Simulations of qubit communication in prepare-and-measure and Bell scenarios

Iñaki Ortiz de Landaluce Sáiz
Aido Cortés Alcaráz

Supervised by Gael Sentís Herrera (Universitat Autònoma de Barcelona)

A project submitted for the Postgraduate Degree in Quantum Engineering

Abstract

In the early years of the current century, a breakthrough was made in quantum information theory by David Bacon and Ben Toner, who developed a quantum communication protocol showing that any prediction based on projective measurements (PVMs) on a qubit could be simulated by communicating only two classical bits. This result has been very recently extended by Martin Renner, Armin Tavakoli and Marco Tulio Quintino to positive operator-valued measures (POVMs) without loss of generality, keeping the classical cost of a qubit transmission still as two bits. In this project we have simulated such extended protocol classically and compared its outcomes against a quantum simulator and a noisy intermediate-scale quantum computer, to show how two bits of communication are enough to reproduce all quantum correlations associated to arbitrary POVMs applied to any prepare-and-measure scenario. In addition, we have also reproduced the probability distributions of a Bell experiment with an entangled two-qubit pair state using a novel classical protocol also proposed by the authors. With this investigation we give rise to explore and understand computationally some fundamental limits of quantum over classical information theories.

Contents

1 Introduction **4**
- 1.1 Objective . . . 4
- 1.2 Positive operator-valued measures . . . 4
- 1.3 Prepare-and-measure scenario . . . 5
- 1.4 Bell scenario . . . 6
- 1.5 Classical simulation protocols . . . 8
  - 1.5.1 Prepare-and-measure with one qubit . . . 8
  - 1.5.2 Bell with singlet state . . . 9

2 Methodology **11**
- 2.1 State preparation . . . 11
- 2.2 Rank-1 POVMs implementation . . . 12
  - 2.2.1 Measurement in classical simulation protocols . . . 14
  - 2.2.2 Measurement in quantum circuit model . . . 15
- 2.3 Prepare-and-measure simulations . . . 16
  - 2.3.1 Classical transmission of one qubit . . . 16
  - 2.3.2 Quantum circuit counterpart . . . 17
- 2.4 Bell simulations . . . 18
  - 2.4.1 Bell singlet state with projective measurements . . . 18
  - 2.4.2 CHSH inequality . . . 19

3 Results **20**
- 3.1 Random state and measurement preparation . . . 20
- 3.2 Prepare-and-measure simulation outcomes . . . 21
- 3.3 Bell simulation outcomes . . . 23

4 Conclusion **29**

5 Acknowledgements **30**

References **31**

Appendices **33**

Appendix A Source Code Listings **33**
- A.1 qubit.py . . . 33
- A.2 measurement.py . . . 36
- A.3 random.py . . . 42
- A.4 observable.py . . . 45
- A.5 qudit.py . . . 46
- A.6 bell.py . . . 47
- A.7 classical.py . . . 49
- A.8 quantum.py . . . 59
- A.9 main.py . . . 60

Appendix B Simulator and Quantum Computer benchmarking 80

1 Introduction

1.1 Objective

Quantum information theory provides a framework to quantify the power of quantum theory compared to Shannon's classical communication theory [Sha48]. Over the last decades, the field has flourished compelled to set realistic boundaries to the promises of quantum advantages in fields like quantum communication and computing. An important feature of quantum theory lies in the statistical correlations produced by local measurements of a quantum system. The simplest example of quantum correlations are the ones produced by projective measurements on a maximally-entangled state of two qubits, also known as a Bell pair state. Such correlations are the basic resource of bipartite quantum information theory, where various equivalences are known: one shared Bell pair plus two bits of classical information can be used to teleport one qubit and, conversely, one shared Bell pair state plus a single qubit can be used to send two bits of classical information via superdense coding. Exploring the fundamental limits of quantum over classical advantages in these scenarios is crucial, and it will be the primary objective of this work.

In the early years of the current century, Ben Toner and David Bacon developed a protocol which proves that any prediction based on projective measurements on an entangled Bell pair state could be simulated by communicating only two classical bits [TB03]. Very recently, Martin Renner, Armin Tavakoli and Marco Tulio Quintino, have extended such result to a most generalised set of measurements, the positive operator-valued measures [RTQ23]. Following up such generalization, we will show via computer-based experiments that a qubit transmission can be simulated classically with a total cost of two bits for any general measurement, either in a prepare-and-measure or an entanglement scenario. This will prove experimentally that the protocols described in [RTQ23] can be reproduced computationally using classical computers, and that the probability distributions obtained can be compared against the ones resulting from performing generalized measurements with existing quantum simulators and noisy intermediate-scale quantum computers.

Before starting to describe the different computational experiments we have carried out to achieve such goal, it is worth spending next sections to introduce some preliminary concepts and notations that have been used extensively throughout this work, specifically the definition of positive operator-valued measures, the set-up of both the prepare-and-measure and Bell scenarios and the particular details of the classical simulation protocols applied to such scenarios.

1.2 Positive operator-valued measures

Even when most of the introductory textbooks on quantum mechanics describe the measurement postulates using projective measurements only, there exists a more general and less restrictive set of measurements called Positive Operator-

Valued Measures or POVMs, see [NC00][Per95]. The underlying formalism behind POVMs is uniquely well adapted for some applications where the main focus is on describing the probabilities of the different measurement outcomes rather than on the post-measurement state of the system. This is of particular interest in quantum communication and quantum information, where a more comprehensive formalism for the description of the measurements is needed, and highlights how important are the results from [RTQ23], where all the results are extended to POVMs without any loss of generalisation regarding the the classical simulation cost of the protocol. For reference, we define explicitely a POVM as a set

$$\mathbb{P}_N = \{B_k\} \qquad \forall k = 1, ..., N \tag{1}$$

of positive semidefinite operators acting on a Hilbert space $\mathcal{H}$ of dimension d_Q, which satisfies the closure property

$$\sum_{k=1}^{N} B_k = \mathbb{1} \tag{2}$$

The operator B_k is called a POVM element, and it is associated to the outcome k of the measurement. In this work we will use extensively the property which states that every qubit POVM can be written as a coarse graining of rank-1 projectors [Bar02], such that we will restrict our POVM calculations to the case of rank-1 projectors.

1.3 Prepare-and-measure scenario

As for many other communication protocols we have two well-known characters playing different roles in the quantum prepare-and-measure set-up: Alice and Bob. The prepare-and-measure scenario starts with Alice preparing a random quantum state and sending it to Bob. In a general set-up, i.e. not restricted to single qubit communication, the state prepared by Alice is a state of dimension d_Q described by a positive semidefinite density matrix $\rho \in \mathcal{L}(\mathbb{C}_{d_Q}), \rho \geq 0$ with unit trace $tr(\rho) = 1$. Once the state is prepared and communicated by Alice, Bob then receives it and performs a random quantum measurement on it, obtaining an outcome k. In the case of general POVM measurements $\mathbb{P}_N = \{B_k\}$, the probability of outcome k when performing the measurement on the state ρ is given by the Born's rule

$$p_Q(k|\rho, \{B_k\}) = tr(\rho B_k) \tag{3}$$

In the context of this work, we are interested in a counterpart of the quantum prepare-and-measure scenario, where the probability distributions predicted by the quantum theory (3) are reproduced classically. All these classical counterparts of existing quantum protocols, see examples in [CGM00], [TB03] and [RTQ23], require a shared randomness λ among Alice and Bob subject to

some probability function $\pi(\lambda)$ to correlate their classical communication strategies. As it is not possible to reproduce these correlations without communication, a classical message c, encoding classically the quantum state ρ and taking its value from a d_C-valued alphabet $\{1, ..., d_C\}$, is also required. Alice's actions are therefore described by the conditional probability $p_A(c|\rho, \lambda)$, whereas Bob's actions are similarly described by the conditional probability $p_B(k|\{B_k\}, c, \lambda)$. If we consider both probabilities, the correlations from the classical counterpart become

$$p_C(k|\rho, \{B_k\}) = \int_\lambda d\lambda \ \pi(\lambda) \sum_{c=1}^{d_C} p_A(c|\rho, \lambda) \ p_B(k|\{B_k\}, c, \lambda) \tag{4}$$

Given Equations (3) and (4), the classical simulation would be considered successful when, for every random state and POVM, the classical probability distribution reproduces the quantum predictions, i.e.

$$\forall \rho, \{B_k\} : \quad p_C(k|\rho, \{B_k\}) = p_Q(k|\rho, \{B_k\}) \tag{5}$$

1.4 Bell scenario

In a Bell scenario, there is a bipartite quantum system of two entangled and separated qudits, one with Alice and another one with Bob. Alice chooses a random local measurement A_x among two possible observables $\{A_0, A_1\}$, and produces an output a_x according to the distribution of her measurement elements. Following the same procedure, Bob chooses his own random local measurement B_y among two possible observables $\{B_0, B_1\}$, and produces an outcome b_y. Even if both outcomes appear random, their joint probabilities $p_{A_x,B_y}(a_x, b_y)$ are correlated. We refer to these correlations as Bell correlations.

Similarly to the prepare-and-measure case described in Section 1.3, it is not possible to reproduce the correlations using a classical protocol with shared random variables without allowing classical communication among Alice and Bob once they have selected their measurements, see [Bel64]. The main question here is to determine how much classical communication is needed to reproduce the probability distributions.

As [RTQ23] shows, it is straight-forward to adapt the prepare-and-measure classical scenario to any entangled qudit-qubit state. Here Alice chooses a random local POVM on a d_Q-dimensional quantum system, and produces an output according to the marginal distribution of her POVM elements. Based on her output, she computes Bob's entangled qubit post-measurement state, which is sent to Bob using the prepare-and-measure scenario. Given that Bob's post-measurement qubit state is communicated by Alice using the existing prepare-and-measure classical protocol, the classical cost of the qubit transmission will be exactly the same: two bits.

If we restrict our quantum system to a bipartite state of two qubits and local and projection valued measures, the maximally entangled states in such scenario are the famous Bell states $|\Phi_{ij}\rangle$ as follows

$$\begin{aligned}
\left|\Phi^{+}\right\rangle &:= \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \\
\left|\Phi^{-}\right\rangle &:= \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle) \\
\left|\Psi^{+}\right\rangle &:= \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle) \\
\left|\Psi^{-}\right\rangle &:= \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)
\end{aligned} \tag{6}$$

Each local projective measurement has two eigenvalues, either $|a_x\rangle$ or $|b_y\rangle$, with outcomes $a_x, b_y = \pm 1$ respectively. In this scenario, the joint probabilities can be defined as

$$p_{A_x,B_y}(a_x, b_y) = tr[|a_x\rangle \langle a_x| \otimes |b_y\rangle \langle b_y|\Phi_{ij}\rangle \langle \Phi_{ij}|] \tag{7}$$

The expected values for a given set of observables (A_x, B_y) can be defined from the joint probabilities as follows

$$\begin{aligned}
\mathbb{E}[A_x, B_y] := p_{A_x,B_y}(+1,+1) \; - p_{A_x,B_y}(+1,-1) \\
-p_{A_x,B_y}(-1,+1) \; + p_{A_x,B_y}(-1,-1)
\end{aligned} \tag{8}$$

which also leads to the famous Clauser-Horne-Shimony-Holt expression

$$CHSH := \mathbb{E}[A_0, B_0] \; + \mathbb{E}[A_1, B_0] \; + \mathbb{E}[A_0, B_1] \; - \mathbb{E}[A_1, B_1] \tag{9}$$

The classical protocol described in [RTQ23] using a Bell singlet state $|\Psi^{-}\rangle$, proves that a classical communication of one single bit is enough to reproduce the joint probabilities when Alice performs projective measurements an Bob can either perform projective or positive operator-valued measurements. Similarly as in the prepare and measure scenario, the classical probability distribution can be defined as

$$p_C(a_x, b_y|A_x, B_y) = \int_\lambda d\lambda \; \pi(\lambda) p_A(c|A_x, \lambda) \; p_B(b_y|B_y, c, \lambda) \tag{10}$$

Given Equations (7) and (10), the classical simulation would be considered successful when, given a bipartite singlet state $|\Psi^{-}\rangle$, for every set of observables (A_x, B_y), the classical probability distribution reproduces the quantum predictions, i.e.

$$\forall A_x, B_y : \quad p_C(a_x, b_y|A_x, B_y) = p_{A_x,B_y}(a_x, b_y) \tag{11}$$

1.5 Classical simulation protocols

1.5.1 Prepare-and-measure with one qubit

The classical prepare-and-measure protocol proposed by Renner, Tavakoli and Quintino [RTQ23] is restricted to qubits ($d_Q = 2$), and makes an extensive use of the geometrical properties of a qubit state in the Bloch sphere. Since mixed qubit states are convex combinations of pure states, the protocol is also restricted to the usage of pure states. Toner and Bacon proved that making a further restriction to projective measurements, i.e. $B_k^2 = B_k$, the classical simulation cost was upper bounded by two classical bits ($d_C = 2$) [TB03], but [RTQ23] generalizes the results to positive operator-valued measures with a minimal and therefore necessary classical cost of two bits. Finally, the protocol is also restricted without any loss of generality to POVMs proportional to rank-1 projectors, following results in [Bar02].

In Bloch notation, qubit states ρ are represented by three-dimensional real normalized vectors $\vec{x} \in \mathbb{R}^3$, and rank-1 POVM projectors as

$$B_k = 2p_k \left|\vec{y}_k\right\rangle \left\langle\vec{y}_k\right| \tag{12}$$

where $p_k \geq 0$, $\sum_{k=1}^{N} p_k = 1$ and $\left|\vec{y}_k\right\rangle \left\langle\vec{y}_k\right| = (\mathbb{1} + \vec{y}_k \cdot \vec{\sigma})/2$ for some normalized vector $\vec{y}_k \in \mathbb{R}^3$, such that

$$tr(\rho B_k) = p_k(1 + \vec{x} \cdot \vec{y}_k) \tag{13}$$

Two additional functions need to be defined prior to make the classical protocol steps explicit, these are the Heaviside function

$$H(z) = \begin{cases} 1 & \text{when } z \geq 0 \\ 0 & \text{when } z < 0 \end{cases} \tag{14}$$

and $\Theta(z) := z \cdot H(z)$. Under all these considerations, the protocol, as defined in [RTQ23] and sketched in Figure 1, is literally as follows:

1. Alice and Bob share two normalized vectors $\vec{\lambda}_1, \vec{\lambda}_2 \in \mathbb{R}^3$, which are uniformly and independently distributed on the unit radius sphere S_2.

2. Instead of sending a pure qubit $\rho = (\mathbb{1} + \vec{x} \cdot \vec{\sigma})/2$, Alice prepares two bits via the formula $c_1 = H(\vec{x} \cdot \vec{\lambda}_1)$ and $c_2 = H(\vec{x} \cdot \vec{\lambda}_2)$ and sends them to Bob.

3. Bob flips each vector $\vec{\lambda}_i$ when the corresponding bit c_i is zero. This is equivalent to set $\vec{\lambda}_i' := (-1)^{1+c_i}\vec{\lambda}_i$.

4. Instead of performing a POVM with elements $B_k = 2p_k \left|\vec{y}_k\right\rangle \left\langle\vec{y}_k\right|$, Bob picks one vector $\vec{y}_k$ from the set $\{\vec{y}_k\}$ according to the probabilities $\{p_k\}$. Then he sets $\vec{\lambda} := \vec{\lambda}_1'$ if $|\vec{\lambda}_1' \cdot \vec{y}_k| \geq |\vec{\lambda}_2' \cdot \vec{y}_k|$ and $\vec{\lambda} := \vec{\lambda}_2'$ otherwise. Finally, Bob outputs k with probability

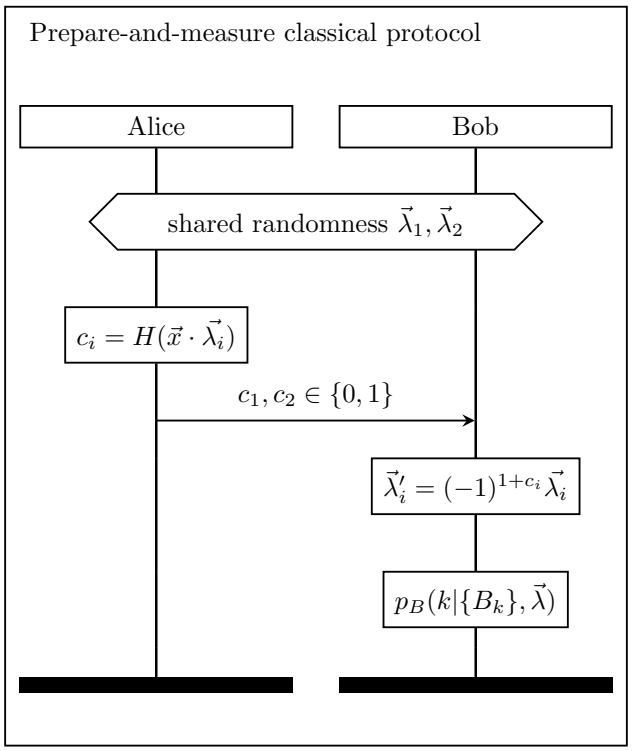

Figure 1: Classical prepare-and-measure protocol sequence: Alice sends two bits c_1 and c_2 resulting from projecting the qubit state's Bloch vector $\vec{x}$ with respect to the shared random vectors $\vec{\lambda}_1$ and $\vec{\lambda}_2$, through a classical channel, Bob flips the shared random vectors when necessary, and finally computes classical probability outcomes according to Equation (15).

$$p_B(k|\{B_k\}, \vec{\lambda}) = \frac{p_k \ \Theta(\vec{y}_k \cdot \vec{\lambda})}{\sum_j^N p_j \ \Theta(\vec{y}_j \cdot \vec{\lambda})} \tag{15}$$

1.5.2 Bell with singlet state

Toner and Bacon [TB03] proved that only a single bit was required to simulate classically local projective measurements on a qubit pair in a singlet state $|\psi^-\rangle = (|01\rangle - |10\rangle)/\sqrt{2}$. Renner, Tavakoli and Quintino [RTQ23] have again extended the result being Alice restricted to local projective measurements with outcomes $a = \pm 1$, and Bob allowed to perform any arbitrary POVM measure. The steps for this protocol, sketched in Figure 2, are the following:

1. Alice and Bob share two normalized vectors $\vec{\lambda}'_1, \vec{\lambda}_2 \in \mathbb{R}^3$, which are uniformly and independently distributed on the unit radius sphere S_2.
2. Instead of performing a measurement with projectors $|\pm\vec{x}\rangle \langle\pm\vec{x}| = (\mathbb{1} + \vec{x} \cdot$

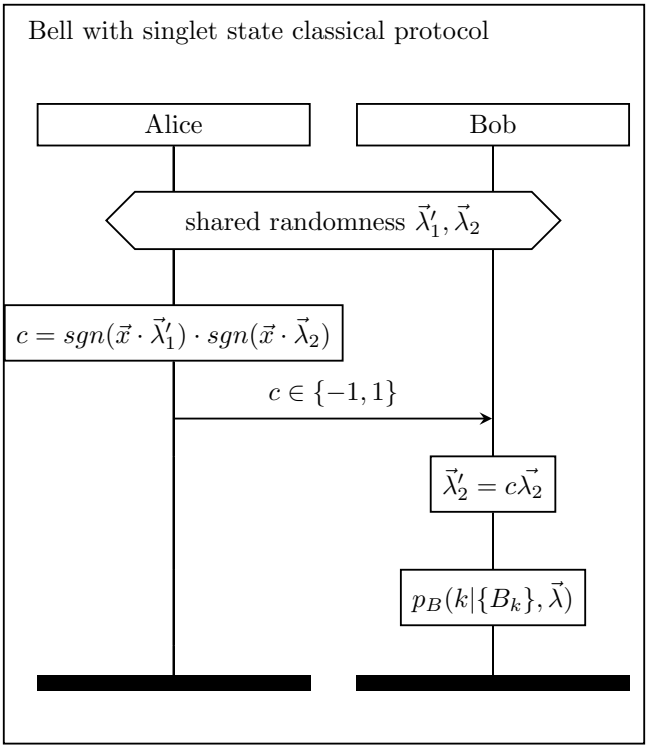

Figure 2: Classical Bell with singlet state protocol sequence: Alice sends one bit c resulting from projecting the associated projective measurement's Bloch vector $\vec{x}$ with respect to the shared random vectors $\vec{\lambda}_1$ and $\vec{\lambda}_2$ through a classical channel, Bob flips the shared random vectors when necessary, and finally computes classical probability outcomes according to the response function from the prepare-and measure protocol, see Equation (15).

