite peculiar how we may achieve a fidelity error of $\approx 1\%$ with only 100 shots. This comes as no surprise as 100 shots per circuit means $100 \times 6^2 \times 3^2 = 32400$ number of shots in total per experiment.[3] Also, we need to point out that we are using an overcomplete set of bases. Thus, we are performing more than necessary circuits that eventually contribute in this low statistical error.

We have to also stress that that Fig. 10 does not hold for a different number of qubits or for a different calculation of the transfer matrix (e.g. with less or more redundancy). In all of these cases, the propagation of the error will be different. We expect those to still obey the $1/\sqrt{n}$ but with a different offset.

4.4 Non-Unitary Single Qubit Processes

Below we present the results of some simple processes that we modelled according to the scheme we discussed in 3.2.

4.4.1 Symmetrical Mixing - The depolarising channel

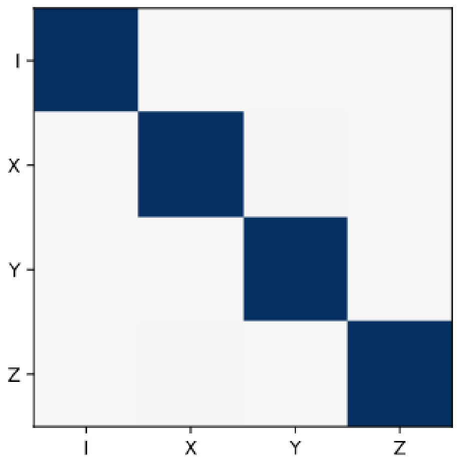
Let us recall the action of a depolarising channel [1, p. 378]:

$$\rho \longrightarrow \rho' = (1-p)\rho + p\frac{\mathbb{1}}{2} \tag{17}$$

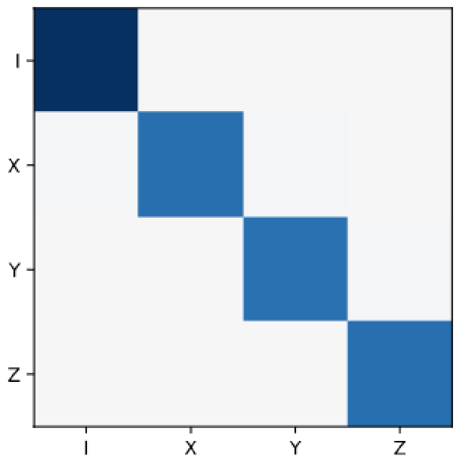
This is the formal way to introduce symmetrical mixing of the initial state, since there is no bias towards any direction. This can be considered a non-unitary process and can be modelled as such:

- with frequency $= 1-p$: we apply nothing to our circuit.
- with frequency $= p$: we lose our initial state and instead we get random state.

Below we can see the Transfer Matrices for $p = 0, 0.25, 0.5, 0.75, 1$.

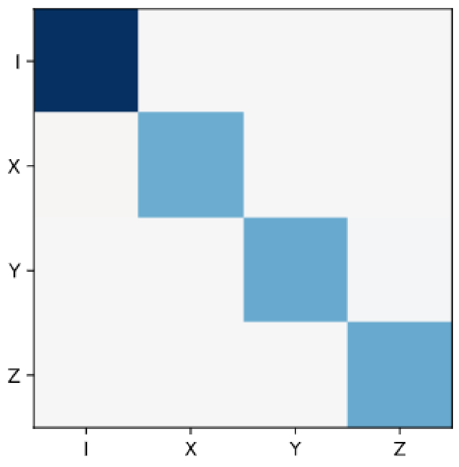


(a) Depolarising channel $p = 0$

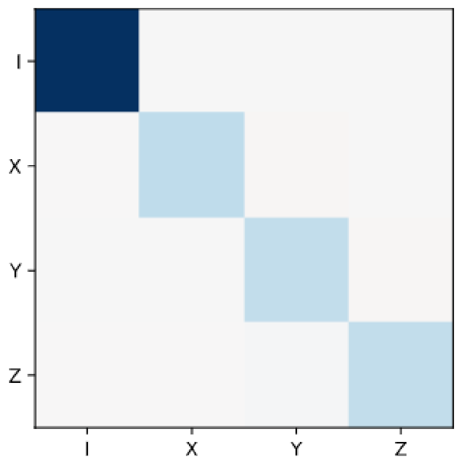


(b) Depolarising channel $p = 0.25$

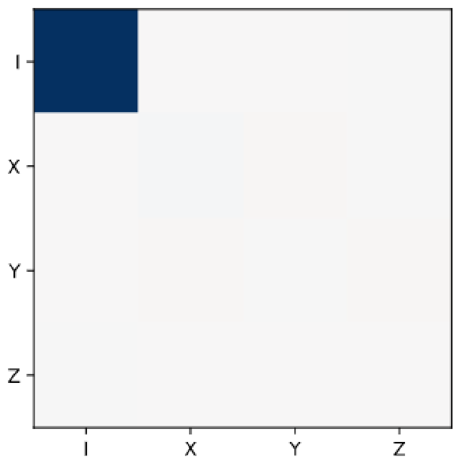
[3] 6^2 corresponds to the number of cardinal states and 3^2 corresponds to the lowest number of circuits per cardinal state we can have in order to obtain all the correlations.



(c) Depolarising channel $p = 0.50$



(d) Depolarising channel $p = 0.75$



(e) Depolarising channel $p = 1$

Figure 11: Depolarising channel Transfer Matrix for various p

We can see that as p grows, the elements of the diagonal shrink (except for the first element). This is expected since the higher the value of p, the more depolarised the channel is. Indeed we can see:

- $p = 0$ Identity-gate behaviour.
- $p = 1$ Complete loss of information. No matter what the input state is, the output is the maximally mixed state.
- $0 < p < 1$ Partial symmetrical mixing of the state.

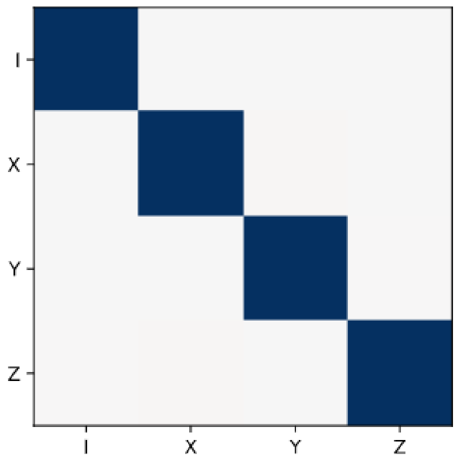
4.4.2 Asymmetrical Processes - Amplitude damping channel

Let's suppose that there is some environmental parameter that creates a bias towards the state $|0\rangle$. Depending on the time that we hold onto a qubit in such an environment, the qubit shall attain the state $|0\rangle$ with some probability, no matter what the initial state was. Thus, this can be considered a process that acts asymmetrically on the input state.

