Simulation of Quantum Circuits on a Laptop

Pablo Rafael Miró Ruiz

4th Year Project Report
Computer Science and Physics
School of Informatics
University of Edinburgh

2022

Abstract

The field of quantum computing is growing at an incredible rate due to the promise that quantum computers will execute tasks exponentially faster than a classical processor. In 2019, Google claimed that their Sycamore quantum processor takes several minutes to perform a sampling task, while they estimated that the most powerful supercomputer would take thousands of years to produce similar results. This project develops an algorithm that can be used to generate samples from quantum circuits in a classical computer. The presented algorithm is based on the analysis of Boolean functions, which allows to limit the order of the Fourier coefficients used and to make the simulation faster. In a reasonable time frame, a standard laptop can execute the implemented algorithm to generate thousands of samples from random quantum circuits with less than 20 qubits. The implementation of the algorithm allows it to be modified and adapted for high-performance computing and to have the correlators computed more efficiently. Three different approaches are discussed in this paper, with the one that performed the best having a learning phase. Nevertheless, due to the exponential time of such a learning procedure, it is recommended to use the hybrid approach when the size of the circuit increases.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Pablo Rafael Miró Ruiz*)

Acknowledgements

I would like to thank my supervisor, Dr. Raúl García-Patrón, for his wholehearted encouragement, for his continued support, flexibility and understanding in all the meetings we have had, and specially for proposing a project that has made me more interested in the topic of quantum computation. I want to thank my personal tutor, Dr. Pavlos Andreadis, for having given me his sincere opinion on every topic I raised, for making me be clearer when I want to expose my issues, and for being super responsive any time I have wanted to talk to him.

Furthemore, I want to thank my parents, Rafael and Teresa, for providing me with moral and financial support during these whole four years, and for being truly interested in what I study; to my grandparents, Rafael, Julia and Teresa for asking me all the time when I am coming back to Valencia, and to my uncle-grandfather, Rafael, who still asked about me before passing away.

Lastly, I want to thank Laura for listening to my thoughts and complaints, even when I get very annoying. Thanks to Orges, Alex, Samuel, Paula, Gabrijel, Marina, Despina, Ferrán, Peto for their continuous love and support, specially during this last fourth year. A peculiar mention to the baristas in Edinburgh, namely those who work in cafés close to the library, for their kindness and providing me with caffeine to support my studies.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

The field of quantum computing is developing at an impressive pace, with governments and private investors worldwide funding quantum research. IBM, Google, Amazon, and Microsoft are only a few companies heavily invested in developing large-scale quantum computing hardware and software. There are many well-based reasons for this accelerated development: quantum computers are ideally suited for solving complex optimisation tasks and performing fast searches of unsorted data [7, 51]. This could be relevant for many applications, from sorting climate, health or financial data, to optimising supply chain logistics, workforce management or traffic flow. They could also help in complex simulations that might guide us to drug discoveries and improvements in medicine [53, 52, 40].

Furthermore, existing protocols such as quantum key distribution [15] show how much can cryptography benefit from quantum technologies. Not only cryptography but our communication systems will change through the introduction of quantum channels [11, 10, 6]. They will lay the groundwork for a potential quantum security-based global communication network [31]. However, while small-scale quantum computers are available today, scaling this technology requires advanced techniques to deal with the errors they suffer [38].

1.2 The big picture

My dissertation takes part in a more extensive project from my supervisor, Raúl García-Patrón, which deals with the complexity of simulating quantum circuits. The number of qubits n and depth of the circuit m are usually used to determine the exponential hardness of simulating a quantum circuit. Nevertheless, current quantum devices are not fault-tolerant, they are subject to noise that affects their output. For example, Google claimed in 2019 that their quantum processor performed a task that would take thousands of years to the most powerful supercomputer [23]. This is a bold statement to make. We might be able to reduce their classical estimation by designing simulations

more intelligently.