$\vec{\sigma})/2$, Alice outputs $a = -sgn(\vec{x} \cdot \vec{\lambda}_1')$ and sends the bit $c = sgn(\vec{x} \cdot \vec{\lambda}_1') \cdot sgn(\vec{x} \cdot \vec{\lambda}_2)$ to Bob, where

$$sgn(z) = \begin{cases} 1 & \text{when } z \geq 0 \\ -1 & \text{when } z < 0 \end{cases} \tag{16}$$

3. Bob flips the vector $\vec{\lambda}_2$ if and only if $c = -1$, i.e. he sets $\vec{\lambda}_2' := c\vec{\lambda}_2$.

4. Same as step 4 in the previous prepare-and-measure protocol.

It can be proved that when Alice outputs $a = +1$, $\vec{\lambda}_1'$ and $\vec{\lambda}_2'$ are distributed on S_2 according to $\rho(\vec{\lambda}_i') = H(-\vec{x} \cdot \vec{\lambda}_i')/(2\pi)$, which corresponds to a classical description of Bob's post-measurement state $-\vec{x}$ and, conversely, when Alice outputs $a = -1$, the random vectors are distributed according to a distribution corresponding to Bob's post-measurement state $\vec{x}$, such that Bob can apply the same response function as in the prepare-and-measure protocol.

2 Methodology

In this section we will discuss the different methods required to effectively simulate the classical protocols described in Section 1.5 such that, for any choice of qubit pure states and rank-1 POVMs, the quantum predictions in Equation (3) are reproduced classically following Equation (4), i.e.

$$\forall \rho, \{B_k\}: \quad tr(\rho B_k) = \int_\lambda d\lambda \; \pi(\lambda) \sum_{c=1}^{d_C} p_A(c|\rho, \lambda) \; p_B(k|\{B_k\}, c, \lambda) \tag{17}$$

and, for a Bell singlet state $|\Psi^-\rangle$ and any choice of local projective measurements, the quantum predictions in Equation (7) are reproduced classically following Equation (10), i.e.

$$\forall A_x, B_y : |\, \langle \Psi_{ij}|a_x\rangle \otimes |b_y\rangle \,|^2 = \int_\lambda d\lambda \; \pi(\lambda) p_A(c|A_x, \lambda) \; p_B(b_y|B_y, c, \lambda) \tag{18}$$

Even when the main goal is to carry out these simulations using standard classical computations, some particular results will be confronted with the outcomes from quantum simulators and quantum computers, therefore there will be specific subsections devoted to the implementation of generalized measurements in a quantum circuit model.

2.1 State preparation

In the classical simulations, the state preparation would require to produce a random qubit pure state first, and then to compute the corresponding Bloch vector to be used later by Alice.

In the prepare-and-measure scenario, Alice uses the Bloch vector representation of the qubit's pure state, and Bob's POVMs, proportional to rank-1 projectors, are expressed as the outer product of the associated Bloch vectors. In the Bell scenario, Alice also uses the Bloch vector corresponding to the local projective measurement projectors. In addition, in all these protocols, Alice and Bob share two random normalized vectors $\vec{\lambda}_1, \vec{\lambda}_2 \in \mathbb{R}^3$, which must be uniformly and independently distributed on the unit sphere S_2, which is analogous to generate random Bloch vectors in the Bloch sphere. Given the fact that the classical probabilities obtained with the different protocols must be equivalent to the quantum probabilities for any random state and POVM measure, and that generating a true shared randomness is a fundamental element of the classical protocols, it is of key importance to be able prepare random normalized vectors uniformly distributed along the unit radius sphere S_2. Hence, the randomized qubit state preparation in the form of Bloch vectors is not only the building block for the state preparation itself, but plays also a critical role in the measurement construction and the shared randomness creation.

To produce a random qubit pure state we should obtain a random unitary matrix and then apply the unitary transformation to the zero qubit state,

resembling the time evolution of a qubit from a zero initial state. The random unitary matrix can be generated by just building a matrix of normally distributed complex numbers, and then apply the Gram-Schmidt QR decomposition to orthogonalize the matrix, see [Ozo09], [ZK94].

The resulting random qubit state distribution can be validated using the corresponding Bloch vector distribution along the unit radius sphere. A tessellation scheme called HEALPix [Gór+05], which produces a hierarchical and equal-area iso-latitude pixelation of a sphere, could be used to show that the random Bloch vectors are uniformly and independently distributed in the Bloch sphere. Given that each pixel in HEALPix covers the same surface area as every other pixel (see Figure 3), we can group the Bloch vector distribution along the different pixel indices (see Figure 4), and check whether the resulting distribution is uniform as expected.

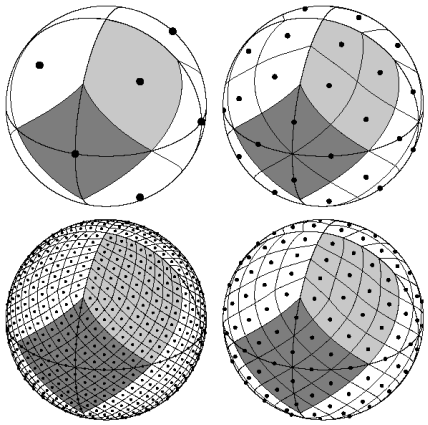

Figure 3: Orthographic view of HEALPix partition of the sphere.

2.2 Rank-1 POVMs implementation

We have already discussed how every qubit POVM can be written as a coarse graining of rank-1 projectors [Bar02], such that the protocol implementation can restrict without any loss in generality to POVM elements proportional to rank-1 projectors.

Even if the final goal is to build POVMs to test how the classical protocol converges with the quantum theory for any random state and measurement, we will also discuss some general rank-1 POVMs with interesting properties, for example,

1. The measure needed in the eavesdropping of the BB84 protocol [NC00]

$$\mathbb{P}_4 = \{\frac{1}{2}|0\rangle\langle 0|, \frac{1}{2}|1\rangle\langle 1|, \frac{1}{2}|+\rangle\langle +|, \frac{1}{2}|-\rangle\langle -|\} \tag{19}$$

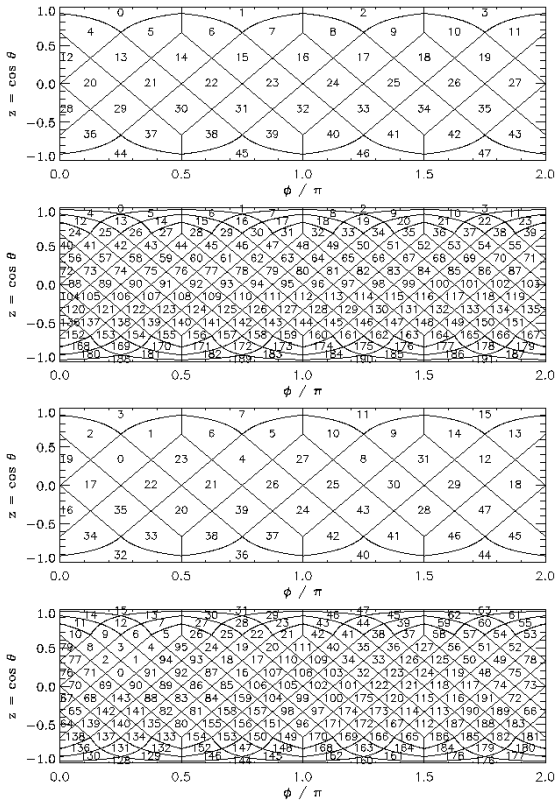

Figure 4: Cylindrical projection of the HEALPix division of a sphere and two natural pixel numbering schemes (RING and NESTED). Both numbering schemes map the two dimensional distribution of discrete area elements on a sphere into the one dimensional, integer pixel number array.

2. The Trine-POVM, consisting of POVM elements uniformly distributed on an equatorial plane of the Bloch sphere, with $\mathbb{P}_3 = \{E_1, E_2, E_3\}$ and $E_k = \frac{2}{3} |\Psi_k\rangle \langle\Psi_k|$, where

$$\begin{aligned} |\Psi_1\rangle &= |0\rangle \\ |\Psi_2\rangle &= \frac{1}{2} |0\rangle + \frac{\sqrt{3}}{2} |1\rangle \\ |\Psi_3\rangle &= \frac{1}{2} |0\rangle - \frac{\sqrt{3}}{2} |1\rangle \end{aligned} \tag{20}$$

3. The SIC-POVMs, a well-known family of symmetric informationally complete positive operator-valued measures, which are proven to be very relevant in quantum state tomography and quantum cryptography fields among others [Ren+04]. The simplest SIC-POVM is the one with states

the vertices of a regular tetrahedron in the Bloch sphere, see Figure 5, with $\mathbb{P}_4 = \{E_1, E_2, E_3, E_4\}$ and $E_k = \frac{1}{2}\,|\Psi_k\rangle\,\langle\Psi_k|$, where

$$\begin{aligned}
|\Psi_1\rangle &= |0\rangle \\
|\Psi_2\rangle &= \frac{1}{\sqrt{3}}\,|0\rangle + \sqrt{\frac{2}{3}}\,|1\rangle \\
|\Psi_3\rangle &= \frac{1}{\sqrt{3}}\,|0\rangle + \sqrt{\frac{2}{3}}\,e^{i\frac{2\pi}{3}}\,|1\rangle \\
|\Psi_4\rangle &= \frac{1}{\sqrt{3}}\,|0\rangle + \sqrt{\frac{2}{3}}\,e^{i\frac{4\pi}{3}}\,|1\rangle
\end{aligned} \tag{21}$$

The strategies to build the rank-1 POVMs and perform the measurement are different in the classical simulation protocol and in the quantum circuit model, as we will see later, so next sections will describe the methodologies applied for each case.

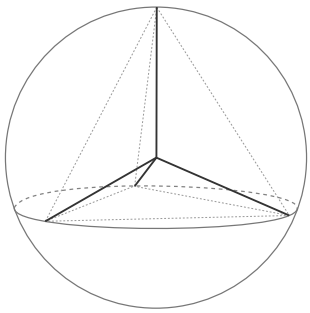

Figure 5: In the Bloch sphere representation of a qubit, the states of a SIC-POVM form a regular tetrahedron with vertices $|\Psi_1\rangle = |0\rangle$, $|\Psi_2\rangle = 1/\sqrt{3}\,|0\rangle + \sqrt{2/3}\,|1\rangle$, $|\Psi_3\rangle = 1/\sqrt{3}\,|0\rangle + \sqrt{2/3}\;e^{i\frac{2\pi}{3}}\;|1\rangle$ and $|\Psi_4\rangle = 1/\sqrt{3}\,|0\rangle + \sqrt{2/3}\;e^{i\frac{4\pi}{3}}\;|1\rangle$.

2.2.1 Measurement in classical simulation protocols

As described by Sentís et al. [Sen+13], the conditions under which a set of N arbitrary rank-1 operators $\{E_k\}$ comprises a qubit POVM such that $\sum_{k=1}^{N} a_k E_k = \mathbb{1}$, can be equivalently written in a system of four linear equations

$$\sum_{k=1}^{N} a_k = 2 \tag{22}$$

$$\sum_{k=1}^{N} a_k \vec{y}_k = \vec{0} \tag{23}$$

where $\vec{y}_k \in \mathbb{R}^3$ are the Bloch vectors corresponding to the qubit pure states $|v_k\rangle$, such that $E_k = |v_k\rangle\,\langle v_k|$. The existence of the set $\{a_k\}$ has a direct

translation into a linear programming feasibility problem we would have to solve computationally.

As an example, to build a random POVM set of $N = 4$ elements, we could apply the following procedure:

1. Assign two rank-1 operators as projective measurement elements $E_i = |v_i\rangle \langle v_i|$ with unknown weights $\{a_i\}$, where $i = 1, 2$.
2. Apply the closure relation such that the third rank-1 operator is $E_3 = \mathbb{1} - \sum_{i=1}^{2} E_i$. Note that this will not be necessarily a rank-1 operator.
3. Diagonalize E_3 to obtain the relevant qubit states as eigenvectors $|v_3\rangle$ and $|v_4\rangle$.
4. Convert all qubit states $|v_i\rangle$ to Bloch vectors $\vec{y}_i$, where $i = 1, 2, ...4$.
5. Solve the linear programming feasibility problem

$$\begin{array}{ll} \text{find} & x = \{a_1, a_2, \ldots, a_N\} \\ \text{subject to} & Ax = b \text{ where column } A_{*k} = (\vec{y}_k, 1), \text{ and } b = (\vec{0}, 2) \\ & x \geq 0 \end{array}$$

Provided the optimization problem is feasible, we obtain the weights $\{a_k\}$ and compute the rank-1 operators $E_k = |v_k\rangle \langle v_k|$ which conform the POVM set elements $\{B_k\}$ such that $B_k = a_k E_k$. Then we can use Equation (12) to perform the following assignment

$$p_k = \frac{a_k}{2} \tag{24}$$

$$|\vec{y}_k\rangle \langle \vec{y}_k| = E_k \tag{25}$$

which will implement the POVMs in the form required by the classical simulation protocols, i.e. $B_k = 2p_k |\vec{y}_k\rangle \langle \vec{y}_k|$.

2.2.2 Measurement in quantum circuit model

To compare the probability distributions obtained from classical protocols with those from quantum simulators or noisy quantum computers, we must develop a technique for encoding positive operator-valued measures in a quantum circuit model. For a POVM of N elements, such technique requires to create a $N \times N$ unitary matrix U representing the measurement process.

Neumark's theorem [Neu40] asserts that one can extend the Hilbert space of states $\mathcal{H}$ in which rank-1 POVM elements

$$B_k = |v_k\rangle \langle v_k|, \text{ where } \sum_{k=1}^{N} B_k = \mathbb{1} \tag{26}$$

are defined, in such a way that there exists in the extended space $\mathcal{K}$ a set of orthogonal projectors Π_k such that B_k is the result of projecting Π_k from $\mathcal{K}$ into $\mathcal{H}$. Following Peres [Per95], we can add $N-2$ extra dimensions to $\mathcal{H}$ by introducing unit vectors $|u_k\rangle$ orthogonal to each other and to all $|v_k\rangle$ in Equation (26). Then we can build a complete orthonormal basis $|w_k\rangle$ in the enlarged space $\mathcal{K}$ such that

$$|w_k\rangle := |v_k\rangle + \sum_{s=3}^{N} c_{ks} |u_k\rangle \tag{27}$$

$$\langle w_j|w_k\rangle := \langle v_j|v_k\rangle + \sum_{s=3}^{N} c_{js}^{\star} c_{ks} = \delta_{jk} \tag{28}$$

where c_{ks} are the complex coefficients to be determined. Eqs. (26) and (28) can be rewritten in index notation as

$$\sum_{k=1}^{N} v_{ki}^{\star} v_{kj} = \delta_{ij} \tag{29}$$

$$\sum_{i=1}^{2} v_{ji}^{\star} v_{ki} + \sum_{s=3}^{N} c_{js}^{\star} c_{ks} = \delta_{jk} \tag{30}$$

According to Equation (30) the following matrix U is a unitary matrix which satisfies the closure property in Equation (29) and encapsulates orthonormal states in the enlarged space $\mathcal{K}$

$$U = \begin{pmatrix} v_{11} & v_{12} & c_{13} & \dots & c_{1N} \\ v_{21} & v_{22} & c_{23} & \dots & c_{2N} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ v_{N1} & v_{N2} & c_{N3} & \dots & c_{NN} \end{pmatrix} \tag{31}$$

By computing the complex coefficients c_{ks}, we can then encode any rank-1 POVM measure into a unitary matrix U which can be readily used within a quantum circuit model.

2.3 Prepare-and-measure simulations

In the following sections we will discuss the methods applied to implement the prepare-and-measure classical simulation protocol described in Section 1.5.1.

2.3.1 Classical transmission of one qubit

So far we have described the methods available to generate random qubit states and random measurements proportional to rank-1 projectors. Once these are available, we should convert them to the corresponding elements in the Bloch sphere, i.e. to vectors $\vec{x}$, $\vec{y}_k \in \mathbb{R}^3$, as per protocol description (see Section 1.5.1).

A qubit state in density matrix form ρ can be easily be transformed into the dual Bloch vector $\vec{x}$ with components $\{x_k\}$ by applying the equation

$$x_k = tr(\rho \cdot \sigma_k), \text{ where } \vec{\sigma} = (\sigma_x, \sigma_y, \sigma_z) \tag{32}$$

Similarly, the vector $\vec{y}_k$ associated to the POVM will be the Bloch vector corresponding to the rank-1 projector $|\vec{y}_k\rangle \langle\vec{y}_k|$, so the same procedure could be flawlessly applied.

For every random state and POVM, we will then sample the shared randomness $\vec{\lambda}_1$, $\vec{\lambda}_2 \in \mathbb{R}^3$ following a uniform distribution and will apply steps 1 to 4 in the protocol such that for every run we get the probability for each measurement outcome. Equation (4) can then be computed by just using the probabilities outcomes as weights in a random choice whose outcome gets accumulated for each shared randomness run. The accumulated random choices will lead to the final probabilities which will be then compared against the ones obtained with either the theoretical probabilities as per Born's rule, or the probabilities obtained executing the associated quantum circuit in a quantum simulator (see Section 2.3.2).

2.3.2 Quantum circuit counterpart

Following Neumark's theorem described in Section 2.2.2, we can now implement any POVM measure of N elements in a quantum circuit by applying the $N \times N$ unitary matrix U resulting from Neumark's theorem[1] to an initial state made of the original qubit state we want to measure $|\Psi\rangle$, together with a set of $n-1$ ancillary qubits in a zero state $|0\rangle$ such that $2^n \geq N$ (see Figure 6).

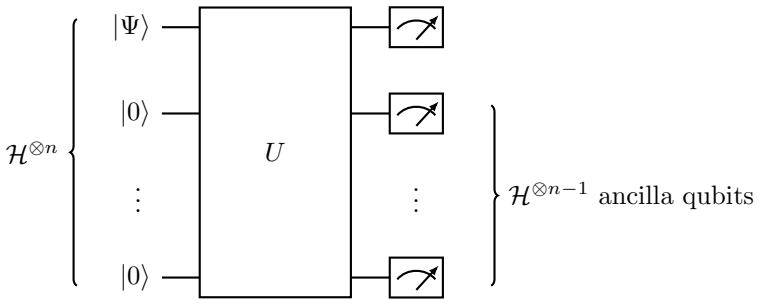

Figure 6: Quantum circuit implementing a POVM measure of N outcomes following Neumark's extension theorem.

As we can see in the quantum circuit of Figure 6, we will obtain the different probabilities for each of the N possible outcomes of the POVM, by adding n

[1] In this project we have just focused on unitary matrices of dimension $N \times N$ where $log_2 N$ is a positive integer greater than one. For a complete treatment of the subject, the procedure mentioned in Section 2.2.2, based on [Per95], could be complemented with the one described in [Joz+03].

classical registers to the circuit, which will then perform classical measures on the circuit's computational basis. Each outcome from a total set of 2^n possible outcomes of the classical registers will therefore correspond to a POVM element measurement outcome. As an example, the measurement of a qubit state $|\Psi\rangle$ with a 4-element POVM as in Figure 5, will be encoded in a quantum circuit as a 4×4 unitary matrix applied to the qubit state $|\Psi\rangle$ plus an ancillary qubit $|0\rangle$ such that all possible outcomes from the classical 2-bit register $\{00, 01, 10, 11\}$, will correspond to the measurement outcomes for each POVM element ($N = 4$).

As we will show in Section 3, these circuits will be implemented with Qiskit [Con23], IBM's quantum computing SDK, and will be run using IBM Quantum processors to obtain the experimental probability distributions to be compared with the results from the classical simulations.