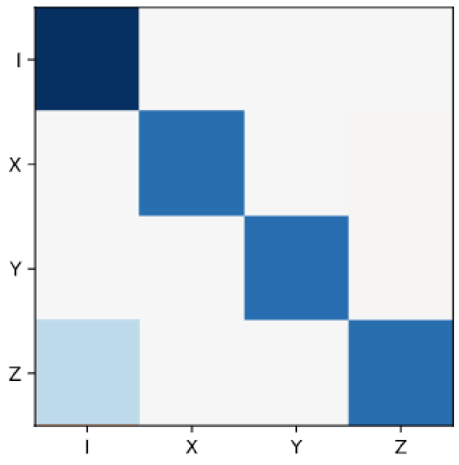
What was described above is the amplitude damping channel. [1, p. 380] In the photon presence/absence protocol, if we choose $|0\rangle$ to represent the absence of a photon, sending the photon through a fiber will result in the loss of the photon with some probability p and therefore any state will decay to state $|0\rangle$ with this probability. Equivalently, with probability $1 - p$ the initial state will not decay to $|0\rangle$.

This particular procedure is a non-unitary process and can be modelled as such:

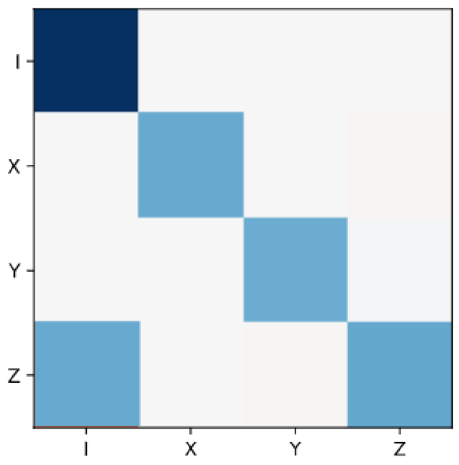
- with frequency $= 1 - p$: we apply nothing to our circuit.
- with frequency $= p$: we lose our initial state and instead we get the state $|0\rangle$.



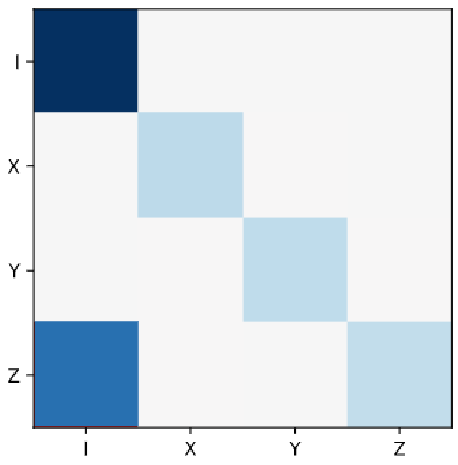
(a) Amplitude damping channel $p = 0$



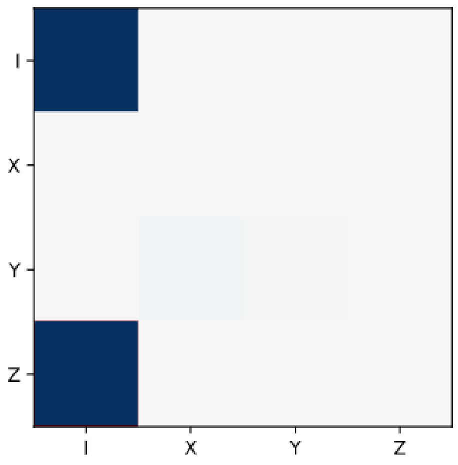
(b) Amplitude damping channel $p = 0.25$



(c) Amplitude damping channel $p = 0.50$



(d) Amplitude damping channel $p = 0.75$



(e) Amplitude damping channel $p = 1$

Figure 12: Amplitude damping channel Transfer Matrix for various p

Similarly to the previous paragraph, we see that the as p grows, the diagonal elements shrink but the element ZI gets closer to the value 1.

Note that the Pauli state vector:

$$\vec{p} = \frac{1}{2}(< I >, < X >, < Y >, < Z >) = \frac{1}{2}(1, 0, 0, 1)$$

corresponds to the state $|0\rangle$.

We can observe that:

- $p = 0$ Identity-gate behaviour.
- $p = 1$ Complete decay of initial state. Output state corresponds to the pure state $|0\rangle$.

- $0 < p < 1$ Partial decay of the initial state.

It is important to note that this is the first time that we see systematical contributions to the first column of the transfer matrix. And this came as a result of having an asymmetrical behaviour in our quantum process.

Thus, we can conclude that including the I row and column was really important for the definition of the Transfer Matrix. The first row seems to stay constant for any process we may devise. However, the first column seems to be capturing any asymmetricality introduced by our quantum process!

5 Conclusion

Summarising what has been described in this paper: quantum process tomography was executed using the Pauli transfer matrix, where readout error correction was applied to minimise the readout error that is present in state tomography measurements. To get a good sense of the error in the fidelity, with which we quantify how good the realised quantum process matches the theoretical process, the bootstrapping technique was applied. To find the Pauli transfer matrix corresponding to performing a quantum process on a certain backend, we wrote an algorithm. The code can be found in the Appendix and it was partly elucidated in Section 3. After having done QPT on two qubits we went on to write a more general algorithm which could handle any dimensionality you want. To show this we applied Hadamard gates up to 4 qubits.

After the groundwork had been done, the results of these efforts were presented. We found that when we compared the 3 backends IBM Yorktown, IBM Ourense and Starmon-5 the fidelities were usually abiding the order: $F_{ourense} > F_{yorktown} > F_{Starmon-5}$. Some assumptions were made to obtain these results. It was assumed that the readout-error will be constant over time, as would be the input state of the process that was used to estimate the transfer matrix. In some papers it was found that MLE methods can be used to better determine the Pauli transfer matrix [2] so some future research could be how to apply this MLE methods.

Besides unitary processes the Pauli transfer matrix could also describe non-unitary processes. We explored both the symmetrical mixing in a depolarising channel as well as the asymmetrical mixing in an amplitude damping channel. From this we conclude that the asymmetrical behaviour in a process seems to be mostly captured in the first column of the transfer matrix.

Furthermore, one could research whether doing readout-calibration multiple times during the whole quantum process tomography would improve the results. And of course it would be very interesting to compare more hardware backends with the code that is currently available, to see which system for example can realise a CNOT gate the best. Lastly, with more time and perhaps a closer connection to the backend, tomography experiments on hardware backends could be performed on more qubits.

References

[1] Michael A. Nielsen and Isaac L. Chuang. *Quantum computation and quantum information.* 10th Anniversary Edition. Cambridge University Press, 2010.

[2] Jerry M. Chow et al. "Universal Quantum Gate Set Approaching Fault-Tolerant Thresholds with Superconducting Qubits". In: *Physical Review Letters* 109.6 (Aug. 2012). ISSN: 1079-7114. DOI: `10.1103/physrevlett.109.060501`. URL: `http://dx.doi.org/10.1103/PhysRevLett.109.060501`.

[3] O.-P. Saira et al. "Entanglement Genesis by Ancilla-Based Parity Measurement in 2D Circuit QED". In: *Physical Review Letters* 112.7 (Feb. 2014). ISSN: 1079-7114. DOI: `10.1103/physrevlett.112.070502`. URL: `http://dx.doi.org/10.1103/PhysRevLett.112.070502`.

[4] Leo DiCarlo. "Intro/Review of one-and two-qubit state tomography". Presentation slides. Dec. 2020.

[5] *Welcome to Python.org.* URL: `https://www.python.org/`.

[6] *Open-Source Quantum Development.* URL: `https://qiskit.org/`.

[7] *The multi hardware Quantum Technology platform.* 2020. URL: `https://www.quantum-inspire.com/`.

[8] *ibmq_5_yorktown v2.2.5, IBM Quantum team. Retrieved from.* 2020. URL: `https://quantum-computing.ibm.com`.

[9] *ibmq_ourense v1.3.5, IBM Quantum team. Retrieved from.* 2020. URL: `https://quantum-computing.ibm.com`.

[10] *Starmon-5, Quantum Inspire.* URL: `https://www.quantum-inspire.com/backends/starmon-5`.

[11] Juan Carlos Garcia-Escartin and Pedro Chamorro-Posada. *Equivalent Quantum Circuits.* 2011. arXiv: `1110.2998 [quant-ph]`.

Appendix

Main code

```python
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
import qiskit as qk
import math
from qiskit import Aer
import os


from calibration_functions import *
# Functions:
# get_readout_err(experimental_backend, number_of_shots)      find calibrating matrix B
# calibrate_readout_err(exper_expect_val , B)       find calibrated exp_value vector

from transfer_matrix_tools import *
# Functions:
# calculate_gate_fidelity(n, Transfer_matrix_ideal,Transfer_matrix)     n=number f qubits
# calculate_transfer_matrix(input_expected_values,output_expected_values)
# plot_transfer_matrix(Transfer_matrix)

from theoretical_tools import *
# Functions:
# exp_value_braket(statevector,operator)

from backend_tools import *
# Functions:
# IBM_backend('ibm_backend')
# QI_backend('qi_backend' )
# simulator_backend()
# noisy_simulator_backend()


# In[2]:


#BACKEND AND NUMBER OF SHOTS

experimental_backend = IBM_backend('ibmqx2')
number_of_shots = 8192
experiment_name='identity_ibmqx2_8192_20_12'


# In[3]:


states = [np.array([1,0]),np.array([0,1]),1/np.sqrt(2) * np.array([1,1]),           1/np.sqrt(2) * np.array([1,-1]), 1/np.sqrt(2) * np.array([1, complex(0,1)]), 1/np.sqrt(2) * np.array([1, complex(0,-1)])]
possible_directions = ['I', 'X','Y', 'Z']


experiment_directory=os.path.join('data_collected', experiment_name)

#if the folder does not exist, create it
if not os.path.exists(experiment_directory):
    os.makedirs(experiment_directory)


```

```
#PAULI GATES
pauli_g=np.zeros((4,2,2),dtype = np.complex_)
pauli_g[0] = np.array([[1,0],[0,1]])
pauli_g[1] = np.array([[0,1],[1,0]])
pauli_g[2] = np.array([[0,complex(0,-1)],[complex(0,1),0]])
pauli_g[3] = np.array([[1,0],[0,-1]])

sv_backend = Aer.get_backend('statevector_simulator') #statevector for theoretical transfer matrix


#get calibration data

B=get_readout_err(experimental_backend, 8192)

np.save( os.path.join(experiment_directory, 'B'), B )


sv_expected_values=np.zeros((36, 16))
precalibration_exp_values=np.zeros((36, 16))
output_expected_values = np.zeros((36, 16))
input_expected_values = np.zeros((36, 16))


m=0                      # goes from 0 to 35 (36 cardinal states)
for state1 in states:
    for state2 in states:
        d1d2=0           # direction 1 + direction 2 numbering  -  goes from 0 to 15
        d1=0             # direction 1 numbering
        n=0              # goes from 0 to 8 (9 experiments per cardinal state )
        for direction1 in possible_directions:
            d2=0         #direction 2 numbering
            for direction2 in possible_directions:


                input_state_vector=np.kron( state2 , state1)

                total_direction_matrix = np.kron(pauli_g[d2],pauli_g[d1])

                input_expected_values[m,d1d2]=  exp_value_braket(input_state_vector, total_direction_matrix )    # <'Direction2' 'Direction1'> =  <bra|D2D1_matrix   |ket>



                combined_directions = direction1 + direction2
                if combined_directions == 'II' : # For the II case we don't need a circuit
                    output_expected_values[m,0] = 1
                    precalibration_exp_values[m,0] = 1
                    sv_expected_values[m,0]=1
                    d2+=1
                    d1d2+=1
                    continue
                #print("Let's find the expectation value for the", direction1, direction2, 'measurements')

                # Define circuit
                q = qk.QuantumRegister(2)
                c = qk.ClassicalRegister(2)
                circuit = qk.QuantumCircuit(q, c)

                # Do some initialization for the input states
                circuit.initialize(state1, 0)  # Initialize the 0th qubit using a complex vector
                circuit.initialize(state2, 1)  # Initialize the 1st qubit using a complex vector