In order to translate the POVM's unitary matrices into universal quantum gates available in the underlying IBM Quantum processors, we could either follow Nielsen and Chuang's textbook [NC00], and decompose the unitary into a sequence of two-level unitary gates, or rely on the Qiskit's transpiler, which translates any generic circuit into an optimized circuit using a backend's native gate set, allowing users to program for any quantum processor or processor architecture with minimal inputs. For the sake of simplicity we will follow on using Qiskit's transpiler.

2.4 Bell simulations

Most of the methods required to implement the Bell's classical simulation protocol have already been addressed in the previous section. Hence, the underlying methodology for the generation of random states and measurements, dual Bloch vectors and shared randomness for the Bell scenario will be used without further discussion.

2.4.1 Bell singlet state with projective measurements

The classical protocol described in Section 1.5.2 is applicable to Bell singlet states only. Alice is restricted to projective measurements with outcomes $a = \pm 1$, and Bob can perform arbitrary POVMs, but we will restrict ourselves further with Bob performing arbitrary PVMs only, as per Toner and Bacon's original protocol [TB03].

The expected joint probabilities will be computed for every possible combination of observables A_x, B_y, and outcomes $a_x, b_y = \pm 1$. For every Bell singlet state and set of observables, we will then sample the shared randomness $\vec{\lambda}_1$, $\vec{\lambda}_2 \in \mathbb{R}^3$ following a uniform distribution and will apply steps 1 to 4 in the protocol such that for every run we get the probability for each measurement outcome. Equation (10) can then be computed by just using the probabilities outcomes as weights in a random choice whose outcome gets accumulated for each shared randomness run. The accumulated random choices will lead to the

final probabilities which will be then compared against the ones obtained with Equation (7).

2.4.2 CHSH inequality

In addition to the computation of joint probabilities, the expectation values for every duple of observables $\mathbb{E}[A_x, B_y]$ will also be calculated according to Equation (8). That would allow us to use Equation (9) and try to prove through the classical protocol the breaking of Bell's inequality under a suitable set of observables by arguing that the CHSH absolute value exceeds the classical upper bound that was deduced from the hypothesis of local hidden variable model, i.e.

$$|CHSH| \leq 2 \tag{33}$$

3 Results[2]

3.1 Random state and measurement preparation

At the backbone of all classical protocols described in Section 1.5, it lies the need to generate random uniform distributions of either qubit states, Bloch vectors or shared randomness in $\mathbb{R}^3$. In Figure 7, we can see that applying random unitary transformations to zero qubit states leads to uniformly distributed random qubit pure states, which can later be transformed into random Bloch vectors or normalized random vectors in $\mathbb{R}^3$ as needed. These random states are also the seed to compute random POVMs with rank-1 projectors, as described in Section 2.2.1. Hence, once proven we have means to generate random states and rank-1 POVMs, we can address the measurement process.

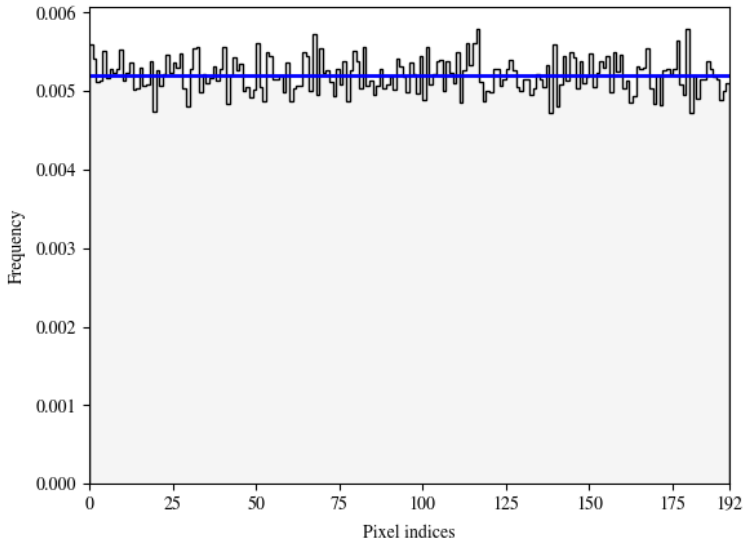

Figure 7: In this histogram we see how the frequency distribution of $N = 10^4$ random qubit states generated with [OC23] follows a random uniform distribution of state vectors along the Bloch sphere (blue solid line). The frequencies are binned by HEALPix indices corresponding to an equal-area iso-latitude partition of the Bloch sphere with resolution $NSIDE = 4$.

[2] All the methods described in Section 2 have been implemented from scratch in a software package, and have been made publicly available as an open-source project on a code repository [OC23]. The main building blocks of the code have also been included in this document for self completeness (see Appendix A). All outcomes presented in this section have been acquired and are reproducible by employing the aforementioned software.

3.2 Prepare-and-measure simulation outcomes

The outcome probabilities for a given random POVM measure can be obtained analytically using the Born's rule, but we have gone a step further and used simulators and noisy intermediate-scale quantum computers (see Table 1) to calculate such probabilities and compare them to the theoretical ones as described in Section 2.3.2.

Name	Qubits	Quantum Volume	CNOT Error	Readout Error
Nairobi	7	32	0.01357	0.0227
Perth	7	32	0.01733	0.0188
Oslo	7	32	0.01	0.01667
Jakarta	7	16	0.00773	0.0258
Lagos	7	32	0.007243	0.0145

Table 1: Properties of Noisy Intermediate-Scale Quantum computers available at IBM Quantum to compute the prepare-and-measure outcome probabilities.

The quantum circuit resulting from applying Neumark's extension to a particular state and POVM measure is shown in Figure 8. A summary of the achievable probability performances for such circuit using different quantum resources can be found in Table 2. Noisy quantum computers currently available to the public are limited to run circuits with a maximum of twenty thousand shots. This, together with the lack of error correction measures available, results in less accuracy than desired. On the other hand, the Qiskit Aer simulator can perform noise-free simulations with a full range of shots, hence leading to better performances, as we will see later in the classical simulation results discussion. A full and comprehensive benchmark of IBM's free-access simulators and quantum computers for this prepare-and-measure scenario can be found in Appendix B.

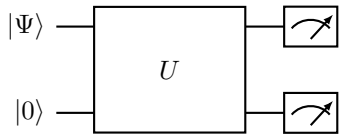

Figure 8: Quantum circuit implementing a POVM measure of four elements. The unitary gate U resulting from Neumark's is applied to the prepared state together with a single ancilla qubit such that the classical measurement leads to four possible outcomes: $\{00, 01, 10, 11\}$.

Beyond all this, a set of simulations have been conducted for the prepare-and-measure classical protocol using a random assortment of states and measurements, in addition to a collection of well-known measures like the Cross-POVM (19), the Trine-POVM (20) and a four-element SIC-POVM sample (21).

Shots	10^2				10^3				10^4				$2{\cdot}10^4$			
Measure	00	01	10	11	00	01	10	11	00	01	10	11	00	01	10	11
Aer Simulator	0.370	0.150	0.070	0.410	0.362	0.117	0.067	0.454	0.377	0.122	0.062	0.438	0.378	0.120	0.058	0.443
Nairobi	0.400	0.140	0.030	0.430	0.335	0.162	0.112	0.391	0.378	0.172	0.090	0.359	0.345	0.115	0.097	0.443
Perth	0.350	0.260	0.100	0.290	0.337	0.185	0.121	0.357	0.370	0.187	0.118	0.325	0.328	0.152	0.113	0.408
Oslo	0.400	0.080	0.040	0.480	0.379	0.100	0.052	0.469	0.358	0.108	0.070	0.463	0.378	0.114	0.064	0.444
Jakarta	0.350	0.160	0.110	0.380	0.378	0.160	0.076	0.386	0.383	0.143	0.088	0.386	0.366	0.149	0.099	0.387
Lagos	0.380	0.100	0.060	0.460	0.366	0.113	0.060	0.461	0.357	0.113	0.062	0.467	0.366	0.130	0.070	0.434
Born's rule	0.375	0.125	0.064	0.435	0.375	0.125	0.064	0.435	0.375	0.125	0.064	0.435	0.375	0.125	0.064	0.435

Table 2: Example of probability distributions for circuit's classical measurement outcomes $\{00, 01, 10, 11\}$ obtained with different IBM Quantum simulators and quantum computers. The prepare-and-measure scenario used is that with state $|\Psi\rangle = \frac{3+i\sqrt{3}}{4}|0\rangle - \frac{1}{2}|1\rangle$ and POVM measure $\mathbb{P}_4 = \{\frac{1}{2}|0\rangle\langle 0|, \frac{1}{2}|1\rangle\langle 1|, \frac{1}{2}|+\rangle\langle +|, \frac{1}{2}|-\rangle\langle -|\}$.

The results shown in Table 3 clearly indicate that the correlations obtained by the classical prepare-and-measure simulations reproduce the quantum probabilities with extraordinary accuracy. The relative entropy distances among the theoretical probability distribution P, and the classical simulation probability distribution Q, on the sample space $\mathcal{X}$, have also been computed following the Kullback-Leibler divergence formula [Mac03]

$$D_{KL}(P||Q) = \sum_{x\in\mathcal{X}} P(x)\ log\frac{P(x)}{Q(x)} \tag{34}$$

Figure 9 shows the Kullback-Leibler divergence among Born's rule and the classical simulation probabilities obtained for the set of prepare-and-measure scenarios in Table 3.

Scenario	Probabilities			
Cross-POVM[1]	$0.3749^{0.3750}$	$0.1250^{0.1250}$	$0.0625^{0.0625}$	$0.4376^{0.4375}$
Trine-POVM[2]	$0.4998^{0.5000}$	$0.0335^{0.0335}$	$0.4667^{0.4665}$	-
SIC-POVM[3]	$0.3751^{0.3750}$	$0.0315^{0.0316}$	$0.3851^{0.3851}$	$0.2082^{0.2083}$
Random-PVM[4]	$0.9669^{0.9669}$	$0.0331^{0.0331}$	-	-
Random-POVM[5]	$0.0097^{0.0097}$	$0.0057^{0.0057}$	$0.8825^{0.8825}$	$0.1021^{0.1021}$
Random-POVM[6]	$0.2386^{0.2389}$	$0.1242^{0.1242}$	$0.6341^{0.6337}$	$0.0031^{0.0031}$

Table 3: Probability outcomes of several prepare-and-measure classical simulations after 10^7 shots, and using a diverse set of measurements and prepared states $|\Psi\rangle$. For the purpose of comparison, theoretical probabilities calculated using Born's rule are presented in superscript blue color.
[1,2,3] $|\Psi\rangle = \frac{3+i\sqrt{3}}{4}|0\rangle - \frac{1}{2}|1\rangle$
[4,5] $|\Psi\rangle = -(0.606 + 0.642i)|0\rangle - (0.336 + 0.327i)|1\rangle$
[6] $|\Psi\rangle = (-0.461 + 0.195i)|0\rangle - (0.767 - 0.402i)|1\rangle$

Taking advantage of the analysis performed comparing theoretical and simulated probability distributions, we have extended the Kullback-Leibler divergence analysis to the prepare-and-measure scenario using IBM Quantum simulators. Figure 10 shows that existing quantum simulators converge to the expected theoretical values slightly slower than the classical simulations. The outcome probabilities after 10^7 simulation runs are available in Table 4.

3.3 Bell simulation outcomes

Following the protocol described in Section 1.5.2, and the corresponding methodology in Section 2.4, a complementary set of simulations have also been carried out for a Bell scenario with a singlet state and local projective measurements.

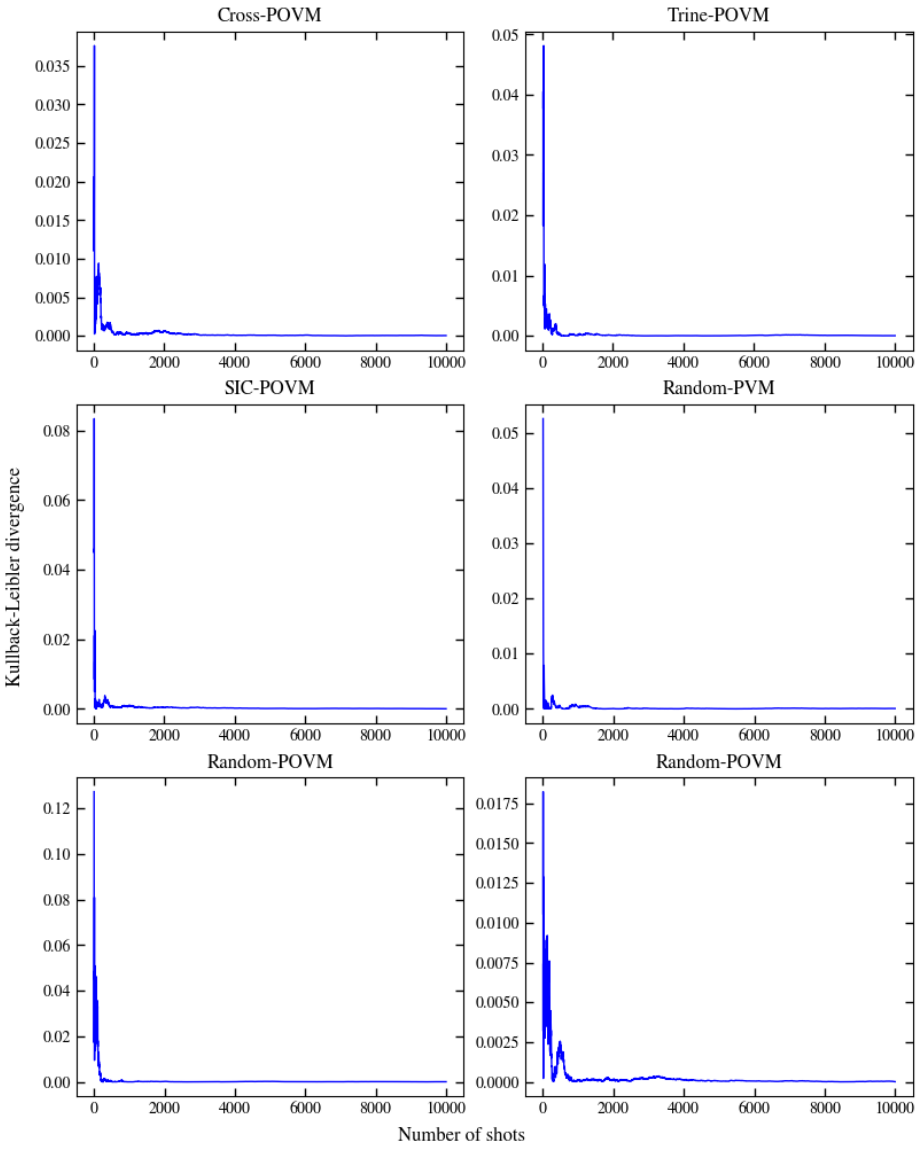

Figure 9: The Kullback-Leibler relative entropy among the theoretical probabilities and the simulation probability distributions for the prepare-and-measure scenarios specified in Table 3. A total of 10^4 shots were used for the simulations, and it can be clearly seen that the results converge to zero statistical distance rather quickly.

Similarly to the prepare-and-measure scenario case, the results in Table 5 show that the correlations obtained by the classical Bell simulations reproduce the quantum joint probabilities and expectation values with remarkable accuracy. In Table 5 it is straightforward to see that, in the classical simulation performed,

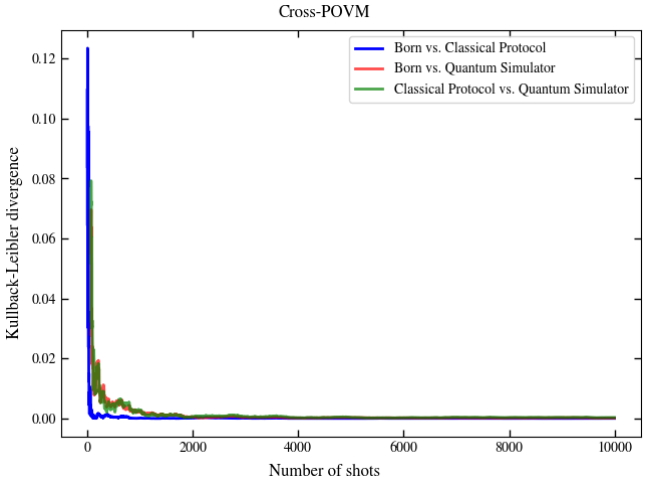

Figure 10: The Kullback-Leibler relative entropy among the theoretical probabilities and the simulation probability distributions using the classical protocol and the IBM Quantum simulator. The prepared state selected is $|\Psi\rangle = 3/4 + i\sqrt{3}/4\,|0\rangle - 1/2\,|1\rangle$, and the POVM measure the four-element Cross-POVM (19). A total of 10^4 shots were used for the simulations, where it can be noticed some differences in the convergence, particularly during the first thousands of runs.

Method	Probabilities			
Classical Simulation	0.3749	0.1250	0.0625	0.4376
Quantum Simulation	0.3751	0.1249	0.0625	0.4374
Born's rule	0.3750	0.1250	0.0625	0.4375

Table 4: Probability outcomes of prepare-and-measure classical and quantum simulations after 10^7 shots. The prepared state selected is $|\Psi\rangle = 3/4 + i\sqrt{3}/4\,|0\rangle - 1/2\,|1\rangle$, and the POVM measure the four-element Cross-POVM (19). The theoretical probabilities obtained applying Born's rule are also provided for ease of comparison.

the CHSH inequality is violated in the chosen set of Alice and Bob observables, and it is therefore consistent with quantum theory predictions.

The convergence of joint probabilities to theoretical values is illustrated more visually in Figure 11, which demonstrates a rapid convergence as the number of shots increases. On the other hand, Figure 12 exhibits the accuracy of this convergence to the theoretical values.

(a_x, b_y)	$p_C(a_x, b_y \vert A_x, B_y)$				CHSH
	(A_0, B_0)	(A_0, B_1)	(A_1, B_0)	(A_1, B_1)	
$(+1, +1)$	$0.4268^{0.4267}$	$0.4269^{0.4267}$	$0.4267^{0.4267}$	$0.0734^{0.0732}$	-
$(+1, -1)$	$0.0731^{0.0732}$	$0.0732^{0.0732}$	$0.0732^{0.0732}$	$0.4265^{0.4267}$	-
$(-1, +1)$	$0.0732^{0.0732}$	$0.0733^{0.0732}$	$0.0732^{0.0732}$	$0.4270^{0.4267}$	-
$(-1, -1)$	$0.4268^{0.4267}$	$0.4267^{0.4267}$	$0.4269^{0.4267}$	$0.0731^{0.0732}$	-
$\mathbb{E}[A_x, B_y]$	0.7072	0.7073	0.7072	-0.7070	2.8287

Table 5: Probability outcomes of a classical Bell simulation after 10^7 shots. The entanglement state is the Bell singlet state $\left|\Psi^-\right\rangle = (|00\rangle - |11\rangle)/\sqrt{2}$, and the observables chosen are $A_0 = Z$, $A_1 = X$, $B_0 = -(X + Z)/\sqrt{2}$, $B_1 = (X - Z)1/\sqrt{2}$. For the purpose of comparison, theoretical probabilities calculated using Born's rule are presented in superscript blue color.