```

```
                        #INPUT PROCESS

        # The gates for the process
        #circuit.h(q[0])
        #circuit.cx(q[0], q[1])

        #circuit.cz(q[0],q[1])


        ## statevector simulator for theoretical output
        sv_job = qk.execute(circuit, sv_backend)
        sv_result = sv_job.result()
        sv_output = sv_result.get_statevector(circuit, decimals=3)

        sv_expected_values[m,d1d2]  =  exp_value_braket(sv_output, total_direction_matrix )


        d1d2+=1  #iterating d1d2 used for theoretical exp values



        #EXPERIMENT ON THE BACKEND FROM NOW ON

        # Rotate to make measurements in different bases // go on to experiment only if neither direction is I
        if direction1 == 'I' :
            d2+=1
            continue
        elif direction1 == 'X' :
            circuit.ry(-math.pi/2, q[0])
        elif direction1 == 'Y' :
            circuit.rx(math.pi/2, q[0])

        if direction2 == 'I' :
            d2+=1
            continue
        elif direction2 == 'X' :
            circuit.ry(-math.pi/2, q[1])
        elif direction2 == 'Y' :
            circuit.rx(math.pi/2, q[1])


        circuit.measure(q, c)    # Measure both bits

        # Define the experiment


        experimental_job = qk.execute(circuit, backend=experimental_backend, shots=number_of_shots)
        experimental_result = experimental_job.result()

        # Look at the results
        histogram = experimental_result.get_counts(circuit)

        # Add the results to the results matrix
        expected_value=np.zeros(3)
        for state, counts in histogram.items() :
            expected_value[0] += (-1)**(int(state[1]))*int(counts)    # corresponds to IZ
            expected_value[1] += (-1)**(int(state[0]))*int(counts)    # corresponds to ZI
            expected_value[2] += (-1)**(int(state[0])+int(state[1]))*int(counts)  #corresponds to ZZ

        expected_value = expected_value / number_of_shots

        calibrated_exp_val=calibrate_readout_err(expected_value,B)

```

```

                ## Calculation of <U,U>
                output_expected_values[ m ,4*d1+d2] = calibrated_exp_val[2]
                precalibration_exp_values[ m ,4*d1+d2] = expected_value[2]

                ## Calculation of <I,U>
                output_expected_values[ m ,d2] = calibrated_exp_val[1]
                precalibration_exp_values[ m ,d2] = expected_value[1]

                ## Calculation of <U,I>
                output_expected_values[ m ,4*d1] = calibrated_exp_val[0]
                precalibration_exp_values[ m ,4*d1] = expected_value[0]


                #save data
                np.save(os.path.join(experiment_directory, 'output_expected_values') ,output_expected_values)
                np.save(os.path.join(experiment_directory, 'precalibration_exp_values') ,precalibration_exp_values)
                np.save(os.path.join(experiment_directory, 'sv_expected_values' ) ,sv_expected_values)
                np.save(os.path.join(experiment_directory, 'input_expected_values') ,input_expected_values)

                print(' m = ',m,' n = ',n)
                n += 1
                d2+=1
            d1+=1
        m += 1


# In[4]:


th_transfer_matrix = calculate_transfer_matrix( input_expected_values, sv_expected_values )
exp_transfer_matrix= calculate_transfer_matrix( input_expected_values, output_expected_values )
precal_transfer_matrix= calculate_transfer_matrix( input_expected_values, precalibration_exp_values )


# In[5]:


calculate_gate_fidelity(2, th_transfer_matrix,exp_transfer_matrix)


# In[6]:


plot_transfer_matrix(th_transfer_matrix,'Theoretical Transfer Matrix')
plot_transfer_matrix(exp_transfer_matrix,'Experimental Transfer Matrix')
plot_transfer_matrix(precal_transfer_matrix,'Precalibration Transfer Matrix')


# In[ ]:
```

Functions

backend_tools.py

```
from qiskit import Aer
import qiskit as qk

def IBM_backend( ibm_backend ):
    '''
    Backends:
    'ibmq_qasm_simulator',
    'ibmqx2'
    'ibmq_16_melbourne',
    'ibmq_vigo'
    'ibmq_ourense'
    'ibmq_valencia'
    'ibmq_armonk'
    'ibmq_athens'
    'ibmq_santiago'
    '''
    from qiskit import IBMQ
    ibm_token='467524841963
        e35eb595f88a3884dc21cdb9ac157b26a0d976785344b089cf1979c80e27c9c62f31dec260745c7731eb0a08a714c69
        '
    ibm_provider = IBMQ.enable_account(ibm_token)
    return  ibm_provider.get_backend(ibm_backend)







def QI_backend( qi_backend  ):
    '''
    Backends:
    'QX single-node simulator',
    'Spin-2',
    'Starmon-5'
    '''
    from quantuminspire.qiskit import QI
    from quantuminspire.credentials import save_account
    qi_token='7ff8243ba6d4643e4ec1774b7079f8086df7e872'
    save_account(qi_token)
    QI.set_authentication()
    return QI.get_backend(qi_backend) # Possible options: 'QX single-node simulator', '
        Spin-2', 'Starmon-5'


def noisy_simulator_backend():
    '''
    Simulator with noise.
    '''
    from qiskit.test.mock import FakeVigo
    return  FakeVigo()

def simulator_backend():
    '''
    Noise free simulator backend.
    '''
    return Aer.get_backend('qasm_simulator')
```

calibration_functions.py

```
#function to calibrate calculated exp_values
#inputs:
#A: experimental vector expected_values((1,3))  \\ IZ ZI ZZ
#Bd: calibration data B 3x4
import numpy as np
import qiskit as qk
def calibrate_readout_err(expectation_value_vector , calibration_array_B):
    '''
    Function that returns calibrated expectation value vector.
    '''
```

```python
    #offset vector
    b0=np.zeros((3,1))
    b0=calibration_array_B[:,0]
    #B square matrix
    B=calibration_array_B[:,1:4]


    #calculate calibrated values vector
    calibrated=np.zeros(3)
    calibrated=np.dot( B , np.transpose( expectation_value_vector - b0 ) )

    return calibrated




##function get readout
# CALL ONCE IN THE BEGINNING TO GET MATRIX B
# FINDS Bd 3x4 which contains matrix B and offset B0
def get_readout_err(experimental_backend,number_of_shots ):
    '''
    Function that returns the calibration array for a given backend.
    '''
    #DEFINE HELPING ARRAY M_msq = M_lsq = M_corr = M
    M = np.array([[1,1,1,1], [1,1,-1,-1], [1,-1,1,-1], [1,-1,-1,1]])


    results=np.zeros((4,3))
    print('Calibration has begun.\n')
    n=0
    for  i in range(2):
        for j in range(2):

            #initialise circuit
            q = qk.QuantumRegister(2)
            c = qk.ClassicalRegister(2)
            circuit = qk.QuantumCircuit(q, c)

            #initialise 00 01 10 11 state
            circuit.initialize(np.array([1-i,i]), 0)
            circuit.initialize(np.array([1-j,j]), 1)

            circuit.measure(q, c)

            # Define the experiment
            qi_job = qk.execute(circuit, backend=experimental_backend, shots=number_of_shots)
            qi_result = qi_job.result()

            #results
            histogram = qi_result.get_counts(circuit)

            #calculate expected_value IZ ZI ZZ
            expected_value=np.zeros(3)
            for state, counts in histogram.items() :
                expected_value[0] += (-1)**(int(state[1]))*int(counts)              # corresponds to IZ
                expected_value[1] += (-1)**(int(state[0]))*int(counts)              # corresponds to ZI
                expected_value[2] += (-1)**(int(state[0])+int(state[1]))*int(counts)    # corresponds to ZZ

            expected_value=expected_value/number_of_shots

            results[n,:]=expected_value
            n=n+1

    Bd=np.zeros((3,4))

    for i in range(3):
        Bd[i,:]= np.dot( np.linalg.inv(M), results[:,i] )


    B=Bd[:,1:4]
    B = np.linalg.inv(B)
    Bd[:,1:4]=B
    print('Calibration done!\n')
    return Bd
```

theoretical_tools.py

```
import numpy as np
from transfer_matrix_tools import  calculate_transfer_matrix,calculate_gate_fidelity

def exp_value_braket(statevector,operator):
    '''
    Function that returns mathematically calculated expectation value for given
        statevector and operator.
    '''
    return np.dot(  np.conjugate(statevector)    , np.dot ( operator    , statevector) )


def get_bootstrap_standard_error(number_of_qubits , number_of_shots,
    bootstrap_iterations , input_expected_values , output_expected_values ,
    sv_expected_values, exp_input_expected_values):
    '''
    Returns the standard error for the calculation of the transfer matrix.
    '''

    ideal_transfer_matrix=calculate_transfer_matrix(input_expected_values,
        sv_expected_values)
    std= 1/np.sqrt(number_of_shots)  #standard error of each exp_value  / std deviation
        for the bootstrap gaussian

    new_expected_values = np.zeros((6**number_of_qubits,2**(2*number_of_qubits)))  #new
        possible measurements will be added here for each iteration
    new_input_expected_values = np.zeros((6**number_of_qubits,2**(2*number_of_qubits)))
        #new possible measurements will be added here for each iteration
    fidelities=np.zeros(bootstrap_iterations)



    for n in range(bootstrap_iterations):

        for i in range(6**number_of_qubits):
            for j in range(2**(2*number_of_qubits)):
                m = output_expected_values[i,j]    # the average value of the gaussian
                new_expected_values[i,j]=  np.random.normal(m,std)

                m = exp_input_expected_values[i,j]    # the average value of the gaussian
                new_input_expected_values[i,j]=  np.random.normal(m,std)

        # after getting another possible realisation of the experiment we calculate the
            new tr_matrix and fidelity
        new_transfer_matrix = calculate_transfer_matrix(new_input_expected_values,
            new_expected_values) #calculate transfer matrix
        fidelities[n] = calculate_gate_fidelity(number_of_qubits , ideal_transfer_matrix
             , new_transfer_matrix )      #get corresponding fidelity

    return  np.std(fidelities) # std dev of realisations/ std error of fidelities
```

transfer_matrix_tools.py

```python
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np

#mapping according to Leo
new_map  =   ['II','XI','YI','ZI','IX','IY','IZ','XX','XY','XZ','YX','YY','YZ','ZX','ZY'
    ,'ZZ']

#mapping according to Chow
#new_map  =   ['II','IX','IY','IZ','XI','YI','ZI','XX','XY','XZ','YX','YY','YZ','ZX','ZY
    ','ZZ']

def rearrange_transfer_matrix(tr_matrix):
    '''
    Rearranges matrix if the current mapping is as follows:
    ['II','IX','IY','IZ','XI','XX','XY','XZ','YI','YX','YY','YZ','ZI','ZX','ZY','ZZ']

    to the new_map mapping
    '''
    def direction_to_index( direction ):
        if   direction=='I':
            return 0
        elif direction=='X':
            return 1
        elif direction=='Y':
            return 2
        elif direction=='Z':
            return 3

    rearranged_tr_matrix=np.zeros((16,16))
    i=0
    for directionrow in new_map:
        row1= direction_to_index(directionrow[0])
        row2= direction_to_index(directionrow[1])
        j=0
        for directioncol in new_map:
            col1= direction_to_index(directioncol[0])
            col2= direction_to_index(directioncol[1])

            rearranged_tr_matrix[i,j] = tr_matrix[4*row1+row2 , 4*col1+col2]
            j+=1
        i+=1
    return rearranged_tr_matrix



#gate fidelity calculation
def calculate_gate_fidelity(n, Transfer_matrix_ideal,Transfer_matrix):
    d= 2 ** n
    fpro = (np.trace(np.dot(np.matrix(Transfer_matrix_ideal).transpose() ,
        Transfer_matrix))) / d**2
    gate_fidelity= (fpro*d+1)/(d+1)
    return gate_fidelity


#transfer matrix calculation
def calculate_transfer_matrix(input_expected_values,output_expected_values):
    inverse_input_matrix = np.linalg.pinv(np.matrix(input_expected_values.transpose()))
    tr_matrix= np.matrix(output_expected_values).transpose() * np.matrix(
        inverse_input_matrix)
    return rearrange_transfer_matrix(tr_matrix)

#plot transfer matrix
def plot_transfer_matrix(Transfer_matrix, plot_title):

    labelling=new_map


    fig, ax = plt.subplots()
    im = ax.imshow(Transfer_matrix, vmin=-1, vmax=1, interpolation='nearest', cmap=cm.
        RdBu )
    # We want to show all ticks...
    ax.set_xticks(np.arange(len(labelling)))
    ax.set_yticks(np.arange(len(labelling)))
    # ... and label them with the respective list entries
    ax.set_xticklabels(labelling)
    ax.set_yticklabels(labelling)

    # Rotate the tick labels and set their alignment.
    ax.set_title(plot_title)
    plt.show()
```