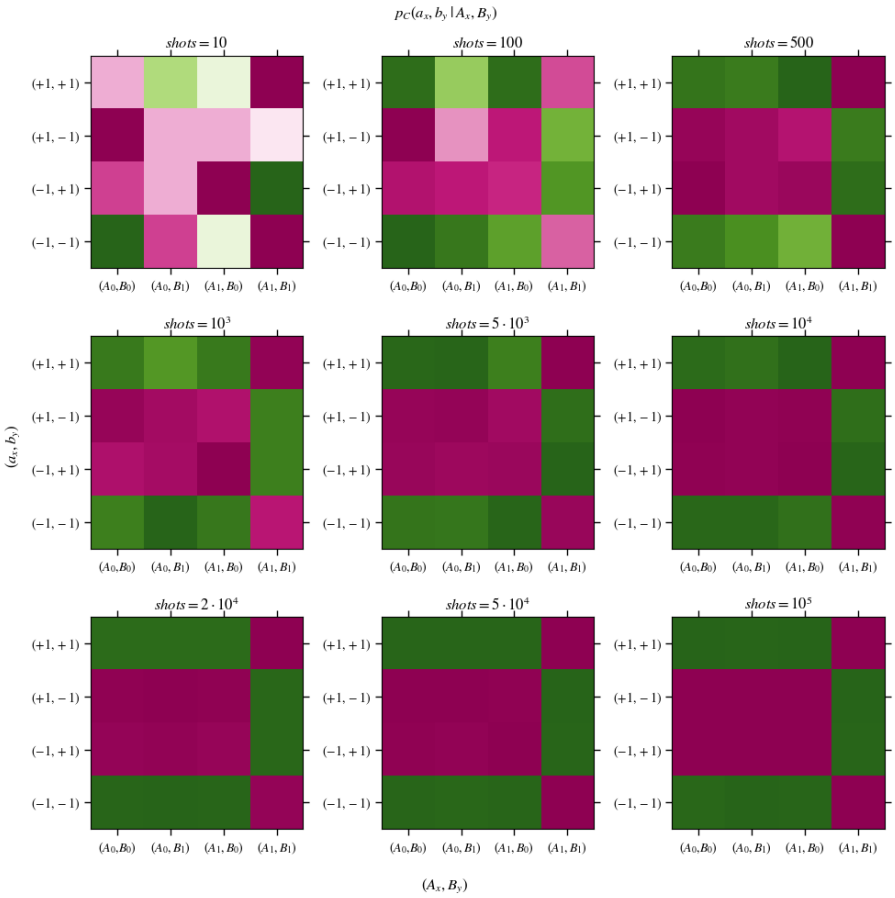

Figure 11: The joint probabilities for the Bell classical simulations are plotted in a heatmap for various numbers of shots, revealing that the outcomes begin to converge after a few thousand shots.

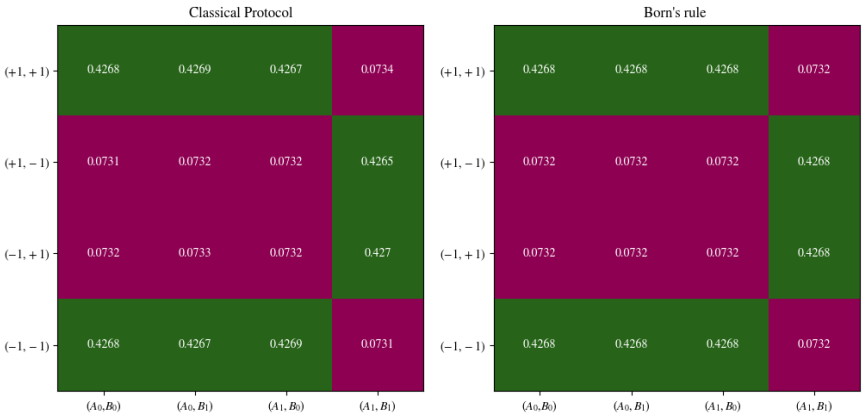

Figure 12: The heatmap displays the joint probabilities obtained from a classical Bell simulation after 10^7 shots, alongside the theoretical values computed via Born's rule, revealing a significant level of accuracy.

4 Conclusion

This project has focused on exploring the fundamental limits of quantum over classical information theories in scenarios related to prepare-and-measure and Bell experiments. Specifically, we have reproduced the quantum correlations and probability distributions produced by local measurements of a quantum system, which are the basic resource of quantum information theory. We have shown that any prediction based on projective measurements on any qubit state could be simulated classically by communicating only two classical bits, and we have extended this result to a most generalised set of measurements, the positive operator-valued measures. Furthermore, we have demonstrated through computer-based experiments that the predicted probabilities can also be reproduced, to a certain degree of precision, by existing quantum simulators and noisy intermediate-scale quantum computers. Our simulations have also proven to replicate non-locality in scenarios featuring entangled states, leading to well-known results like the CHSH inequality breaking for maximally entangled states with a well chosen set of observables.

Overall, this project has provided practical hands-on experience on fundamental quantum information concepts and and facilitated the acquisition of a functional understanding in the formulation of quantum communication protocols, thereby offering valuable insights for any future work related to quantum technologies in our professional careers.

The results of the current project provide a solid foundation for further exploration of quantum information theory. One avenue for future work could be extending the classical simulations to higher dimensional quantum prepare-and-measure scenarios, e.g. preparing qutrit states ($d_Q = 3$), given the current project was restricted to qubits ($d_Q = 2$). Another potential direction could be adapting the Bell scenario protocol to other states beyond the singlet state, as this could provide insight into the power of different types of entanglement. We could also extend this work by running classical Bell simulations where Bob can perform arbitrary POVMs. Additionally, increasing the quantum computer resources, limited by the number of shots available, and applying quantum error correction techniques to the generalized measurement experiments with noisy intermediate-scale quantum computers could improve the accuracy and reliability of the results. These avenues of research would deepen our understanding of quantum information theory and could have practical applications in the development of more advanced quantum communication protocols.

5 Acknowledgements

We would like to express our sincere gratitude to our supervisor, Gael Sentís Herrera, for his guidance, support, and encouragement throughout this project. We are also grateful to the faculty and staff of the Quantum Engineering Postgraduate degree program at Universitat Politècnica de Catalunya (UPC), Universitat Autònoma de Barcelona (UAB) and Institut de Ciències Fotòniques (ICFO), for the excellent education they have provided throughout this degree. Thank you all for your assistance, motivation, and direction, without which this project would not have been possible.

References

[Neu40] M. A. Neumark. In: *Izv. Akad. Nauk SSSR, Ser. Mat.* 4 (1940).

[Sha48] Claude Elwood Shannon. "A Mathematical Theory of Communication". In: *The Bell System Technical Journal* 27 (1948), pp. 379–423. URL: http://plan9.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf (visited on 04/22/2003).

[Bel64] J. S. Bell. "On the Einstein Podolsky Rosen paradox". In: *Physics Physique Fizika* 1 (3 Nov. 1964), pp. 195–200. DOI: 10.1103/PhysicsPhysiqueFizika.1.195. URL: https://link.aps.org/doi/10.1103/PhysicsPhysiqueFizika.1.195.

[ZK94] K Zyczkowski and M Kus. "Random unitary matrices". In: *Journal of Physics A: Mathematical and General* 27.12 (June 1994), p. 4235. DOI: 10.1088/0305-4470/27/12/028. URL: https://dx.doi.org/10.1088/0305-4470/27/12/028.

[Per95] A. Peres. *Quantum Theory: Concepts and Methods*. Fundamental Theories of Physics. Springer Dordrecht, 1995. ISBN: 978-0-7923-3632-7. URL: https://link.springer.com/book/10.1007/0-306-47120-5.

[CGM00] N. J. Cerf, N. Gisin, and S. Massar. "Classical Teleportation of a Quantum Bit". In: *Physical Review Letters* 84.11 (Mar. 2000), pp. 2521–2524. DOI: 10.1103/physrevlett.84.2521. URL: https://doi.org/10.1103/physrevlett.84.2521.

[NC00] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[Bar02] Jonathan Barrett. "Nonsequential positive-operator-valued measurements on entangled mixed states do not always violate a Bell inequality". In: *Physical Review A* 65.4 (Mar. 2002). DOI: 10.1103/physreva.65.042302. URL: https://doi.org/10.1103/physreva.65.042302.

[Joz+03] Richard Jozsa et al. "Entanglement cost of generalised measurements". In: (2003). DOI: 10.48550/ARXIV.QUANT-PH/0303167. URL: https://arxiv.org/abs/quant-ph/0303167.

[Mac03] David J. C. Mackay. *Information Theory, Inference and Learning Algorithms*. 2003.

[TB03] B. F. Toner and D. Bacon. "Communication Cost of Simulating Bell Correlations". In: *Physical Review Letters* 91.18 (Oct. 2003). DOI: 10.1103/physrevlett.91.187904. URL: https://doi.org/10.1103/physrevlett.91.187904.

[Ren+04] Joseph M. Renes et al. "Symmetric informationally complete quantum measurements". In: *Journal of Mathematical Physics* 45.6 (June 2004), pp. 2171–2180. DOI: 10.1063/1.1737053. URL: https://doi.org/10.1063/1.1737053.

[Gór+05] K. M. Górski et al. “HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere”. In: *apj* 622 (Apr. 2005), pp. 759–771. DOI: 10.1086/427976. eprint: arXiv:astro-ph/0409513.

[Ozo09] Maris A. Ozols. “How to generate a random unitary matrix”. In: 2009. URL: http://home.lu.lv/~sd20008/papers/essays/Random%20unitary%20[paper].pdf.

[Sen+13] G Sentis et al. “Decomposition of any quantum measurement into extremals”. In: *Journal of Physics A: Mathematical and Theoretical* 46.37 (Aug. 2013), p. 375302. DOI: 10.1088/1751-8113/46/37/375302. URL: https://doi.org/10.1088/1751-8113/46/37/375302.

[Con23] Qiskit Contributors. *Qiskit: An Open-source Framework for Quantum Computing*. 2023. DOI: 10.5281/zenodo.2573505.

[OC23] Inaki Ortiz de Landaluce and Aido Cortes Alcaraz. *Simulations of qubit communication in prepare-and-measure and Bell scenarios*. Version 1.0. May 2023. URL: https://github.com/inaki-ortizdelandaluce/qubit-communication-simulations/.

[RTQ23] Martin J. Renner, Armin Tavakoli, and Marco Túlio Quintino. “Classical Cost of Transmitting a Qubit”. In: *Phys. Rev. Lett.* 130 (12 Mar. 2023), p. 120801. DOI: 10.1103/PhysRevLett.130.120801. URL: https://link.aps.org/doi/10.1103/PhysRevLett.130.120801.

Appendices

Appendix A Source Code Listings

A.1 qubit.py

This module provides basic operations for qubit pure states either in two-dimensional the Hilbert space $\mathcal{H}$ or in the Bloch sphere S_2.

```
import numpy as np
import cmath
import math
from enum import Enum


class Axis(Enum):
    X = 0
    Y = 1
    Z = 2


X = np.array([[0, 1], [1, 0]])
Y = np.array([[0, -1.j], [1j, 0]])
Z = np.array([[1, 0], [0, -1]])
paulis = np.array([X, Y, Z])


class Qubit:
    def __init__(self, ket=np.array([1, 0])):
        """
        Initializes a qubit in the computational basis. If no arguments are
↪   provided, it returns the zero state.

        Parameters
        ----------
        ket : ndarray
            The qubit components in the computational basis in a 1-d complex
↪   array.
        """
        self.alpha = complex(ket[0])
        self.beta = complex(ket[1])
        self.normalize()

    def __repr__(self):
        return '{} |0> + {} |1>'.format(self.alpha, self.beta)
```

```
    def ket(self):
        return np.array([self.alpha, self.beta], dtype=np.complex_)

    def normalize(self):
        arr = self.ket()
        self.alpha, self.beta = arr/np.linalg.norm(arr)

    def rho(self):
        """
        Returns the density matrix corresponding to the qubit in a pure state.

        Returns
        -------
        ndarray
            A 2x2 density matrix corresponding to the qubit in a pure state.
        """
        return np.outer(self.ket(), self.ket().conj())

    def bloch_angles(self):
        """
        Returns the spherical coordinates of the qubit in the Bloch sphere, with
↪   polar and azimuthal angles in radians.

        Returns
        -------
        (float, float)
            The Bloch sphere coordinates, first the polar angle and then the
↪   azimuthal angle (both in radians).
        """
        r0, phi0 = cmath.polar(self.alpha)
        r1, phi1 = cmath.polar(self.beta)
        theta = 2 * math.acos(r0)
        phi = phi1 - phi0

        return theta, phi

    @staticmethod
    def density2bloch(rho):
        """
        Returns the cartesian coordinates of the specified qubit state in the
↪   Bloch sphere.

        Parameters
        ----------
        rho : ndarray
            The qubit state in density matrix form

        Returns
```

```
        -------
        (float, float, float)
            The cartesian coordinates of the qubit in the Bloch sphere (xyz).

        """
        # cast complex to real to avoid throwing ComplexWarning, imaginary part
        ↪  should always be zero
        return [np.real(np.trace(np.matmul(rho, sigma))) for sigma in paulis]

    def bloch_vector(self):
        """
        Returns the cartesian coordinates of the qubit in the Bloch sphere.

        Returns
        -------
        (float, float, float)
            The cartesian coordinates of the qubit in the Bloch sphere (xyz).

         """
        return Qubit.density2bloch(self.rho())

    def rotate(self, axis: Axis, angle):
        """
        Rotates the qubit along the specified axis by the specified angle in
↪  radians in counterclockwise direction.

        Parameters
        ----------
        axis : Axis

        angle : float
            Angle in radians
        """
        r = math.cos(angle/2) * np.eye(2, 2) - 1.j * math.sin(angle/2) *
        ↪  paulis[axis.value]
        self.alpha, self.beta = r @ self.ket()
        return None
```

A.2 measurement.py

This module implements projection-valued measures and rank-1 positive operator-valued measures for qubit states.

```
import numpy as np
import scipy
from qt.qubit import Qubit


class PVM:
    def __init__(self, proj):
        """
        Creates a PVM with the specified rank-1 projectors.

        Parameters
        ---------
        proj : ndarray
            A 3-d array with the constituting rank-1 projectors.
        """
        # check input
        if not np.allclose(np.identity(2), np.sum(proj, axis=0)):
            raise ValueError('PVM projectors do not sum up the identity')

        self.proj = proj
        self.bloch = np.asarray([Qubit.density2bloch(p) for p in proj])

    @classmethod
    def new(cls, qubit: Qubit):
        """
        Creates a PVM with the rank-1 projectors corresponding to the specified
↪   qubit state.

        Parameters
        ---------
        qubit : Qubit
            The specified qubit state from which the two rank-1 projectors are
↪   generated.
        """
        rho = qubit.rho()
        sigma = np.identity(2) - rho
        proj = np.array([rho, sigma])
        return cls(proj)

    def projector(self, index):
        """
        Returns rank-1 projector for the corresponding index
```

```
        Parameters
        ---------
        index : the projector index

        Returns
        -------
        ndarray
            A 2-d array with the corresponding projector.
        """
        return self.proj[index]

    def probability(self, qubit: Qubit):
        """
        Returns the probabilities of the different outcomes for a given qubit
↪   state

        Parameters
        ---------
        qubit : the qubit state

        Returns
        -------
        ndarray
            The probabilities for each outcome given the input state, stored in a
↪   1-d array.
        """

        # repeat density matrix along zero axis
        rho = qubit.rho()
        rho = np.repeat(rho[np.newaxis, :, :], self.proj.shape[0], axis=0)

        # compute trace of projectors by density matrix
        return np.real(np.trace(np.matmul(self.proj, rho), axis1=1, axis2=2))


class POVM:

    def __init__(self, weights, proj):
        """
        Creates a POVM with the specified weights and rank-1 projectors.

        Parameters
        ---------
        weights : ndarray
            The positive coefficients of the constituting rank-1 projectors.

        proj : ndarray
            A 3-d array with the constituting rank-1 projectors.
```

```
        """
        self.weights = weights
        self.elements = proj * weights[:, np.newaxis, np.newaxis]

        # check input
        if not np.allclose(np.identity(2), np.sum(self.elements, axis=0)):
            raise ValueError('POVM elements do not sum up the identity')

        positive = [np.all(np.linalg.eig(element)[0] >=
        ↪  -np.finfo(np.float32).eps) for element in self.elements]
        if not np.all(positive):
            raise ValueError('Some POVM elements are not definite positive')

        self.bloch = v = np.asarray([Qubit.density2bloch(p) for p in proj])

    @classmethod
    def new(cls, qubits):
        """
        Creates a POVM with the rank-1 projectors corresponding to the specified
↪   qubit states.

        Parameters
        ---------
        qubits : ndarray
           The specified array of N-2 qubit states from which the N rank-1 POVM
↪   projectors are generated.
        """
        # last element normalizes all POVM elements
        rhos = np.asarray([q.rho() for q in qubits])
        e = np.identity(2) - np.sum(rhos, axis=0)

        # diagonalize last element to obtain remaining rank-1 projectors
        _, w = np.linalg.eig(e)
        q1 = Qubit(w[:, 0])
        q2 = Qubit(w[:, 1])
        qubits = np.append(qubits, np.array([q1, q2]), axis=0)

        # compute POVM weights and elements as a linear program (see Sentís et al.
        ↪  2013)
        v = np.asarray([q.bloch_vector() for q in qubits])
        n = len(qubits)

        a = np.vstack((np.ones((n,)), v.T))
        b = np.append(np.array([2]), np.zeros(3, ), axis=0)

        # c = np.zeros(n, ) finds a solution instead of minimizing a function
        # lower bounds set to 0.01, this could be fine-tuned
```

```
        lp = scipy.optimize.linprog(np.zeros(n, ), A_eq=a, b_eq=b, bounds=(0.01,
        ↪  1), method='highs')
        _a, _e = lp['x'], np.asarray([q.rho() for q in qubits])

        return cls(_a, _e)

    def element(self, index):
        """
        Returns the POVM element for the corresponding index

        Parameters
        ---------
        index : the POVM element index

        Returns
        -------
        ndarray
            A 2-d array with the corresponding POVM element.
        """
        return self.element[index]

    def probability(self, qubit: Qubit):
        """
        Returns the probabilities of the different outcomes for a given qubit
↪   state

        Parameters
        ---------
        qubit : the qubit state

        Returns
        -------
        ndarray
            The probabilities for each outcome given the input state, stored in a
↪   1-d array.
        """

        # repeat density matrix along zero axis
        rho = qubit.rho()
        rho = np.repeat(rho[np.newaxis, :, :], self.elements.shape[0], axis=0)

        # compute trace of projectors by density matrix
        return np.real(np.trace(np.matmul(self.elements, rho), axis1=1, axis2=2))

    def size(self):
        """
        Returns the number of POVM elements
```

```
        Returns
        -------
        int
            The number of POVM elements.
        """
        return np.size(self.elements, axis=0)

    def unitary(self):
        """
        Returns the associated unitary matrix in the extended Hilbert space
↪   according to Neumark's theorem

        Returns
        -------
        ndarray
            The nxn unitary matrix where n is the number of POVM elements.

        """
        d = 2
        n = self.size()
        u = np.zeros((n, n), dtype=np.complex_)

        # compute the kets of the rank-1 POVM projectors and assign to first d
        ↪  columns
        # v, _, _ = np.linalg.svd(self.elements, full_matrices=True,
        ↪  compute_uv=True, hermitian=False)
        # u[:, 0:d] = v[:, :, 0] / np.linalg.norm(v[:, :, 0], axis=0)
        w, v = np.linalg.eig(self.elements)
        v = v[np.where(w != 0)]
        u[:, 0:d] = v / np.linalg.norm(v, axis=0)

        # remaining n-d columns should correspond to orthogonal projectors in
        ↪  extended space
        p = np.eye(n, dtype=np.complex_)
        for idx in range(d):
            p -= np.outer(u[:, idx], u[:, idx].conj())

        counter = 0
        for b in np.eye(n, dtype=np.complex_):
            w = np.matmul(p, b)
            if not np.isclose(w, 0.0).all():
                w /= np.linalg.norm(w)
                u[:, counter + d] = w
                p -= np.outer(w, w.conj())
                counter += 1
            if counter == (n - d):
                break
```

```
        if not np.allclose(np.matmul(u, u.conj().T), np.eye(n)):
            raise ValueError('Neumark\'s square matrix is not unitary')

        return u
```