Code for n-dimensional Quantum Process Tomography

Quantum_Process_Tomography.py

```
import matplotlib.pyplot as plt  # for plotting
import matplotlib.cm as cm  # for colour maps
from qiskit import Aer  # for simulating on own pc
import numpy as np  # for arrays
import qiskit as qk  # for quantum
import math  # for simple math
from progress.bar import IncrementalBar  # so you can see the pc is doing something
# So you can see the progress with the bar run this file in the Terminal by typing: "
    python main.py"


class QuantumProcessTomography:
    """
    This class contains all functions necessary for doing n-dimensional Qubit Process
        Tomography.
    The process you want to do tomography on can be defined in the 'quantum_process()'
        function.
    The calculation size grows dramatically as the number of qubits increases. So make
        sure the
    dimensionality of the tomography is the same as that of the process. So that no
        computational
    resources are being wasted and you can have the results ASAP.
    """
    counter = 0

    def __init__(self, number_of_qubits, number_of_shots=8192):
        """
        :param number_of_qubits: The amount of qubits involved in the Quantum Process
            you want to do QPT on.
        :param number_of_shots: The number of shots to do for every individual
            experiment, default is 8192.

        In this function the tools that are being used throughout the class are being
            initialized.
        """
        self.n = number_of_qubits
        self.backend = Aer.get_backend('qasm_simulator')
        self.shots = number_of_shots

        # Setting up some tools which are used for the calculations
        self.cardinal_states = [np.array([1, 0]),
                                np.array([0, 1]),
                                1/np.sqrt(2) * np.array([1, 1]),
                                1/np.sqrt(2) * np.array([1, -1]),
                                1/np.sqrt(2) * np.array([1, complex(0, 1)]),
                                1/np.sqrt(2) * np.array([1, complex(0, -1)])]
        self.directions = ['I', 'X', 'Y', 'Z']
```

```python
        self.pauli_vector_string = self.compute_pauli_vector_string()  # Contains the combined direction labels

        # Creating initial arrays so they are ready for computations
        self.qubit_inputs = np.zeros((len(self.cardinal_states) ** self.n, self.n, 2), dtype=np.complex_)
        for state_index in range(np.shape(self.qubit_inputs)[0]):
            for qubit_index in range(np.shape(self.qubit_inputs)[1]):
                self.qubit_inputs[state_index, qubit_index, :] = np.array([0, 0])  # Init. with an empty bloch vector

        self.pauli_input_matrix = np.zeros((len(self.directions) ** self.n, len(self.cardinal_states) ** self.n))
        self.pauli_output_matrix = np.zeros((len(self.directions) ** self.n, len(self.cardinal_states) ** self.n))
        self.theoretical_output_matrix = np.zeros((len(self.directions) ** self.n, len(self.cardinal_states) ** self.n))
        self.pauli_transfer_matrix = np.zeros((4 ** self.n, 4 ** self.n))
        self.theoretical_transfer_matrix = np.zeros((4 ** self.n, 4 ** self.n))

        # Initializing the circuit
        self.qubits = qk.QuantumRegister(self.n)
        self.classical_bits = qk.ClassicalRegister(self.n)
        self.circuit = qk.QuantumCircuit(self.qubits, self.classical_bits)  # Changes per experiment

    def compute_pauli_vector_string(self):
        """
        This function is called on initialization, it creates a list which contains the combined directional labels of
        the Pauli vectors. The second for loop works as follows, for every index it tries to find a character at a
        position which it can use, if it cannot find any an 'I' will be used. By looping through all indices in this way
        all combinations of directions are made. This is done similarly to a number system (binary, decimal,
        hexadecimal). With here the I, X, Y, Z representing the characters to form 'numbers'.

        :return: combined_directions
        """
        length = len(self.directions) ** self.n
        combined_directions = []
        start_string = ''
        for i in range(self.n):  # Creating a string "n * I"
            start_string += 'I'

        for index in range(length):  # This for loop creates strings with all possible directions
            leftover_index = index
            new_string_list = list(start_string)
            for position in reversed(range(self.n)):
                if leftover_index >= (3 * 4**position):
                    new_string_list[-1-position] = 'Z'
                    leftover_index -= 3 * 4**position
                elif leftover_index >= (2 * 4**position):
                    new_string_list[-1 - position] = 'Y'
                    leftover_index -= 2 * 4 ** position
                elif leftover_index >= (4**position):
                    new_string_list[-1 - position] = 'X'
                    leftover_index -= 4**position
                elif leftover_index <= 0:
                    break
            new_string = "".join(new_string_list)
            combined_directions.append(new_string)

        return combined_directions

    def get_pauli_matrix(self, direction):
        """
        This function simply returns the Pauli matrices needed depending on the direction given in the input.
        :param direction: The direction you want the pauli-matrix for.
        :return: pauli_matrix, returns either Identity or Pauli X/Y/Z
        """
        pauli_matrix = np.zeros((2, 2))
        if direction == 'I':
            pauli_matrix = np.array([[1, 0], [0, 1]])
        elif direction == 'X':
            pauli_matrix = np.array([[0, 1], [1, 0]])
        elif direction == 'Y':
            pauli_matrix = np.array([[0, complex(0, -1)], [complex(0, 1), 0]])
        elif direction == 'Z':
            pauli_matrix = np.array([[1, 0], [0, -1]])
        return pauli_matrix

    def compute_pauli_input_vector(self, qubit_states):
        """
        In this function the Pauli state vector is calculated using the kron/tensor product.
        :param qubit_states: The input can be characterized by the combined states of the qubits.
        :return: pauli_input_vector, this is the Pauli state vector of the input.
        """
        pauli_input_vector = np.zeros((len(self.directions) ** self.n, 1))

        input_state_vector = qubit_states[0]  # Pick the first element
        for state in qubit_states[1:]:  # calculate the kron product for all states combined
            input_state_vector = np.kron(state, input_state_vector)

        index = 0
        for combined_directions in self.pauli_vector_string:
            directional_matrix = self.get_pauli_matrix(list(combined_directions)[0])  # Pick the first element
            for char in list(combined_directions)[1:]:  # calc. the kron prod. for all possible directional matrices
                directional_matrix = np.kron(self.get_pauli_matrix(char), directional_matrix)

            pauli_input_vector[index] = np.real(np.dot(np.conjugate(input_state_vector), np.dot(directional_matrix, input_state_vector)))
            index += 1

        return pauli_input_vector[:, 0]

    def compute_all_qubit_inputs(self):
        """
        This function computes an array containing all possible qubit inputs. Given that we have n qubits and that we
        want all possible combinations of cardinal states. The operation used to calculate this list is similar to what
        was used in compute_pauli_vector_string(). The second for loop works as follows, for every index it tries to
        find a cardinal state at a position which it can use, if it cannot find any an '0 state' will be used (notice
        that the start_input is first initialized with these states in the first for loop.  By looping through all
        indices in this way all combinations of directions are made.
        :return: qubit_inputs, the list of all combinations of cardinal states for the n qubits.
        """
        qubit_inputs = self.qubit_inputs

        start_input = []

        for i in range(self.n):  # Creating a list "n * np.array([1, 0])"
            start_input.append(np.array([1, 0]))

        length = len(self.cardinal_states) ** self.n
        for state_index in range(length):  # This for loop goes over all possible inputs
            leftover_index = state_index
            new_input = start_input
            for position in reversed(range(self.n)):
                if leftover_index >= (5 * 6**position):
                    new_input[position] = self.cardinal_states[5]
                    leftover_index -= 5 * 6**position
                elif leftover_index >= (4 * 6**position):
                    new_input[position] = self.cardinal_states[4]
                    leftover_index -= 4 * 6**position
                elif leftover_index >= (3 * 6**position):
                    new_input[position] = self.cardinal_states[3]
                    leftover_index -= 3 * 6**position
                elif leftover_index >= (2 * 6**position):
                    new_input[position] = self.cardinal_states[2]
                    leftover_index -= 2 * 6**position
                elif leftover_index >= (6**position):
                    new_input[position] = self.cardinal_states[1]
                    leftover_index -= 6**position
                elif leftover_index <= 0:
                    break
            qubit_inputs[state_index, :, :] = new_input

        self.qubit_inputs = qubit_inputs
        return qubit_inputs

    def compute_pauli_input_matrix(self):
        """
        This function simply returns an array containing all possible input Pauli State vectors. It does this by calling
        the compute_pauli_input_vector() on every input which was found with compute_all_qubit_inputs().
        :return: pauli_input_matrix
        """
        pauli_input_matrix = self.pauli_input_matrix
        self.compute_all_qubit_inputs()

        state_index = 0
        for new_input in self.qubit_inputs:
            pauli_input_matrix[:, state_index] = self.compute_pauli_input_vector(new_input)
            state_index += 1

        self.pauli_input_matrix = pauli_input_matrix
        return pauli_input_matrix

    def clear_circuit(self):
        """
        This function initializes the circuit and bits again to clear out the old circuit and bits.
        :return: none
        """
        self.qubits = qk.QuantumRegister(self.n)
        self.classical_bits = qk.ClassicalRegister(self.n)
        self.circuit = qk.QuantumCircuit(self.qubits, self.classical_bits)

    def setup_circuit(self, qubit_states):
        """
        This function does some initialization for the input states that are given.
        :param qubit_states: The input states for the QPT.
        :return: none
        """
        qubit_index = 0
        for state in qubit_states:
            self.circuit.initialize(state, qubit_index)  # Initialize the i-th qubit using a complex vector
            qubit_index += 1

    def quantum_process(self):
        """
        This function is used to do the actual quantum process that we are doing QPT on.
             CHANGE THIS AS YOU SEE FIT. You
        can do QPT on any process as you wish. Make sure that the dimension of the
            process defined here is the same as
        that when you create an instance of this QubitProcessTomography class.
        :return: none
        """
        for i in range(self.n):
            self.circuit.h(self.qubits[i])  # Do a Hadadamard gate on every input

    def compute_pauli_output_vector(self, qubit_states):
        """
        I would say this function is at the core of the class. In this function the
            backend is actually used and the
        experiments are being done. From the backend we get a histogram containing the
            states and amount of counts for
        each specific state. Based on the measurements for X, Y and Z and the non-
            measurement I we can then calculate
        the expected values in the Pauli state vector.
        :param qubit_states: The input qubit state(s) which are used to setup the
            circuits and do the experiments.
        :return: pauli_output_vector, this is the Pauli state vector of the output.
        """
        pauli_output_vector = np.zeros((len(self.directions) ** self.n, 1))
        index = 0
        for combined_directions in self.pauli_vector_string:
            # Start with clearing out the old circuit, setting up and adding the process
            self.clear_circuit()
            self.setup_circuit(qubit_states)
            self.quantum_process()

            # These lists are used so that the qubits with direction 'I' are not
                measured
            measured_qubits = []
            measured_bits = []

            for i in range(len(combined_directions)):
                if combined_directions[i] == 'X':  # Do an X measurement
                    self.circuit.ry(-math.pi / 2, self.qubits[i])
                    measured_qubits.append(self.qubits[i])
                    measured_bits.append(self.classical_bits[i])
                elif combined_directions[i] == 'Y':  # Do an Y measurement
                    self.circuit.rx(math.pi / 2, self.qubits[i])
                    measured_qubits.append(self.qubits[i])
                    measured_bits.append(self.classical_bits[i])
                elif combined_directions[i] == 'Z':  # Do an Z measurement
                    measured_qubits.append(self.qubits[i])
                    measured_bits.append(self.classical_bits[i])

            self.circuit.measure(measured_qubits, measured_bits)
            experimental_job = qk.execute(self.circuit, self.backend, shots=self.shots)
            experimental_result = experimental_job.result()
            histogram = experimental_result.get_counts(self.circuit)

            expected_value = 0
            for state, counts in histogram.items():
                bitsum = 0
                for bit in state:
                    bitsum += int(bit)
                expected_value += ((-1)**bitsum)*int(counts)
            expected_value = expected_value / self.shots
            pauli_output_vector[index] = expected_value
            index += 1

        return pauli_output_vector[:, 0]