A.3 random.py

This module provides means to create random qubit states and random measurements using object classes available on A.1 and A.2 modules.

```
import math
import numpy as np
from qt.qubit import Qubit
from qt.measurement import PVM, POVM


def bloch_vector():
    """
    Generates a normalised vector uniformly distributed on the Bloch sphere

    Returns
    -------
    ndarray
            A normalised vector on the Bloch sphere

    """
    theta, phi = qubit().bloch_angles()
    return np.array([math.sin(theta) * math.cos(phi),
                     math.sin(theta) * math.sin(phi),
                     math.cos(theta)])


def qubit():
    """
    Generates a random qubit.

    Returns
    -------
    Qubit
        A random Qubit.
    """
    # evolve the zero state with a random unitary matrix
    # same as returning first column of random unitary matrix
    u = unitary((2, 2))
    return Qubit(u[:, 0])


def pvm():
    """
    Generates a random projection value measure for a qubit

    Returns
    -------
```

```
    PVM
            A projection value measure instance.
    """
    q = qubit()
    measurement = PVM.new(q)
    return measurement


def povm(n):
    """
    Generates a random positive operator value measure for a qubit

    Parameters
    ---------
    n : int
        Number of POVM elements. Must be greater than two.

    Returns
    -------
    PVM
            A positive operator value measure instance.
    """
    if n <= 2:
        raise ValueError('Number of POVM elements must be greater thant two')

    qubits = [qubit() for _ in range(n - 2)]
    measurement = POVM.new(qubits)
    return measurement


def unitary(shape):
    """
    Generates a random unitary matrix with the given shape.

    Parameters
    ---------
    shape : int or tuple of ints
        Shape of the unitary matrix.

    Returns
    -------
    ndarray
        Unitary matrix with the given shape.
    """
    # build random complex matrix
    m = np.random.normal(0, 1, shape) + 1.j * np.random.normal(0, 1, shape)

    # apply Gram-Schmidt QR decomposition to orthogonalize the matrix
```

```
    q, *_ = np.linalg.qr(m, mode='complete')

    return q
```

A.4 observable.py

This module provides means to obtain the eigenvalues and eigenvectors of an observable.

```
import numpy as np
import scipy as sp


class Observable:
    def __init__(self, matrix):

        if not sp.linalg.ishermitian(matrix):
            raise ValueError('Input matrix is not hermitian')

        self.matrix = matrix
        self.eigenvalues, self.eigenvectors = np.linalg.eig(matrix)

    def eigen(self):
        return self.eigenvalues, self.eigenvectors

    def eigenvector(self, eigenvalue):
        m = self.eigenvectors[:, np.where(np.isclose(self.eigenvalues,
        ↪  eigenvalue))]
        m = m.reshape(m.size, )
        if m.size == 0:
            return None
        else:
            return m
```

A.5 qudit.py

This module provides means to create a qudit state, and particularly to obtain the qudit corresponding to a bipartite state of two qubits required to compute the probabilities for a Bell scenario as per A.6 module.

```
import numpy as np
from qt.qubit import Qubit


class Qudit:
    def __init__(self, ket):
        """
        Initializes a qudit in the computational basis.

        Parameters
        ----------
        ket : ndarray
            The qudit components in the computational basis in a 1-d complex
↪   array.
        """
        self.ket = ket
        self.normalize()

    def normalize(self):
        self.ket = self.ket/np.linalg.norm(self.ket)

    def rho(self):
        """
        Returns the density matrix corresponding to the qudit in a pure state.

        Returns
        -------
        ndarray
            A nxn density matrix corresponding to the qubit in a pure state.
        """
        return np.outer(self.ket, self.ket.conj())

    @classmethod
    def bipartite(cls, q1: Qubit, q2: Qubit):
        ket = np.tensordot(q1.ket(), q2.ket(), axes=0).reshape(4, )
        return cls(ket)
```

A.6 bell.py

This module provides means to compute the probabilities, expected values and CHSH inequality for any Bell state and the set of Alice and Bob observables.

```
from enum import Enum
import numpy as np
import math
from qt.observable import Observable
from qt.qudit import Qudit
from qt.qudit import Qubit


class BellState(Enum):
    PHI_PLUS = 0
    PHI_MINUS = 1
    PSI_PLUS = 2
    PSI_MINUS = 3


class BellScenario:

    _states = {
        0: Qudit(1 / math.sqrt(2) * np.array([1, 0, 0, 1])),
        1: Qudit(1 / math.sqrt(2) * np.array([1, 0, 0, -1])),
        2: Qudit(1 / math.sqrt(2) * np.array([0, 1, 1, 0])),
        3: Qudit(1 / math.sqrt(2) * np.array([0, 1, -1, 0]))
    }

    def __init__(self, state: BellState, alice: tuple[Observable, Observable],
    ↪  bob: tuple[Observable, Observable]):

        if type(alice) is not tuple:
            raise ValueError('Alice\'s observables is not a valid tuple')
        elif len(alice) != 2:
            raise ValueError('Alice\'s number of observables is not
            ↪  valid:{}'.format(str(len(alice))))

        if type(bob) is not tuple:
            raise ValueError('Bob\'s observables is not a valid tuple')
        elif len(bob) != 2:
            raise ValueError('Bob\'s number of observables is not
            ↪  tuple:{}'.format(str(len(bob))))

        self.state = self._states[state.value]
        self.alice = alice
        self.bob = bob
```

```
def probability(self):
    p = np.zeros((4, 4), dtype=float)
    for u in range(4):
        for v in range(4):
            i, j, m, n = (-1) ** (u >> 1), (-1) ** (u % 2), v >> 1, v % 2

            a = Qubit(self.alice[m].eigenvector(i))
            b = Qubit(self.bob[n].eigenvector(j))
            ab = Qudit.bipartite(a, b)
            p[u, v] = np.real(np.trace(np.matmul(ab.rho(),
            ↪  self.state.rho())))
            # print('p{}{}=({},{})x(A{},B{})={}'.format(u, v, i, j, m, n,
            ↪  p[u, v]))
    return p

def expected_values(self):
    p = self.probability()
    sign = np.array([1, -1, -1, 1])
    return np.sum(p * sign[:, np.newaxis], axis=0)

def chsh(self):
    e = self.expected_values()
    return abs(np.sum(e * np.array([1, 1, 1, -1])))
```

A.7 classical.py

This module implements the classical prepare-and-measure and Bell simulation protocols.

```
import math
import numpy as np
import random
import qt.random
from qt.qubit import Qubit
from qt.measurement import PVM, POVM
from qt.bell import BellState, BellScenario
from qt.observable import Observable


def heaviside(a):
    if isinstance(a, np.ndarray):
        return (a >= 0).astype(int)
    else:
        return int(a >= 0)


def theta(a):
    return a * heaviside(a)


def prepare(lambdas, qubit):
    """
    Alice prepares and sends two bits to Bob

    Parameters
    ---------
    lambdas : ndarray
        Shared randomness as two normalized vectors in a numpy 2-d array

    qubit: Qubit
        A uniformly sampled pure state qubit

    Returns
    -------
    dict
        A dictionary with the shared randomness ('lambdas'), the random qubit
↪   ('qubit')
        and the bits to be communicated to Bob ('bits')
    """

    x = qubit.bloch_vector()
    bits = heaviside(np.matmul(x, lambdas.T))
```

```
    return {
        "lambdas": lambdas,
        "qubit": qubit,
        "bits": bits
    }


def measure_pvm(lambdas, bits, measurement: PVM):
    """
    Bob receives two bits from Alice and performs a random PVM

    Parameters
    ----------
    lambdas : ndarray
        Shared randomness as two normalized vectors in a numpy 2-d array

    bits: ndarray
        Bits communicated by Alice in a numpy 1-d array

    measurement: PVM
        A uniformly sampled PVM

    Returns
    -------
    dict
        A dictionary with the random measurement ('measurement') and
        the probabilities for each measurement outcome ('probabilities')
    """
    # flip shared randomness
    flip = np.where(bits == 0, -1, 1).reshape(2, 1)
    lambdas = np.multiply(lambdas, flip)

    # pick bloch vectors from PVM elements
    y = measurement.bloch
    pb = 0.5 * np.array([1, 1])

    # compute lambda according to the probabilities {pb}
    index = random.choices(range(0, 2), cum_weights=np.cumsum(pb), k=1)[0]
    a = np.abs(np.matmul(lambdas, y.T))
    _lambda = lambdas[np.argmax(a, axis=0)[index]]

    # compute probabilities
    thetas = theta(np.matmul(y, _lambda.reshape(-1, 1)))
    weighted_thetas = np.multiply(thetas, pb.reshape(-1, 1))
    p = weighted_thetas[:, 0] / np.sum(weighted_thetas, axis=0)

    return {
        "measurement": measurement,
```

```
        "probabilities": p
    }


def prepare_and_measure_pvm(shots):
    """
    Runs a prepare-and-measure classical simulation with a random PVM measurement

    Parameters
    ---------
    shots : int
        Number of shots the simulation is run with

    Returns
    -------
    dict
        A dictionary with the random state ('qubit'), random PVM measurement
↪   ('measurement') and
        the probabilities for each measurement outcome ('probabilities') in a
↪   nested structure including the
        theoretical probability ('born'), the execution runs ('runs') and the
↪   probability statistics ('stats')
    """

    # Alice prepares a random qubit
    qubit = qt.random.qubit()

    # Bob prepares a random measurement
    measurement = qt.random.pvm()

    experiment = {
        "qubit": qubit,
        "measurement": measurement,
        "probabilities": {
            "runs": np.zeros((shots, 2)),
            "stats": np.zeros((2,)),
            "born": np.ones((2,))
        }
    }

    for i in range(shots):

        # Alice and Bob's shared randomness
        shared_randomness = np.array([qt.random.bloch_vector(),
        ↪   qt.random.bloch_vector()])

        # Alice prepares
        alice = prepare(shared_randomness, qubit)
```

```
        # Bob measures
        bob = measure_pvm(shared_randomness, alice['bits'], measurement)

        # save simulation runs
        p = np.abs(bob['probabilities'])
        experiment['probabilities']['runs'][i, :] = p

        # accumulate counts according to Bob's probabilities
        index = random.choices(range(0, 2), cum_weights=np.cumsum(p), k=1)[0]
        experiment['probabilities']['stats'][index] =
        ↪  experiment['probabilities']['stats'][index] + 1

    experiment['probabilities']['stats'] = experiment['probabilities']['stats'] /
    ↪  shots
    experiment['probabilities']['born'] = bob['measurement'].probability(qubit)

    return experiment


def measure_povm(lambdas, bits, measurement: POVM):
    """
    Bob receives two bits from Alice and performs a random POVM

    Parameters
    ----------
    lambdas : ndarray
        Shared randomness as two normalized vectors in a numpy 2-d array

    bits: ndarray
        Bits communicated by Alice in a numpy 1-d array

    measurement: POVM
        A uniformly sampled POVM

    Returns
    -------
    dict
        A dictionary with the random measurement ('measurement') and
        the probabilities for each measurement outcome ('probabilities')
    """
    # flip shared randomness
    flip = np.where(bits == 0, -1, 1).reshape(2, 1)
    lambdas = np.multiply(lambdas, flip)

    # pick bloch vectors and weights from POVM elements
    y = measurement.bloch
    pb = measurement.weights / 2.
```

```
    # compute lambda according to the probabilities {pb}
    index = random.choices(range(0, measurement.size()),
    ↪  cum_weights=np.cumsum(pb), k=1)[0]
    a = np.abs(np.matmul(lambdas, y.T))
    _lambda = lambdas[np.argmax(a, axis=0)[index]]

    # compute probabilities
    thetas = theta(np.matmul(y, _lambda.reshape(-1, 1)))
    weighted_thetas = np.multiply(thetas, pb.reshape(-1, 1))
    p = weighted_thetas[:, 0] / np.sum(weighted_thetas, axis=0)

    return {
        "measurement": measurement,
        "probabilities": p
    }


def prepare_and_measure_povm(shots, n=4, qubit=None, measurement=None):
    """
    Runs a prepare-and-measure classical simulation with a random POVM
↪   measurement

    Parameters
    ----------
    shots : int
        Number of shots the simulation is run with

    n: int, optional
        Number of POVM random elements. Used if no measurement argument is
↪   specified, default value is 4.

    qubit : Qubit, optional
        Alice's qubit state. If not specified, a random qubit state will be used
↪   instead

    measurement: POVM, optional
        Bob's POVM measurement. If not specified a random POVM will be used
↪   instead


    Returns
    -------
    dict
        A dictionary with the random state ('qubit'), random POVM measurement
↪   ('measurement') and
        the probabilities for each measurement outcome ('probabilities') in a
↪   nested structure including the
```

```
        theoretical probability ('born'), the execution runs ('runs') and the
↪   probability statistics ('stats')
    """

    if qubit is None:
        # Alice prepares a random qubit
        qubit = qt.random.qubit()

    if measurement is None:
        # Bob prepares a random measurement
        measurement = qt.random.povm(n)

    n = measurement.size()

    experiment = {
        "qubit": qubit,
        "measurement": measurement,
        "probabilities": {
            "runs": np.zeros((shots, n)),
            "stats": np.zeros((n,)),
            "born": np.ones((n,))
        }
    }

    for i in range(shots):

        # Alice and Bob's shared randomness
        shared_randomness = np.array([qt.random.bloch_vector(),
        ↪   qt.random.bloch_vector()])

        # Alice prepares
        alice = prepare(shared_randomness, qubit)

        # Bob measures
        bob = measure_povm(shared_randomness, alice['bits'], measurement)

        # save simulation runs
        p = np.abs(bob['probabilities'])
        experiment['probabilities']['runs'][i, :] = p

        # accumulate counts according to Bob's probabilities
        index = random.choices(range(0, n), cum_weights=np.cumsum(p), k=1)[0]
        experiment['probabilities']['stats'][index] =
        ↪   experiment['probabilities']['stats'][index] + 1

    experiment['probabilities']['stats'] = experiment['probabilities']['stats'] /
    ↪   shots
    experiment['probabilities']['born'] = bob['measurement'].probability(qubit)
```

```
    return experiment


def bell_singlet_full(shots, alice, bob):
    """
    Runs a classical simulation on a Bell singlet state for a set of local
↪   observables

    Parameters
    ---------
    shots : int
        Number of shots the simulation is run with

    alice: tuple[Observable, Observable]
        Alice's local projective measurements described as a tuple of observables

    bob: tuple[Observable, Observable]
        Bob's local projective measurements described as a tuple of observables

    Returns
    -------
    dict
        A dictionary with the Bell state ('state'), Alice's and Bob's local
↪   projective measurements ('alice', 'bob')
        and the joint probabilities for each measurement outcome ('probabilities')
↪   in a nested structure including the
        theoretical probabilities ('born'), the execution runs ('runs') and the
↪   probability statistics ('stats')
    """

    if type(alice) is not tuple:
        raise ValueError('Alice\'s observables is not a valid tuple')
    elif len(alice) != 2:
        raise ValueError('Alice\'s number of observables is not
        ↪  valid:{}'.format(str(len(alice))))

    if type(bob) is not tuple:
        raise ValueError('Bob\'s observables is not a valid tuple')
    elif len(bob) != 2:
        raise ValueError('Bob\'s number of observables is not
        ↪  tuple:{}'.format(str(len(bob))))

    state = BellState.PSI_MINUS

    experiment = {
        "state": state,
        "alice": alice,
```

```
        "bob": bob,
        "probabilities": {
            "runs": np.zeros((shots, 4, 4)),
            "stats": np.zeros((4, 4)),
            "born": np.zeros((4, 4))
        }
    }

    # Compute theoretical probabilities
    bell = BellScenario(state, alice, bob)
    experiment['probabilities']['born'] = bell.probability().T

    for i in range(2):

        # Alice's state corresponding to the positive local projector
        a = Qubit(alice[i].eigenvector(1))

        for j in range(2):

            # Bob's states corresponding to the positive and negative local
            ↪  projectors
            b = (Qubit(bob[j].eigenvector(1)), Qubit(bob[j].eigenvector(-1)))

            ab = bell_singlet(shots, a, b)

            ij = int('{}{}'.format(i, j), 2)
            experiment['probabilities']['runs'][:, :, ij] =
            ↪  ab['probabilities']['runs']
            experiment['probabilities']['stats'][ij] =
            ↪  ab['probabilities']['stats']

    return experiment


def bell_singlet(shots, a, b):
    """
    Runs a classical simulation on a Bell singlet state for a set of states
↪   corresponding to local projection valued
    measurements

    Parameters
    ----------
    shots : int
        Number of shots the simulation is run with

    a: Qubit
        Alice's state corresponding to the positive local projection valued
↪   measurement operator
```

```
    b: tuple(Qubit, Qubit)
        Bob's states corresponding to the positive and negative local projection
↪   valued measurement operators

    Returns
    -------
    dict
        A dictionary with the joint probabilities for each measurement outcome
↪   ('probabilities') in a nested structure
        including the execution runs ('runs') and the probability statistics
↪   ('stats')
    """

    experiment = {
        "probabilities": {
            "runs": np.zeros((shots, 4)),
            "stats": np.zeros((4,))
        }
    }

    # Alice's positive local projector as bloch vector
    x = np.asarray([a.bloch_vector()])

    # Bob's local projectors as bloch vectors
    y = np.asarray([b[0].bloch_vector(), b[1].bloch_vector()])

    for i in range(shots):

        # Alice and Bob's shared randomness
        lambda1, lambda2 = qt.random.bloch_vector(), qt.random.bloch_vector()

        # Alice performs local projective measurements
        a = - np.sign(x @ lambda1)

        # Alice sends bit to Bob
        c = -a * np.sign(x @ lambda2)

        # Bob flips the lambda if c = -1
        lambda2 = c * lambda2

        # compute lambda according to the probabilities {pb}
        lambdas = np.array([lambda1, lambda2])

        pb = 0.5 * np.array([1, 1])
        index = random.choices(range(0, 2), cum_weights=np.cumsum(pb), k=1)[0]
        ly = np.abs(np.matmul(lambdas, y.T))
        _lambda = lambdas[np.argmax(ly, axis=0)[index]]
```

```
        # compute probabilities
        thetas = theta(np.matmul(y, _lambda.reshape(-1, 1)))
        weighted_thetas = np.multiply(thetas, pb.reshape(-1, 1))
        p = weighted_thetas[:, 0] / np.sum(weighted_thetas, axis=0)

        bits = '{}{}'.format(int(0.5 * (1 - a[0])), np.where(p == 1)[0][0])
        index = int(bits, 2)
        experiment['probabilities']['stats'][index] += 1

    experiment['probabilities']['stats'] = experiment['probabilities']['stats'] /
    ↪  shots

    return experiment
```

A.8 quantum.py

This module implements the prepare-and-measure for a given input state and positive operator-valued measure in a quantum simulator using Qiskit.

```
import numpy as np
from qiskit import QuantumCircuit, transpile, Aer, IBMQ
from qt.qubit import Qubit
from qt.measurement import POVM


def prepare_and_measure_povm(shots, qubit: Qubit, povm: POVM):
    # TODO extend usage to any POVM of N elements (currently N =4)
    qc = QuantumCircuit(2, 2)
    u = povm.unitary()

    qc.initialize(qubit.ket(), 0)
    qc.unitary(u, [0, 1])
    qc.measure([0, 1], [0, 1])

    backend = Aer.get_backend('aer_simulator')
    qc_transpiled = transpile(qc, backend)

    job = backend.run(qc_transpiled, shots=shots, memory=True)
    result = job.result()
    counts = result.get_counts(qc_transpiled)

    p = np.array([counts['00'], counts['01'], counts['10'], counts['11']])
    p = p / np.sum(p)

    results = {
        "counts": counts,
        "memory": result.get_memory(),
        "probabilities": p
    }
    return results
```