```

```python
    def compute_pauli_output_matrix(self):
        """
        This function simply returns an array containing all possible output Pauli State vectors. It does this by
        calling the compute_pauli_output_vector() on every input which was found with compute_all_qubit_inputs(). In
        this function there is also a bar which is used to create a progress bar when the main.py is run in a terminal.
        This is quite handy as calculation time dramatically increases with the amount of qubits n.
        :return: pauli_output_matrix
        """
        pauli_output_matrix = self.pauli_output_matrix

        state_index = 0
        bar = IncrementalBar(': Computing output matrix...', max=len(self.qubit_inputs))  # For tracking progress
        for new_input in self.qubit_inputs:
            pauli_output_matrix[:, state_index] = self.compute_pauli_output_vector(new_input)
            state_index += 1
            bar.next()  # For tracking progress
        bar.finish()  # Finished the big calculation!

        self.pauli_output_matrix = pauli_output_matrix
        return pauli_output_matrix

    def compute_theoretical_output_vector(self, qubit_states):
        """
        This function is similar to the calculation for the pauli input/output vectors. But now we use a specific
        statevector_simulator backend. This allows to get a theoretical output vector, which has no noise.
        :return: theoretical_output_vector, this is the theoretical Pauli state vector of the output.
        """
        sv_backend = Aer.get_backend('statevector_simulator')  # statevector for theoretical transfer matrix
        theoretical_output_vector = np.zeros((len(self.directions) ** self.n, 1))

        index = 0
        for combined_directions in self.pauli_vector_string:
            # Start with clearing out the old circuit, setting up and adding the process
            self.clear_circuit()
            self.setup_circuit(qubit_states)
            self.quantum_process()

            # Calculate the directional_matrix
            directional_matrix = self.get_pauli_matrix(list(combined_directions)[0])  # Pick the first element
            for char in list(combined_directions)[1:]:  # calc. the kron prod. for all possible directional matrices
                directional_matrix = np.kron(self.get_pauli_matrix(char), directional_matrix)

            # Now do a statevector analysis to determine the theoretical output
            statevector_job = qk.execute(self.circuit, sv_backend)
            statevector_result = statevector_job.result()
            statevector_output = statevector_result.get_statevector(self.circuit, decimals=3)
            theoretical_output_vector[index, 0] = np.real(np.dot(np.conjugate(statevector_output),
                                                                  np.dot(directional_matrix, statevector_output)))
            index += 1

        return theoretical_output_vector[:, 0]

    def compute_theoretical_output_matrix(self):
        """
        This function simply returns an array containing all possible theoretical output Pauli State vectors. It does
        this by calling the theoretical_output_vector() on every input which was found with compute_all_qubit_inputs().
        :return: theoretical_output_matrix
        """
        theoretical_output_matrix = self.theoretical_output_matrix

        state_index = 0
        bar = IncrementalBar(': Computing theoretical output matrix...', max=len(self.qubit_inputs))  # For progress
        for new_input in self.qubit_inputs:
            theoretical_output_matrix[:, state_index] = self.compute_theoretical_output_vector(new_input)
            state_index += 1
            bar.next()  # For tracking progress
        bar.finish()  # Finished the calculation

        self.theoretical_output_matrix = theoretical_output_matrix
        return theoretical_output_matrix

    def compute_pauli_transfer_matrix(self):
        """
        Now that we have done most of the complicated algorithms this one can be rather simple. Using the
        pauli_input_matrix and pauli_output_matrix we can calculate the Pauli Transfer Matrix which characterizes the
        Quantum Process which was given in quantum_process().
        :return: pauli_transfer_matrix
        """
        pauli_input_matrix = self.compute_pauli_input_matrix()
        inverse_pauli_input_matrix = np.linalg.pinv(pauli_input_matrix)
        pauli_output_matrix = self.compute_pauli_output_matrix()
        pauli_transfer_matrix = pauli_output_matrix * np.matrix(inverse_pauli_input_matrix)
        self.pauli_transfer_matrix = pauli_transfer_matrix
        return pauli_transfer_matrix

    def compute_theoretical_transfer_matrix(self):
        """
        Similar to compute_pauli_transfer_matrix(), calculating the theoretical_transfer_matrix is now rather simple.
        Using the pauli_input_matrix and theoretical_output_matrix we can calculate the Theoretical Transfer Matrix
        which characterizes the Quantum Process perfectly.
        :return: pauli_transfer_matrix
        """
        pauli_input_matrix = self.compute_pauli_input_matrix()
        inverse_pauli_input_matrix = np.linalg.pinv(pauli_input_matrix)
        theoretical_output_matrix = self.compute_theoretical_output_matrix()
        theoretical_transfer_matrix = theoretical_output_matrix * np.matrix(inverse_pauli_input_matrix)
        self.theoretical_transfer_matrix = theoretical_transfer_matrix
        return theoretical_transfer_matrix

    def compute_fidelity(self):
        """
        In this function the gate fidelity is calculated. Do a call to compute_pauli_transfer_matrix() and
        compute_theoretical_transfer_matrix() before this.
        :return: gate_fidelity
        """
        dimension = 2 ** self.n
        process_fidelity = (np.trace(np.dot(np.matrix(self.theoretical_transfer_matrix).transpose(),
                                                    self.pauli_transfer_matrix))) / dimension ** 2
        average_gate_fidelity = (process_fidelity * dimension + 1) / (dimension + 1)

        return average_gate_fidelity

    def plot_pauli_transfer_matrix(self, plot_title):
        """
        Using this we can visualize the QPT in a colour map plot.
        :param plot_title: The title you want above the plot
        :return: none
        """
        labelling = self.pauli_vector_string

        fig, ax = plt.subplots()
        ax.imshow(self.pauli_transfer_matrix, vmin=-1, vmax=1, interpolation='nearest', cmap=cm.RdBu)
        # We want to show all ticks...
        ax.set_xticks(np.arange(len(labelling)))
        ax.set_yticks(np.arange(len(labelling)))
        # ... and label them with the respective list entries
        ax.set_xticklabels(labelling)
        ax.set_yticklabels(labelling)
        # Rotate the x-tick labels so they don't overlap
        plt.xticks(rotation=90)

        ax.set_title(plot_title)
        plt.show(block=False)

    def plot_theoretical_transfer_matrix(self, plot_title):
        """
        Using this we can visualize the theoretical QPT in a colour map plot.
        :param plot_title: The title you want above the plot
        :return: none
        """
        labelling = self.pauli_vector_string

        fig, ax = plt.subplots()
        ax.imshow(self.theoretical_transfer_matrix, vmin=-1, vmax=1, interpolation='nearest', cmap=cm.RdBu)
        # We want to show all ticks...
        ax.set_xticks(np.arange(len(labelling)))
        ax.set_yticks(np.arange(len(labelling)))
        # ... and label them with the respective list entries
        ax.set_xticklabels(labelling)
        ax.set_yticklabels(labelling)
        # Rotate the x-tick labels so they don't overlap
        plt.xticks(rotation=90)

        ax.set_title(plot_title)
        plt.show(block=False)


# -------------------------------------- End of the QubitProcessTomography class --------------------------------------
```

main.py

```
from Quantum_Process_Tomography import *
"""
Now that all functionality is incorporated in the QPT class in the Qubit_Process_Tomography file all that is left to do
is make an instance with the amount of qubits you need and call compute_pauli_transfer_matrix(). If you do this from a
terminal, you can get a nice progressbar to show you how far the calculation is.
"""
```

```python

QPT = QuantumProcessTomography(2)
QPT.compute_pauli_transfer_matrix()
QPT.plot_pauli_transfer_matrix('Two qubits simple hadamard process')

QPT.compute_theoretical_transfer_matrix()
QPT.plot_theoretical_transfer_matrix('Two qubits simple hadamard process (theoretical)')
plt.show()  # This is used so plots stay in place
```