A.9 main.py

This is the main program, which runs the classical and quantum simulations as well as other experiments discussed throughout this document.

```
import math
import matplotlib as mpl
import matplotlib.pyplot as plt
from healpy.pixelfunc import ang2pix
from scipy.special import rel_entr

import qt.classical
import qt.quantum
import qt.random

from qt.qubit import X, Y, Z, Qubit
from qt.bell import BellScenario, BellState
from qt.measurement import POVM
from qt.observable import Observable
from qt.visualization import *

from qiskit import transpile
from qiskit import execute, Aer, IBMQ
from qiskit.visualization import plot_histogram
from qiskit import QuantumCircuit
from qiskit.tools.monitor import job_monitor


def random_states():
    size = 100000
    n = 4
    pixels = 12 * n ** 2
    indexes = np.zeros(size)
    for i in range(size):
        theta, phi = qt.random.qubit().bloch_angles()
        pix = ang2pix(n, theta, phi)
        indexes[i] = pix

    mpl.rcParams['mathtext.fontset'] = 'stix'
    mpl.rcParams['font.family'] = 'STIXGeneral'

    fig, ax = plt.subplots()

    count, bins, ignored = ax.hist(indexes, bins=range(pixels + 1), density=True,
    ↪  fill=True, facecolor='whitesmoke',
                                   edgecolor='k', hatch='', linewidth=1,
                                   ↪  histtype='step')
```

```
    ax.plot(bins, np.ones_like(bins) / pixels, linewidth=2, color='b',
    ↪  linestyle='-', zorder=2)
    ax.set_xlabel('Pixel indices', labelpad=6)
    ax.set_xticks(np.append(np.arange(0, pixels, 25), pixels))
    ax.set_ylabel('Frequency', labelpad=6)
    # ax.set_yticklabels(ax.get_yticklabels() * pixels)
    ax.set_xlim(0, pixels)
    plt.show()
    return None


def random_povm():
    np.random.seed(0)
    q1 = qt.random.qubit()
    q2 = qt.random.qubit()

    povm = POVM.new(np.array([q1, q2]))
    elements = povm.elements

    for i in range(elements.shape[0]):
        print('\nE{} eigenvalues -> {}'.format(i, np.linalg.eig(elements[i])[0]))
        print('\nE{}=\n{}'.format(i, elements[i]))
        print('E{} >=0 > -> {}'.format(i, (np.all(np.linalg.eig(elements[i])[0]
        ↪  >= -np.finfo(np.float32).eps))))

    print('Sum E_i = I -> {}'.format(np.allclose(np.identity(2), np.sum(elements,
    ↪  axis=0))))

    return None


def pm_pvm(shots):
    # run experiment
    np.random.seed(0)
    experiment = qt.classical.prepare_and_measure_pvm(shots)

    # plot probability convergence
    qubit = experiment['qubit']
    runs = experiment['probabilities']['runs']
    stats = experiment['probabilities']['stats']
    born = experiment['probabilities']['born']
    print('Qubit: {}'.format(str(qubit)))
    print('Stats:\np1={},p2={},pt={}'.format(stats[0], stats[1], np.sum(stats)))
    print('Born:\np1={},p2={},pt={}'.format(born[0], born[1], np.sum(born)))

    p = np.cumsum(runs[:, 0]) / (np.arange(len(runs[:, 0])) + 1)

    plt.plot(p)
```

```
    plt.axhline(y=born[0], color='r', linestyle='-')
    plt.show()
    return None


def pm_random(shots):
    # run experiment
    # np.random.seed(1200)
    np.random.seed(0)
    experiment = qt.classical.prepare_and_measure_povm(shots, 4)

    # plot probability convergence
    qubit = experiment['qubit']
    runs = experiment['probabilities']['runs']
    stats = experiment['probabilities']['stats']
    born = experiment['probabilities']['born']
    print('Qubit: {}'.format(str(qubit)))
    print('Stats:\np1={}, p2={}, p3={}, p4={}, pt={}'.format(stats[0], stats[1],
    ↪  stats[2], stats[3], np.sum(stats)))
    print('Born:\np1={}, p2={}, p3={}, p4={}, pt={}'.format(born[0], born[1],
    ↪  born[2], born[3], np.sum(born)))

    p1 = np.cumsum(runs[:, 0]) / (np.arange(len(runs[:, 0])) + 1)
    p2 = np.cumsum(runs[:, 1]) / (np.arange(len(runs[:, 1])) + 1)
    p3 = np.cumsum(runs[:, 2]) / (np.arange(len(runs[:, 2])) + 1)
    p4 = np.cumsum(runs[:, 3]) / (np.arange(len(runs[:, 3])) + 1)

    plt.plot(p1, color='r')
    plt.plot(p2, color='g')
    plt.plot(p3, color='b')
    plt.plot(p4, color='y')
    plt.axhline(y=born[0], color='r', linestyle='-')
    plt.axhline(y=born[1], color='g', linestyle='-')
    plt.axhline(y=born[2], color='b', linestyle='-')
    plt.axhline(y=born[3], color='y', linestyle='-')
    plt.title('Prepare and Measure with random state and POVM')
    plt.show()
    return None


def pm_trine(shots):
    # run experiment
    psi = qt.qubit.Qubit(np.array([(3 + 1.j * math.sqrt(3)) / 4., -0.5]))

    one = Qubit(np.array([1, 0])).rho()
    two = Qubit(0.5 * np.array([1, math.sqrt(3)])).rho()
    three = Qubit(0.5 * np.array([1, -math.sqrt(3)])).rho()
```

```
    povm = POVM(weights=2./3 * np.array([1, 1, 1]), proj=np.array([one, two,
    ↪  three], dtype=complex))

    experiment = qt.classical.prepare_and_measure_povm(shots, qubit=psi,
    ↪  measurement=povm)

    # plot probability convergence
    runs = experiment['probabilities']['runs']
    stats = experiment['probabilities']['stats']
    born = experiment['probabilities']['born']
    print('Stats:\np1={}, p2={}, p3={}, pt={}'.format(stats[0], stats[1],
    ↪  stats[2], np.sum(stats)))
    print('Born:\np1={}, p2={}, p3={}, pt={}'.format(born[0], born[1], born[2],
    ↪  np.sum(born)))

    p1 = np.cumsum(runs[:, 0]) / (np.arange(len(runs[:, 0])) + 1)
    p2 = np.cumsum(runs[:, 1]) / (np.arange(len(runs[:, 1])) + 1)
    p3 = np.cumsum(runs[:, 2]) / (np.arange(len(runs[:, 2])) + 1)

    plt.plot(p1, color='r')
    plt.plot(p2, color='g')
    plt.plot(p3, color='b')
    plt.axhline(y=born[0], color='r', linestyle='-')
    plt.axhline(y=born[1], color='g', linestyle='-')
    plt.axhline(y=born[2], color='b', linestyle='-')
    plt.title('Prepare and Measure with Trine POVM')
    plt.show()

    return None


def pm_cross(shots):
    # run experiment
    psi = qt.qubit.Qubit(np.array([(3 + 1.j * math.sqrt(3)) / 4., -0.5]))

    zero = np.array([[1, 0], [0, 0]])
    one = np.array([[0, 0], [0, 1]])
    plus = 0.5 * np.array([[1, 1], [1, 1]])
    minus = 0.5 * np.array([[1, -1], [-1, 1]])

    povm = POVM(weights=0.5 * np.array([1, 1, 1, 1]), proj=np.array([zero, one,
    ↪  plus, minus], dtype=complex))

    experiment = qt.classical.prepare_and_measure_povm(shots, qubit=psi,
    ↪  measurement=povm)

    # plot probability convergence
    runs = experiment['probabilities']['runs']
```

```
    stats = experiment['probabilities']['stats']
    born = experiment['probabilities']['born']
    print('Stats:\np1={}, p2={}, p3={}, pt={}'.format(stats[0], stats[1],
    ↪  stats[2], np.sum(stats)))
    print('Born:\np1={}, p2={}, p3={}, pt={}'.format(born[0], born[1], born[2],
    ↪  np.sum(born)))

    p1 = np.cumsum(runs[:, 0]) / (np.arange(len(runs[:, 0])) + 1)
    p2 = np.cumsum(runs[:, 1]) / (np.arange(len(runs[:, 1])) + 1)
    p3 = np.cumsum(runs[:, 2]) / (np.arange(len(runs[:, 2])) + 1)

    plt.plot(p1, color='r')
    plt.plot(p2, color='g')
    plt.plot(p3, color='b')
    plt.axhline(y=born[0], color='r', linestyle='-')
    plt.axhline(y=born[1], color='g', linestyle='-')
    plt.axhline(y=born[2], color='b', linestyle='-')
    plt.title('Prepare and Measure with Cross POVM')
    plt.show()

    return None


def pm_sic(shots):
    # run experiment
    psi = qt.qubit.Qubit(np.array([(3 + 1.j * math.sqrt(3)) / 4., -0.5]))

    one = Qubit(np.array([1, 0])).rho()
    two = Qubit(np.array([1/math.sqrt(3), math.sqrt(2/3)])).rho()
    three = Qubit(np.array([1/math.sqrt(3),
                            math.sqrt(2/3) * (math.cos(2 * math.pi/3) + 1.j *
                            ↪  math.sin(2 * math.pi/3))])).rho()
    four = Qubit(np.array([1/math.sqrt(3),
                           math.sqrt(2/3) * (math.cos(4 * math.pi/3) + 1.j *
                           ↪  math.sin(4 * math.pi/3))])).rho()

    povm = POVM(weights=0.5 * np.array([1, 1, 1, 1]), proj=np.array([one, two,
    ↪  three, four], dtype=complex))

    experiment = qt.classical.prepare_and_measure_povm(shots, qubit=psi,
    ↪  measurement=povm)

    # plot probability convergence
    runs = experiment['probabilities']['runs']
    stats = experiment['probabilities']['stats']
    born = experiment['probabilities']['born']
    print('Stats:\np1={}, p2={}, p3={}, p4={}, pt={}'.format(stats[0], stats[1],
    ↪  stats[2], stats[3], np.sum(stats)))
```

```
    print('Born:\np1={}, p2={}, p3={}, p4={}, pt={}'.format(born[0], born[1],
    ↪  born[2], born[3], np.sum(born)))

    p1 = np.cumsum(runs[:, 0]) / (np.arange(len(runs[:, 0])) + 1)
    p2 = np.cumsum(runs[:, 1]) / (np.arange(len(runs[:, 1])) + 1)
    p3 = np.cumsum(runs[:, 2]) / (np.arange(len(runs[:, 2])) + 1)
    p4 = np.cumsum(runs[:, 3]) / (np.arange(len(runs[:, 3])) + 1)

    plt.plot(p1, color='r')
    plt.plot(p2, color='g')
    plt.plot(p3, color='b')
    plt.plot(p4, color='y')
    plt.axhline(y=born[0], color='r', linestyle='-')
    plt.axhline(y=born[1], color='g', linestyle='-')
    plt.axhline(y=born[2], color='b', linestyle='-')
    plt.axhline(y=born[3], color='y', linestyle='-')
    plt.title('Prepare and Measure with SIC-POVM of 4 elements')
    plt.show()
    return None


def neumark():
    psi = qt.qubit.Qubit(np.array([(3 + 1.j * math.sqrt(3)) / 4., -0.5]))
    print(psi.bloch_vector())

    zero = np.array([[1, 0], [0, 0]])
    one = np.array([[0, 0], [0, 1]])
    plus = 0.5 * np.array([[1, 1], [1, 1]])
    minus = 0.5 * np.array([[1, -1], [-1, 1]])

    povm = POVM(weights=0.5 * np.array([1, 1, 1, 1]), proj=np.array([zero, one,
    ↪  plus, minus], dtype=complex))
    unitary = povm.unitary()
    print(unitary)
    return None


def pm_circuit():
    qubit = qt.qubit.Qubit(np.array([(3 + 1.j * math.sqrt(3)) / 4., -0.5]))

    zero = np.array([[1, 0], [0, 0]])
    one = np.array([[0, 0], [0, 1]])
    plus = 0.5 * np.array([[1, 1], [1, 1]])
    minus = 0.5 * np.array([[1, -1], [-1, 1]])
    povm = POVM(weights=0.5 * np.array([1, 1, 1, 1]), proj=np.array([zero, one,
    ↪  plus, minus], dtype=complex))
    print(povm.unitary())
```

```
    shots = 10 ** 7
    results = qt.quantum.prepare_and_measure_povm(shots, qubit, povm)
    print('Probabilities={}'.format(results["probabilities"]))
    return None


def quantum_simulator():
    qc = QuantumCircuit(2, 2)

    psi = ((3 + 1.j * math.sqrt(3)) / 4., -0.5)

    U = [[0.70710678 + 0.j, 0. + 0.j, 0.70710678 + 0.j, 0. + 0.j],
         [0. + 0.j, 0.70710678 + 0.j, 0. + 0.j, 0.70710678 + 0.j],
         [0.5 - 0.j, 0.5 + 0.j, -0.5 + 0.j, -0.5 + 0.j],
         [-0.5 + 0.j, 0.5 + 0.j, 0.5 + 0.j, -0.5 + 0.j]]

    qc.initialize(psi, 0)
    qc.unitary(U, [0, 1])
    qc.measure([0, 1], [0, 1])
    qc.draw()

    backend = Aer.get_backend('aer_simulator')
    qc_transpiled = transpile(qc, backend)
    qc_transpiled.draw()

    job = backend.run(qc_transpiled, shots=4000)
    result = job.result()
    counts = result.get_counts(qc_transpiled)

    print(counts)
    plot_histogram(counts)

    sum(counts.values())
    print(counts['00'] / sum(counts.values()))
    print(counts['01'] / sum(counts.values()))
    print(counts['10'] / sum(counts.values()))
    print(counts['11'] / sum(counts.values()))


def quantum_computer():
    qc = QuantumCircuit(2, 2)

    psi = ((3 + 1.j * math.sqrt(3)) / 4., -0.5)

    U = [[0.70710678 + 0.j, 0. + 0.j, 0.70710678 + 0.j, 0. + 0.j],
         [0. + 0.j, 0.70710678 + 0.j, 0. + 0.j, 0.70710678 + 0.j],
         [- 0.5 + 0.j, -0.5 + 0.j, 0.5 + 0.j, 0.5 + 0.j],
         [-0.5 + 0.j, 0.5 + 0.j, 0.5 + 0.j, -0.5 + 0.j]]
```

```
    qc.initialize(psi, 0)
    qc.unitary(U, [0, 1])
    qc.measure([0, 1], [0, 1])
    qc.draw()

    IBMQ.load_account()
    provider = IBMQ.get_provider(hub='ibm-q', group='open', project='main')
    qcomp = provider.get_backend('ibm_nairobi')
    # running in ibm_nairobi. 4000 shots

    qc_transpiled = transpile(qc, backend=qcomp)
    job = execute(qc_transpiled, backend=qcomp, shots=4000)
    job_monitor(job)
    result = job.result()
    counts = result.get_counts(qc_transpiled)
    plot_histogram(counts)
    sum(counts.values())
    print(counts['00'] / sum(counts.values()))
    print(counts['01'] / sum(counts.values()))
    print(counts['10'] / sum(counts.values()))
    print(counts['11'] / sum(counts.values()))


def pm_kl_classical_born():
    """
    Runs PM classical protocol and plots Kullback-Leibler divergence among
↪   classical and Born probability distributions
    """

    # run experiment
    np.random.seed(0)
    shots = 10 ** 4

    qubit = qt.qubit.Qubit(np.array([(3 + 1.j * math.sqrt(3)) / 4., -0.5]))
    # P4 = {1/2|0x0|, 1/2|1x1|, 1/2|+x+|, 1/2|-x-|}
    proj = np.array([[[1, 0], [0, 0]], [[0, 0], [0, 1]], [[.5, .5], [.5, .5]],
    ↪  [[.5, -.5], [-.5, .5]]])
    measurement = POVM(weights=0.5 * np.array([1, 1, 1, 1]), proj=proj)

    experiment = qt.classical.prepare_and_measure_povm(shots, 4, qubit=qubit,
    ↪  measurement=measurement)

    # plot Kullback-Leibler divergence
    runs = experiment['probabilities']['runs']
    stats = experiment['probabilities']['stats']
    born = experiment['probabilities']['born']
```

```
    print('Stats:\np1={}, p2={}, p3={}, p4={}, pt={}'.format(stats[0], stats[1],
    ↪  stats[2], stats[3], np.sum(stats)))
    print('Born:\np1={}, p2={}, p3={}, p4={}, pt={}'.format(born[0], born[1],
    ↪  born[2], born[3], np.sum(born)))

    p1 = np.cumsum(runs[:, 0]) / (np.arange(len(runs[:, 0])) + 1)
    p2 = np.cumsum(runs[:, 1]) / (np.arange(len(runs[:, 1])) + 1)
    p3 = np.cumsum(runs[:, 2]) / (np.arange(len(runs[:, 2])) + 1)
    p4 = np.cumsum(runs[:, 3]) / (np.arange(len(runs[:, 3])) + 1)

    actual = np.vstack((p1, p2, p3, p4))
    expected = np.repeat(born.reshape(born.shape[0], 1), actual.shape[1], axis=1)

    rows, cols = actual.shape
    kl = np.zeros((cols,))
    for i in range(cols):
        kl[i] = sum(rel_entr(expected[:, i], actual[:, i]))

    plt.plot(kl, color='b')
    plt.show()
    return None


def pm_kl_classical_quantum_simulator():
    """
    Runs classical protocol and quantum simulator and plots Kullback-Leibler
↪   divergence among classical and
    quantum simulator probability distribution
    """
    shots = 10 ** 4

    qubit = qt.qubit.Qubit(np.array([(3 + 1.j * math.sqrt(3)) / 4., -0.5]))
    # P4 = {1/2|0x0|, 1/2|1x1|, 1/2|+x+|, 1/2|-x-|}
    proj = np.array([[[1, 0], [0, 0]], [[0, 0], [0, 1]], [[.5, .5], [.5, .5]],
    ↪  [[.5, -.5], [-.5, .5]]], dtype=complex)
    measurement = POVM(weights=0.5 * np.array([1, 1, 1, 1]), proj=proj)

    # run classical protocol
    experiment1 = qt.classical.prepare_and_measure_povm(shots, qubit=qubit,
    ↪  measurement=measurement)
    runs = experiment1['probabilities']['runs']
    born = experiment1['probabilities']['born']
    p1 = np.cumsum(runs[:, 0]) / (np.arange(len(runs[:, 0])) + 1)
    p2 = np.cumsum(runs[:, 1]) / (np.arange(len(runs[:, 1])) + 1)
    p3 = np.cumsum(runs[:, 2]) / (np.arange(len(runs[:, 2])) + 1)
    p4 = np.cumsum(runs[:, 3]) / (np.arange(len(runs[:, 3])) + 1)
    experimental1 = np.vstack((p1, p2, p3, p4))
```

```
    theoretical = np.repeat(born.reshape(born.shape[0], 1),
    ↪  experimental1.shape[1], axis=1)
    print('Classical protocol executed')

    # run quantum circuit in Qiskit simulator
    experiment2 = qt.quantum.prepare_and_measure_povm(shots, qubit, measurement)
    import collections
    memory = experiment2["memory"]
    experimental2 = np.zeros(experimental1.shape)
    for i in range(len(memory)):
        summary = collections.Counter(memory[0: i + 1])
        summary = np.array([summary[k] for k in sorted(summary.keys())])
        p = np.zeros((measurement.size(),))
        p[:summary.shape[0]] = summary / np.sum(summary)
        experimental2[:, i] = p
    print('Quantum Circuit executed')

    # plot kl divergence
    _, cols = experimental1.shape
    klte1 = np.zeros((cols,))
    klte2 = np.zeros((cols,))
    klee = np.zeros((cols,))

    for i in range(cols):
        klte1[i] = sum(rel_entr(theoretical[:, i], experimental1[:, i]))
        klte2[i] = sum(rel_entr(theoretical[:, i], experimental2[:, i]))
        klee[i] = sum(rel_entr(experimental2[:, i], experimental1[:, i]))

    plt.title('Kullback-Leibler divergence {:.0E} shots'.format(shots))
    plt.plot(klte1, color='b', label='Born vs Classical Protocol')
    plt.plot(klte2, color='r', label='Born vs Quantum Simulator')
    plt.legend()
    plt.show()
    return None


def pm_kl_classical_quantum_simulator_born(shots):
    np.random.seed(1976)
    mpl.rcParams['mathtext.fontset'] = 'stix'
    mpl.rcParams['font.family'] = 'STIXGeneral'

    fig, ax = plt.subplots(1, 1, layout='constrained')

    ax.xaxis.set_tick_params(which='major', size=5, width=1, direction='in',
    ↪  top='on')
    ax.xaxis.set_tick_params(which='minor', size=3, width=1, direction='in',
    ↪  top='on')
```

```
ax.yaxis.set_tick_params(which='major', size=5, width=1, direction='in',
↪  right='on')
ax.yaxis.set_tick_params(which='minor', size=3, width=1, direction='in',
↪  right='on')

fig.suptitle(r'Cross-POVM')
fig.supxlabel('Number of shots')
fig.supylabel('Kullback-Leibler divergence')

qubit = qt.qubit.Qubit(np.array([(3 + 1.j * math.sqrt(3)) / 4., -0.5]))
proj = np.array([[[1, 0], [0, 0]], [[0, 0], [0, 1]], [[.5, .5], [.5, .5]],
↪  [[.5, -.5], [-.5, .5]]], dtype=complex)
measurement = POVM(weights=0.5 * np.array([1, 1, 1, 1]), proj=proj)

# run classical protocol
experiment1 = qt.classical.prepare_and_measure_povm(shots, qubit=qubit,
↪  measurement=measurement)
runs = experiment1['probabilities']['runs']
stats = experiment1['probabilities']['stats']
born = experiment1['probabilities']['born']
p1 = np.cumsum(runs[:, 0]) / (np.arange(len(runs[:, 0])) + 1)
p2 = np.cumsum(runs[:, 1]) / (np.arange(len(runs[:, 1])) + 1)
p3 = np.cumsum(runs[:, 2]) / (np.arange(len(runs[:, 2])) + 1)
p4 = np.cumsum(runs[:, 3]) / (np.arange(len(runs[:, 3])) + 1)
experimental1 = np.vstack((p1, p2, p3, p4))
theoretical = np.repeat(born.reshape(born.shape[0], 1),
↪  experimental1.shape[1], axis=1)
print('Stats:p1={}, p2={}, p3={}, p4={}, pt={}'.format(stats[0], stats[1],
↪  stats[2], stats[3], np.sum(stats)))
print('Born:p1={}, p2={}, p3={}, p4={}, pt={}'.format(born[0], born[1],
↪  born[2], born[3], np.sum(born)))
print('Classical protocol executed')

# run quantum circuit in Qiskit simulator
experiment2 = qt.quantum.prepare_and_measure_povm(shots, qubit=qubit,
↪  povm=measurement)
import collections
memory = experiment2["memory"]
print('Stats={}'.format(experiment2["probabilities"]))
print('Quantum Circuit executed')

experimental2 = np.zeros(experimental1.shape)
for i in range(len(memory)):
    summary = collections.Counter(memory[0: i + 1])
    summary = np.array([summary[k] for k in sorted(summary.keys())])
    p = np.zeros((measurement.size(),))
    p[:summary.shape[0]] = summary / np.sum(summary)
    experimental2[:, i] = p
```

```
    # plot kl divergence
    _, cols = experimental1.shape
    klte1 = np.zeros((cols,))
    klte2 = np.zeros((cols,))
    kle1t = np.zeros((cols,))
    kle1e2 = np.zeros((cols,))

    for i in range(cols):
        klte1[i] = sum(rel_entr(theoretical[:, i], experimental1[:, i]))
        klte2[i] = sum(rel_entr(theoretical[:, i], experimental2[:, i]))
        kle1t[i] = sum(rel_entr(experimental1[:, i], theoretical[:, i]))
        kle1e2[i] = sum(rel_entr(experimental1[:, i], experimental2[:, i]))

    ax.plot(klte1, color='b', label='Born vs. Classical Protocol', linewidth=2)
    ax.plot(klte2, color='r', label='Born vs. Quantum Simulator', linewidth=2,
    ↪  linestyle='-', alpha=0.7)
    ax.plot(kle1e2, color='g', label='Classical Protocol vs. Quantum Simulator',
    ↪  linewidth=2, linestyle='-', alpha=0.7)
    ax.legend()

    plt.show()
    return None


def pm_kl_multiplot(shots):
    np.random.seed(0)
    mpl.rcParams['mathtext.fontset'] = 'stix'
    mpl.rcParams['font.family'] = 'STIXGeneral'

    cases = ['Cross-POVM', 'Trine-POVM', 'SIC-POVM', r"Random-PVM",
    ↪  r"Random-POVM", r"Random-POVM"]

    fig, axs = plt.subplots(3, 2, figsize=(8, 10), layout='constrained')

    for ax, title in zip(axs.flat, cases):
        ax.xaxis.set_tick_params(which='major', size=5, width=1, direction='in',
        ↪  top='on')
        ax.xaxis.set_tick_params(which='minor', size=3, width=1, direction='in',
        ↪  top='on')
        ax.yaxis.set_tick_params(which='major', size=5, width=1, direction='in',
        ↪  right='on')
        ax.yaxis.set_tick_params(which='minor', size=3, width=1, direction='in',
        ↪  right='on')
        # ax.set_ylabel(r'£D_{KL}£', labelpad=2)
        ax.set_title(title)

    fig.supxlabel('Number of shots')
```

```
fig.supylabel('Kullback-Leibler divergence')

qubit = qt.qubit.Qubit(np.array([(3 + 1.j * math.sqrt(3)) / 4., -0.5]))

# EXPERIMENT 1: RANDOM-PVM
experiment1 = qt.classical.prepare_and_measure_pvm(shots)

qubit1 = experiment1['qubit']
runs1 = experiment1['probabilities']['runs']
stats1 = experiment1['probabilities']['stats']
born1 = experiment1['probabilities']['born']
print('EXPERIMENT 1:\nState:{}\nPVM:{}'.format(qubit1, 'RANDOM-PVM'))
print('Stats:p1={}, p2={}, pt={}'.format(stats1[0], stats1[1],
↪  np.sum(stats1)))
print('Born:p1={}, p2={}, pt={}'.format(born1[0], born1[1], np.sum(born1)))

p11 = np.cumsum(runs1[:, 0]) / (np.arange(len(runs1[:, 0])) + 1)
p12 = np.cumsum(runs1[:, 1]) / (np.arange(len(runs1[:, 1])) + 1)

actual1 = np.vstack((p11, p12))
expected1 = np.repeat(born1.reshape(born1.shape[0], 1), actual1.shape[1],
↪  axis=1)
plot_kl(axs[1][1], actual1, expected1)

# EXPERIMENT 2: CROSS-POVM
# P4 = {1/2|0x0|, 1/2|1x1|, 1/2|+x+|, 1/2|-x-|}
proj2 = np.array([[[1, 0], [0, 0]], [[0, 0], [0, 1]], [[.5, .5], [.5, .5]],
↪  [[.5, -.5], [-.5, .5]]])
measurement2 = POVM(weights=0.5 * np.array([1, 1, 1, 1]), proj=proj2)
experiment2 = qt.classical.prepare_and_measure_povm(shots, 4, qubit=qubit,
↪  measurement=measurement2)

runs2 = experiment2['probabilities']['runs']
stats2 = experiment2['probabilities']['stats']
born2 = experiment2['probabilities']['born']
print('EXPERIMENT 2:\nState:{}\nPOVM:{}'.format(qubit, 'CROSS-POVM'))
print('Stats:p1={}, p2={}, p3={}, p4={}, pt={}'.format(stats2[0], stats2[1],
↪  stats2[2], stats2[3], np.sum(stats2)))
print('Born:p1={}, p2={}, p3={}, p4={}, pt={}'.format(born2[0], born2[1],
↪  born2[2], born2[3], np.sum(born2)))

p21 = np.cumsum(runs2[:, 0]) / (np.arange(len(runs2[:, 0])) + 1)
p22 = np.cumsum(runs2[:, 1]) / (np.arange(len(runs2[:, 1])) + 1)
p23 = np.cumsum(runs2[:, 2]) / (np.arange(len(runs2[:, 2])) + 1)
p24 = np.cumsum(runs2[:, 3]) / (np.arange(len(runs2[:, 3])) + 1)

actual2 = np.vstack((p21, p22, p23, p24))
```

```
expected2 = np.repeat(born2.reshape(born2.shape[0], 1), actual2.shape[1],
↪  axis=1)
plot_kl(axs[0][0], actual2, expected2)

# EXPERIMENT 3: TRINE-POVM
one = Qubit(np.array([1, 0])).rho()
two = Qubit(0.5 * np.array([1, math.sqrt(3)])).rho()
three = Qubit(0.5 * np.array([1, -math.sqrt(3)])).rho()
measurement3 = POVM(weights=2. / 3 * np.array([1, 1, 1]), proj=np.array([one,
↪  two, three], dtype=complex))
experiment3 = qt.classical.prepare_and_measure_povm(shots, qubit=qubit,
↪  measurement=measurement3)

runs3 = experiment3['probabilities']['runs']
stats3 = experiment3['probabilities']['stats']
born3 = experiment3['probabilities']['born']
print('EXPERIMENT 3:\nState:{}\nPOVM:{}'.format(qubit, 'TRINE-POVM'))
print('Stats:p1={}, p2={}, p3={}, pt={}'.format(stats3[0], stats3[1],
↪  stats3[2], np.sum(stats3)))
print('Born:p1={}, p2={}, p3={}, pt={}'.format(born3[0], born3[1], born3[2],
↪  np.sum(born3)))

p31 = np.cumsum(runs3[:, 0]) / (np.arange(len(runs3[:, 0])) + 1)
p32 = np.cumsum(runs3[:, 1]) / (np.arange(len(runs3[:, 1])) + 1)
p33 = np.cumsum(runs3[:, 2]) / (np.arange(len(runs3[:, 2])) + 1)

actual3 = np.vstack((p31, p32, p33))
expected3 = np.repeat(born3.reshape(born3.shape[0], 1), actual3.shape[1],
↪  axis=1)
plot_kl(axs[0][1], actual3, expected3)

# EXPERIMENT 4: SIC-POVM
one = Qubit(np.array([1, 0])).rho()
two = Qubit(np.array([1/math.sqrt(3), math.sqrt(2/3)])).rho()
three = Qubit(np.array([1/math.sqrt(3),
                        math.sqrt(2/3) * (math.cos(2 * math.pi/3) + 1.j *
                        ↪  math.sin(2 * math.pi/3))])).rho()
four = Qubit(np.array([1/math.sqrt(3),
                       math.sqrt(2/3) * (math.cos(4 * math.pi/3) + 1.j *
                       ↪  math.sin(4 * math.pi/3))])).rho()
measurement4 = POVM(weights=0.5 * np.array([1, 1, 1, 1]), proj=np.array([one,
↪  two, three, four], dtype=complex))
experiment4 = qt.classical.prepare_and_measure_povm(shots, qubit=qubit,
↪  measurement=measurement4)

runs4 = experiment4['probabilities']['runs']
stats4 = experiment4['probabilities']['stats']
born4 = experiment4['probabilities']['born']
```

```
print('EXPERIMENT 4:\nState:{}\nPOVM:{}'.format(qubit, 'SIC-POVM'))
print('Stats:p1={}, p2={}, p3={}, p4={}, pt={}'.format(stats4[0], stats4[1],
↪  stats4[2], stats4[3], np.sum(stats4)))
print('Born:p1={}, p2={}, p3={}, p4={}, pt={}'.format(born4[0], born4[1],
↪  born4[2], born4[2], np.sum(born3)))

p41 = np.cumsum(runs4[:, 0]) / (np.arange(len(runs4[:, 0])) + 1)
p42 = np.cumsum(runs4[:, 1]) / (np.arange(len(runs4[:, 1])) + 1)
p43 = np.cumsum(runs4[:, 2]) / (np.arange(len(runs4[:, 2])) + 1)
p44 = np.cumsum(runs4[:, 3]) / (np.arange(len(runs4[:, 3])) + 1)

actual4 = np.vstack((p41, p42, p43, p44))
expected4 = np.repeat(born4.reshape(born4.shape[0], 1), actual4.shape[1],
↪  axis=1)
plot_kl(axs[1][0], actual4, expected4)

# EXPERIMENT 5: RANDOM-POVM-1
experiment5 = qt.classical.prepare_and_measure_povm(shots, 4)

qubit5 = experiment5['qubit']
runs5 = experiment5['probabilities']['runs']
stats5 = experiment5['probabilities']['stats']
born5 = experiment5['probabilities']['born']
print('EXPERIMENT 5:\nState:{}\nPOVM:{}'.format(qubit5, 'RANDOM-POVM-1'))
print('Stats:p1={}, p2={}, p3={}, p4={}, pt={}'.format(stats5[0], stats5[1],
↪  stats5[2], stats5[3], np.sum(stats5)))
print('Born:p1={}, p2={}, p3={}, p4={}, pt={}'.format(born5[0], born5[1],
↪  born5[2], born5[2], np.sum(born5)))

p51 = np.cumsum(runs5[:, 0]) / (np.arange(len(runs5[:, 0])) + 1)
p52 = np.cumsum(runs5[:, 1]) / (np.arange(len(runs5[:, 1])) + 1)
p53 = np.cumsum(runs5[:, 2]) / (np.arange(len(runs5[:, 2])) + 1)
p54 = np.cumsum(runs5[:, 3]) / (np.arange(len(runs5[:, 3])) + 1)

actual5 = np.vstack((p51, p52, p53, p54))
expected5 = np.repeat(born5.reshape(born5.shape[0], 1), actual5.shape[1],
↪  axis=1)
plot_kl(axs[2][0], actual5, expected5)

# EXPERIMENT 6: RANDOM-POVM-2
experiment6 = qt.classical.prepare_and_measure_povm(shots, 4)

qubit6 = experiment6['qubit']
runs6 = experiment6['probabilities']['runs']
stats6 = experiment6['probabilities']['stats']
born6 = experiment6['probabilities']['born']
print('EXPERIMENT 6:\nState:{}\nPOVM:{}'.format(qubit6, 'RANDOM-POVM-2'))
```

```
    print('Stats:p1={}, p2={}, p3={}, p4={}, pt={}'.format(stats6[0], stats6[1],
    ↪  stats6[2], stats6[3], np.sum(stats6)))
    print('Born:p1={}, p2={}, p3={}, p4={}, pt={}'.format(born6[0], born6[1],
    ↪  born6[2], born6[2], np.sum(born6)))

    p61 = np.cumsum(runs6[:, 0]) / (np.arange(len(runs6[:, 0])) + 1)
    p62 = np.cumsum(runs6[:, 1]) / (np.arange(len(runs6[:, 1])) + 1)
    p63 = np.cumsum(runs6[:, 2]) / (np.arange(len(runs6[:, 2])) + 1)
    p64 = np.cumsum(runs6[:, 3]) / (np.arange(len(runs6[:, 3])) + 1)

    actual6 = np.vstack((p61, p62, p63, p64))
    expected6 = np.repeat(born6.reshape(born6.shape[0], 1), actual6.shape[1],
    ↪  axis=1)
    plot_kl(axs[2][1], actual6, expected6)

    plt.show()
    return None


def plot_kl(ax, actual, expected, label='', color='b', linewidth=1,
↪  linestyle='-'):
    rows, cols = actual.shape
    kl = np.zeros((cols,))
    for i in range(cols):
        kl[i] = sum(rel_entr(expected[:, i], actual[:, i]))
    # ax.plot(kl, color=color, label=label, linewidth=linewidth,
    ↪  linestyle=linestyle)
    # ax.legend()
    ax.plot(kl, color=color, linewidth=linewidth, linestyle=linestyle)


def bell_sample_probabilities(shots):
    np.random.seed(0)

    a0 = Observable(Z)
    a1 = Observable(X)
    b0 = Observable(-1 / math.sqrt(2) * (X + Z))
    b1 = Observable(1 / math.sqrt(2) * (X - Z))

    alice = Qubit(a0.eigenvector(1))
    bob = (Qubit(b0.eigenvector(1)), Qubit(b0.eigenvector(-1)))

    experiment = qt.classical.bell_singlet_full(shots, alice=(a0, a1), bob=(b0,
    ↪  b1))
    stats = experiment['probabilities']['stats']
    born = experiment['probabilities']['born']
    print('Stats:\n{}'.format(stats))
    print('Born:\n{}'.format(born))
```

```
    return None


def bell(shots):
    np.random.seed(0)

    mpl.rcParams['mathtext.fontset'] = 'stix'
    mpl.rcParams['font.family'] = 'STIXGeneral'

    a0 = Observable(Z)
    a1 = Observable(X)
    b0 = Observable(-1 / math.sqrt(2) * (X + Z))
    b1 = Observable(1 / math.sqrt(2) * (X - Z))

    alice = Qubit(a0.eigenvector(1))
    bob = (Qubit(b0.eigenvector(1)), Qubit(b0.eigenvector(-1)))

    experiment = qt.classical.bell_singlet_full(shots, alice=(a0, a1), bob=(b0,
    ↪  b1))
    actual = experiment['probabilities']['stats'].T
    expected = experiment['probabilities']['born'].T

    x = np.arange(0, 5)
    y = np.arange(0, 5)

    outcomes = ["$(+1,+1)$", "$(+1,-1)$", "$(-1,+1)$", "$(-1,-1)$"]
    measurements = ["($A_0$,$B_0$)", "$(A_0,B_1)$", "$(A_1,B_0)$", "$(A_1,B_1)$"]

    fig, ax = plt.subplots(1, 2, figsize=(14, 5))

    # actual
    im = ax[0].imshow(actual, cmap='PiYG')

    ax[0].set_xticks(np.arange(len(measurements)), labels=measurements)
    ax[0].set_yticks(np.arange(len(outcomes)), labels=outcomes)

    for i in range(len(measurements)):
        for j in range(len(outcomes)):
            text = ax[0].text(j, i, round(actual[i, j], 4),
                              ha="center", va="center", color="w")

    ax[0].set_title("Classical Protocol")

    # expected
    im = ax[1].imshow(expected, cmap='PiYG')

    ax[1].set_xticks(np.arange(len(measurements)), labels=measurements)
    ax[1].set_yticks(np.arange(len(outcomes)), labels=outcomes)
```

```
    for i in range(len(measurements)):
        for j in range(len(outcomes)):
            text = ax[1].text(j, i, round(expected[i, j], 4),
                              ha="center", va="center", color="w")

    ax[1].set_title("Born\'s rule ")

    fig.tight_layout()
    plt.show()
    return None


def bell_multiplot():
    np.random.seed(0)

    mpl.rcParams['mathtext.fontset'] = 'stix'
    mpl.rcParams['font.family'] = 'STIXGeneral'

    cases = [r'$shots=10$', r'$shots=100$', r'$shots=500$',
             r'$shots=10^3$', r'$shots=5\cdot10^3$', r'$shots=10^4$',
             r'$shots=2\cdot10^4$', r'$shots=5\cdot10^4$', r'$shots=10^5$']
    outcomes = ["$(+1,+1)$", "$(+1,-1)$", "$(-1,+1)$", "$(-1,-1)$"]
    measurements = ["($A_0$,$B_0$)", "$(A_0,B_1)$", "$(A_1,B_0)$", "$(A_1,B_1)$"]

    fig, axs = plt.subplots(3, 3, figsize=(10, 10), layout='constrained')

    for ax, title in zip(axs.flat, cases):
        ax.xaxis.set_tick_params(which='major', size=5, width=1, direction='out',
        ↪  top='on')
        ax.xaxis.set_tick_params(which='minor', size=3, width=1, direction='out',
        ↪  top='on')
        ax.yaxis.set_tick_params(which='major', size=5, width=1, direction='out',
        ↪  right='on')
        ax.yaxis.set_tick_params(which='minor', size=3, width=1, direction='out',
        ↪  right='on')
        # ax.set_ylabel(r'£p_{C}(a_x, b_y | A_x, B_y)£', labelpad=2)
        ax.set_title(title)
        ax.set_xticks(np.arange(len(measurements)), labels=measurements)
        ax.set_yticks(np.arange(len(outcomes)), labels=outcomes)

    fig.supxlabel(r'$(A_x, B_y)$')
    fig.supylabel(r'$(a_x, b_y)$')
    fig.suptitle(r'$p_{C}(a_x, b_y\:|\:A_x, B_y)$')

    a0 = Observable(Z)
    a1 = Observable(X)
    b0 = Observable(-1 / math.sqrt(2) * (X + Z))
```

```
b1 = Observable(1 / math.sqrt(2) * (X - Z))

alice = Qubit(a0.eigenvector(1))
bob = (Qubit(b0.eigenvector(1)), Qubit(b0.eigenvector(-1)))

# shots = 10
experiment1 = qt.classical.bell_singlet_full(10, alice=(a0, a1), bob=(b0,
↪  b1))
actual1 = experiment1['probabilities']['stats'].T
axs[0][0].imshow(actual1, cmap='PiYG')

# shots = 100
experiment2 = qt.classical.bell_singlet_full(100, alice=(a0, a1), bob=(b0,
↪  b1))
actual2 = experiment2['probabilities']['stats'].T
axs[0][1].imshow(actual2, cmap='PiYG')

# shots = 500
experiment3 = qt.classical.bell_singlet_full(500, alice=(a0, a1), bob=(b0,
↪  b1))
actual3 = experiment3['probabilities']['stats'].T
axs[0][2].imshow(actual3, cmap='PiYG')

# shots = 1000
experiment4 = qt.classical.bell_singlet_full(1000, alice=(a0, a1), bob=(b0,
↪  b1))
actual4 = experiment4['probabilities']['stats'].T
axs[1][0].imshow(actual4, cmap='PiYG')

# shots = 5000
experiment5 = qt.classical.bell_singlet_full(5000, alice=(a0, a1), bob=(b0,
↪  b1))
actual5 = experiment5['probabilities']['stats'].T
axs[1][1].imshow(actual5, cmap='PiYG')

# shots = 10000
experiment6 = qt.classical.bell_singlet_full(10000, alice=(a0, a1), bob=(b0,
↪  b1))
actual6 = experiment6['probabilities']['stats'].T
axs[1][2].imshow(actual6, cmap='PiYG')

# shots = 20000
experiment7 = qt.classical.bell_singlet_full(20000, alice=(a0, a1), bob=(b0,
↪  b1))
actual7 = experiment7['probabilities']['stats'].T
axs[2][0].imshow(actual7, cmap='PiYG')

# shots = 50000
```

```
    experiment8 = qt.classical.bell_singlet_full(50000, alice=(a0, a1), bob=(b0,
    ↪  b1))
    actual8 = experiment8['probabilities']['stats'].T
    axs[2][1].imshow(actual8, cmap='PiYG')

    # shots = 100000
    experiment9 = qt.classical.bell_singlet_full(100000, alice=(a0, a1), bob=(b0,
    ↪  b1))
    actual9 = experiment9['probabilities']['stats'].T
    axs[2][2].imshow(actual9, cmap='PiYG')

    fig.tight_layout()
    plt.show()
    return None
```

Appendix B Simulator and Quantum Computer benchmarking

The prepare-and-measure scenario selected for the comparison is as follows

$$\begin{aligned} |\Psi\rangle &= \frac{3+i\sqrt{3}}{4}|0\rangle - \frac{1}{2}|1\rangle \\ \mathbb{P}_4 &= \{\frac{1}{2}|0\rangle\langle 0|, \frac{1}{2}|1\rangle\langle 1|, \frac{1}{2}|+\rangle\langle +|, \frac{1}{2}|-\rangle\langle -|\} \end{aligned} \tag{35}$$

The achievable performances when simulating such scenario with IBM Quantum simulators and computers can be found in the tables and figures below.

Processor	Measure			
	00	01	10	11
Aer Simulator	37	15	7	41
Nairobi	40	14	3	43
Perth	35	26	10	29
Oslo	40	8	4	48
Jakarta	35	16	11	38
Lagos	38	10	6	46

Table 6: Measure counts for a quantum simulator and different IBM Quantum computers: 100 shots.

Processor	Measure			
	00	01	10	11
Aer Simulator	182	54	30	234
Nairobi	183	59	67	191
Perth	200	89	67	144
Oslo	189	52	33	226
Jakarta	187	60	47	206
Lagos	164	59	43	234

Table 7: Measure counts for a quantum simulator and different IBM Quantum computers: 500 shots.

Processor	Measure			
	00	01	10	11
Aer Simulator	362	117	67	454
Nairobi	335	162	112	391
Perth	337	185	121	357
Oslo	379	100	52	469
Jakarta	378	160	76	386
Lagos	366	113	60	461

Table 8: Measure counts for a quantum simulator and different IBM Quantum computers: 1000 shots.

Processor	Measure			
	00	01	10	11
Aer Simulator	1484	513	268	1735
Nairobi	1417	453	392	1738
Perth	1534	656	391	1419
Oslo	1455	418	266	1861
Jakarta	1498	547	371	1584
Lagos	1378	424	258	1940

Table 9: Measure counts for a quantum simulator and different IBM Quantum computers: 4000 shots.

Processor	Measure			
	00	01	10	11
Aer Simulator	3772	1223	623	4382
Nairobi	3782	1724	900	3594
Perth	3703	1867	1177	3253
Oslo	3583	1083	702	4632
Jakarta	3829	1427	880	3864
Lagos	3570	1133	625	4672

Table 10: Measure counts for a quantum simulator and different IBM Quantum computers: 10000 shots.

Processor	Measure			
	00	01	10	11
Aer Simulator	7560	2409	1164	8867
Nairobi	6904	2294	1940	8862
Perth	6556	3034	2252	8158
Oslo	7563	2275	1274	8888
Jakarta	7325	2970	1971	7734
Lagos	7315	2594	1405	8686

Table 11: Measure counts for a quantum simulator and different IBM Quantum computers: 20000 shots.

Processor	Measure			
	00	01	10	11
Aer Simulator	0.37	0.15	0.07	0.41
Nairobi	0.4	0.14	0.03	0.43
Perth	0.35	0.26	0.1	0.29
Oslo	0.4	0.08	0.04	0.48
Jakarta	0.35	0.16	0.11	0.38
Lagos	0.38	0.1	0.06	0.46

Table 12: Measure probabilities for a quantum simulator and different IBM Quantum computers: 100 shots.

Processor	Measure			
	00	01	10	11
Aer Simulator	0.364	0.108	0.06	0.468
Nairobi	0.366	0.118	0.134	0.382
Perth	0.4	0.178	0.134	0.288
Oslo	0.378	0.104	0.066	0.452
Jakarta	0.374	0.12	0.094	0.412
Lagos	0.328	0.118	0.086	0.468

Table 13: Measure probabilities for a quantum simulator and different IBM Quantum computers: 500 shots.

Processor	Measure			
	00	01	10	11
Aer Simulator	0.362	0.117	0.067	0.454
Nairobi	0.335	0.162	0.112	0.391
Perth	0.337	0.185	0.121	0.357
Oslo	0.379	0.1	0.052	0.469
Jakarta	0.378	0.16	0.076	0.386
Lagos	0.366	0.113	0.06	0.461

Table 14: Measure probabilities for a quantum simulator and different IBM Quantum computers: 1000 shots.

Processor	Measure			
	00	01	10	11
Aer Simulator	0.371	0.12825	0.067	0.43375
Nairobi	0.35425	0.11325	0.098	0.4345
Perth	0.3835	0.164	0.09775	0.35475
Oslo	0.36375	0.1045	0.0665	0.46525
Jakarta	0.3745	0.13675	0.09275	0.396
Lagos	0.3445	0.106	0.0645	0.485

Table 15: Measure probabilities for a quantum simulator and different IBM Quantum computers: 4000 shots.

Processor	Measure			
	00	01	10	11
Aer Simulator	0.3772	0.1223	0.0623	0.4382
Nairobi	0.3782	0.1724	0.09	0.3594
Perth	0.3703	0.1867	0.1177	0.3253
Oslo	0.3583	0.1083	0.0702	0.4632
Jakarta	0.3829	0.1427	0.088	0.3864
Lagos	0.357	0.1133	0.0625	0.4672

Table 16: Measure probabilities for a quantum simulator and different IBM Quantum computers: 10000 shots.

Processor	Measure			
	00	01	10	11
Aer Simulator	0.378	0.12045	0.0582	0.44335
Nairobi	0.3452	0.1147	0.097	0.4431
Perth	0.3278	0.1517	0.1126	0.4079
Oslo	0.37815	0.11375	0.0637	0.4444
Jakarta	0.36625	0.1485	0.09855	0.3867
Lagos	0.36575	0.1297	0.07025	0.4343

Table 17: Measure probabilities for a quantum simulator and different IBM Quantum computers: 20000 shots.

Processor	Measure			
	00	01	10	11
Nairobi	0.03	0.01	0.04	0.02
Perth	0.02	0.11	0.03	0.12
Oslo	0.03	0.07	0.03	0.07
Jakarta	0.02	0.01	0.04	0.03
Lagos	0.01	0.05	0.01	0.05

Table 18: Error between results obtained with IBM Quantum computers and theoretical probabilities: 100 shots.

Processor	Measure			
	00	01	10	11
Nairobi	0.002	0.01	0.074	0.086
Perth	0.036	0.07	0.074	0.18
Oslo	0.014	0.004	0.006	0.016
Jakarta	0.01	0.012	0.034	0.056
Lagos	0.036	0.01	0.026	0

Table 19: Error between results obtained with IBM Quantum computers and theoretical probabilities: 500 shots.

Processor	Measure			
	00	01	10	11
Nairobi	0.027	0.045	0.045	0.063
Perth	0.025	0.068	0.054	0.097
Oslo	0.017	0.017	0.015	0.015
Jakarta	0.016	0.043	0.009	0.068
Lagos	0.004	0.004	0.007	0.007

Table 20: Error between results obtained with IBM Quantum computers and theoretical probabilities: 1000 shots.

Processor	Measure			
	00	01	10	11
Nairobi	0.01675	0.015	0.031	0.00075
Perth	0.0125	0.03575	0.03075	0.079
Oslo	0.00725	0.02375	0.0005	0.0315
Jakarta	0.0035	0.0085	0.02575	0.03775
Lagos	0.0265	0.02225	0.0025	0.05125

Table 21: Error between results obtained with IBM Quantum computers and theoretical probabilities: 4000 shots.

Processor	Measure			
	00	01	10	11
Nairobi	0.001	0.0501	0.0277	0.0788
Perth	0.0069	0.0644	0.0554	0.1129
Oslo	0.0189	0.014	0.0079	0.025
Jakarta	0.0057	0.0204	0.0257	0.0518
Lagos	0.0202	0.009	0.0002	0.029

Table 22: Error between results obtained with IBM Quantum computers and theoretical probabilities: 10000 shots.

Processor	Measure			
	00	01	10	11
Nairobi	0.0328	0.00575	0.0388	0.00025
Perth	0.0502	0.03125	0.0544	0.03545
Oslo	0.000	0.007	0.006	0.001
Jakarta	0.01175	0.02805	0.04035	0.05665
Lagos	0.01225	0.00925	0.01205	0.00905

Table 23: Error between results obtained with IBM Quantum computers and theoretical probabilities: 20000 shots.

Processor	Shots					
	100	500	10^3	$4 \cdot 10^3$	10^4	$2 \cdot 10^4$
Nairobi	0.1	0.172	0.18	0.0635	0.1576	0.0776
Perth	0.28	0.36	0.244	0.158	0.2396	0.1713
Oslo	0.2	0.04	0.064	0.063	0.0658	0.0134
Jakarta	0.1	0.112	0.136	0.0755	0.1036	0.1368
Lagos	0.12	0.072	0.022	0.1025	0.0584	0.0426

Table 24: Total error between results obtained with IBM Quantum computers and theoretical probabilities.

Results	Average Error
Perth	0.24215
Nairobi	0.12512
Jakarta	0.11065
Oslo	0.07437
Lagos	0.06958

Table 25: Average error between results obtained with IBM Quantum computers and theoretical probabilities.

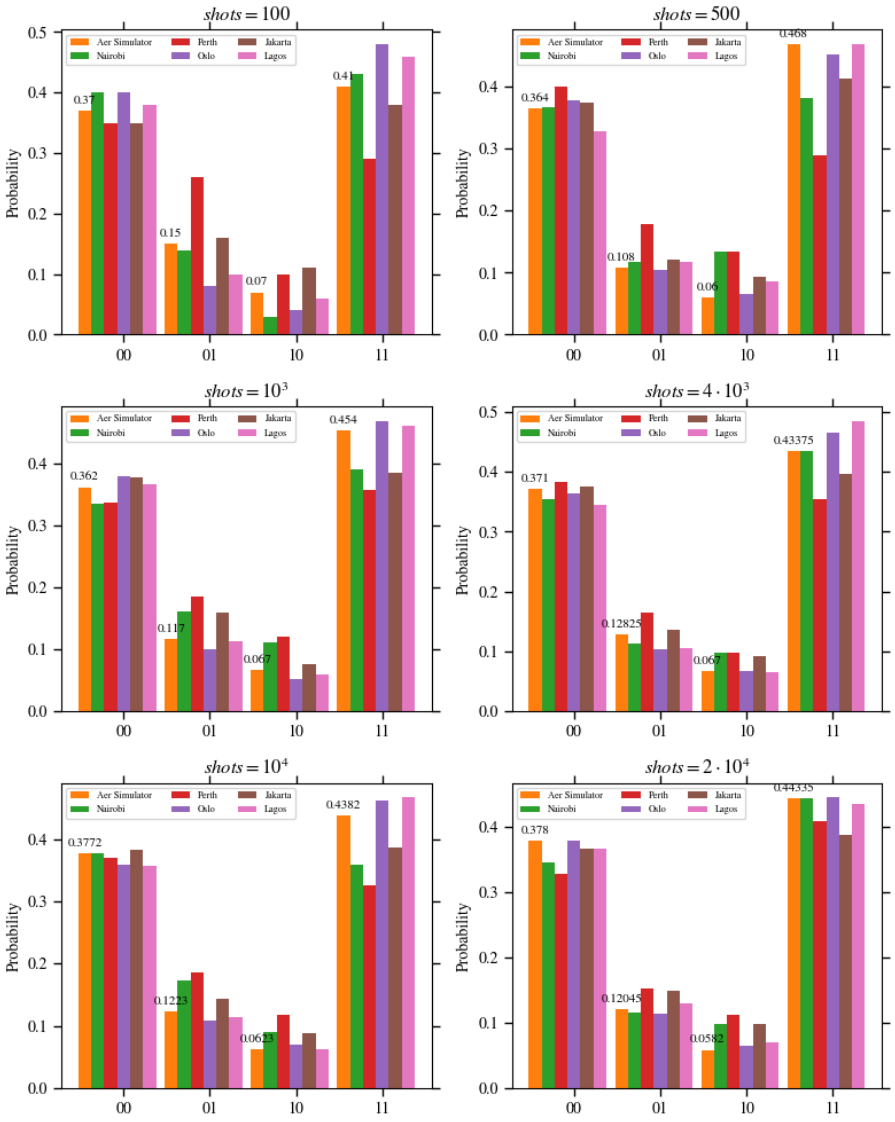

Figure 13: Probabilities obtained in a quantum simulator and various IBM Quantum computers across a range of shots.

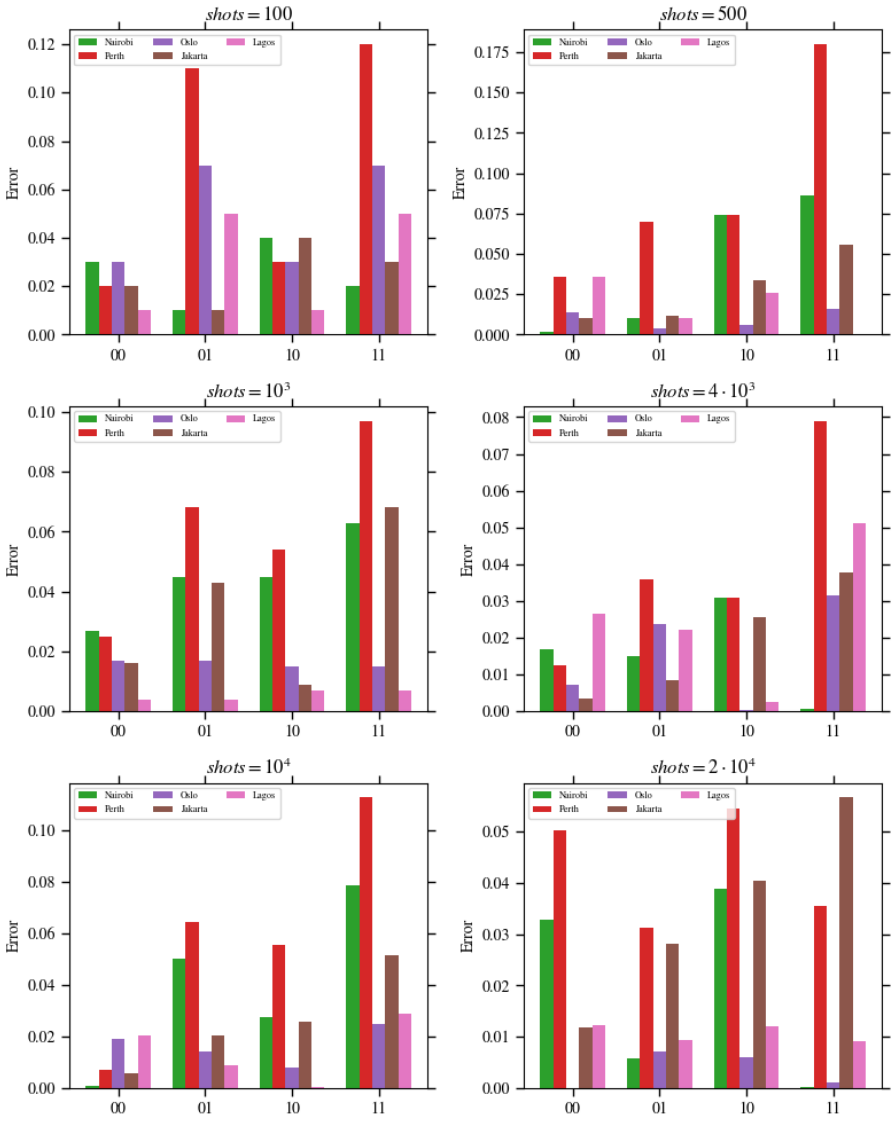

Figure 14: Errors between theoretical probabilities and the results obtained from various IBM Quantum computers compared across a range of shots.

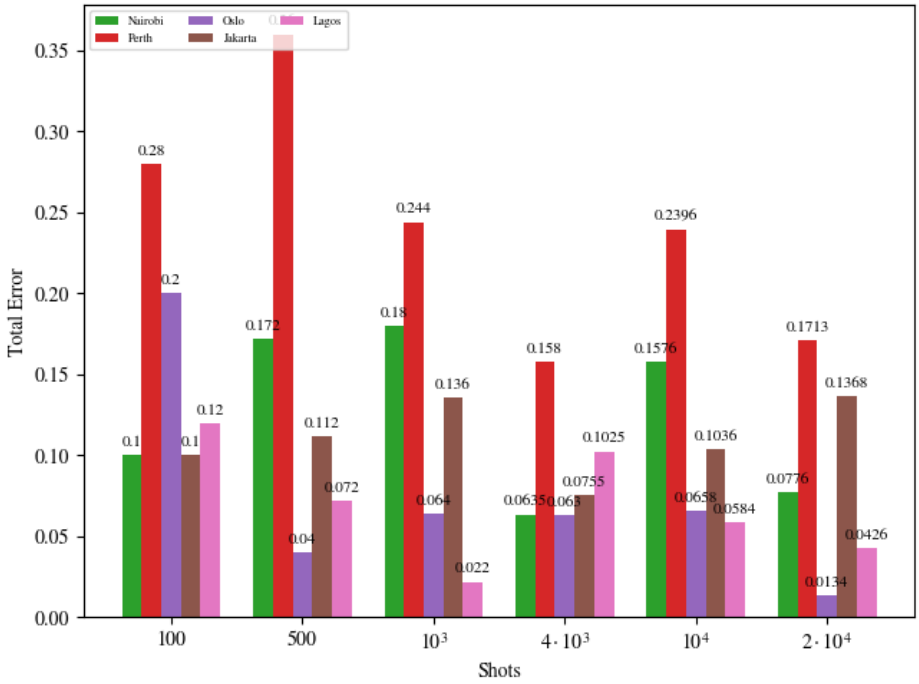

Figure 15: Accumulated errors between theoretical probabilities and the results obtained from various IBM Quantum computers compared across a range of shots.

The figures and tables presented above demonstrate that in the prepare-and-measure scenario, quantum programs running on IBM quantum computers with the highest number of shots closely approximate to quantum simulators. These results suggest that although today's noisy quantum computers are still subject to reasonable error rates, improvements can be expected in the medium term.

It also shows that the quantum computer with the fewest errors, Lagos, is the one that obtains the best overall results in all runs. It will be very interesting to explore this setup and measurement scenario with a higher number of shots in the future.