It seems that such noise could be included in the classical algorithms to make the simulations faster. It would also be possible to obtain the same results as the experiments performed on quantum devices. The insight is that deeper circuits might be easier to simulate because of the accumulation of errors (quantum decoherence, errors in the physical construction of the quantum circuit...). First by obtaining better exponents in their time complexity, then by a transition to polynomial-time at some given depth [42].

The wider project will use Fourier decomposition of Boolean functions for simulating circuits of size similar to Google's. It intends for the classical simulation to provide outputs similar to current quantum computers. To achieve this, we need to understand that the output from the quantum circuit comes from a probability distribution, even if it is unknown to us.

Indeed, with the Fourier coefficients of such distribution, one can generate many outcomes from the quantum circuit, also known as **sampling**. The project will focus on developing a sampling algorithm that uses Fourier coefficients, obtained with state-of-the-art techniques such as tensor network (TN) contractions. This will be implemented to limit the Fourier coefficients used so that the classical simulation could obtain a result similar to the quantum one, and faster.

1.3 My project

My project focuses on developing a sampling algorithm for quantum circuits. To sample a quantum circuit of size and complexity similar to the ones Google used in their experiment, state-of-the-art techniques such as tensor network (TN) contractions are needed alongside a modification for high-performance computing. This, however, will be completed in Raúl's project, and is out of the scope for complexity and timing constraints.

In this report quantum circuits with a smaller number of qubits will be simulated, and Fourier coefficients will be computed from the full wave function (and not by using TN). This is by no means an efficient way to obtain the real probability distribution of the quantum device, but it will be enough to test the sampling algorithm. The fact that circuits with a smaller number of qubits are considered does not modify the algorithm developed.

1.4 Completion of the project

In this report, I achieved the following goals:

- Gained a deep understanding of Google's quantum advantage experiment in 2019, and of recent literature to synthesise research conducted so far to discuss the approaches taken for demonstrating quantum advantage.
- Acquired knowledge on the analysis of Boolean functions and derived the equations that used at the core of the sampling algorithm.

- Familiarised with Google Cirq, a Python library for quantum computing, and dug into its source code to find a way of generating quantum circuits similar to those used in Google's experiment.
- Carried out the pre-processing needed to generate samples and obtained the Fourier coefficients using a brute-force approach.
- Implemented three different approaches for the sampling algorithm and added a learning phase that vastly improved the algorithm's performance.
- Realised several experiments to obtain visible results to show how precise the values obtained are and benchmarked the running time of the different approaches for sampling.
- Tested the correctness of the algorithm and fixed potential bugs that did not generate the correct outputs.

The implementation of the algorithms and some experiments can be found in this github repository. There is a README.md file that explains how to install the dependencies with conda. It also shows which file contains the experiments to try the different sampling approaches, and produce different plots.

1.5 Report structure

Chapter 2 introduces the concept of "quantum advantage" and the difficulty of sampling random quantum circuits with a classical computer. Google's 2019 experiment is thoroughly explained, detailing everything from their quantum circuits to performing the benchmarking. Recent literature is discussed to put into context Google's claim of quantum advantage and its implications.

Chapter 3 develops the mathematical formulation of Boolean Functions. Moreover, it presents the different approaches to generating samples from a quantum circuit; and how we can restrict the Fourier spectrum of Boolean functions to obtain results similar to Google's in a classical device. The marginal probabilities and the equation implemented at the core of the sampling algorithm are derived.

Chapter 4 describes the implementation of the sampling algorithm. Specifically, it describes the three different approaches discussed in Chapter 3. In doing so, it discusses the pre-processing of the algorithm, the programming language and the different libraries used.

Chapter 5 details how testing was performed to ensure the outcomes from the sampling algorithm are not wrong.

Chapter 6 motivates, describes, and presents the outcome of a set of experiments on the algorithms. The goal is to analyse how the restriction of Fourier coefficients can help us obtain similar results to those achieved by a quantum computer.

Chapter 7 draws conclusions from the results of previous experiments, provides suggestions for future work, and summarises the accomplishments of the project.

Chapter 2

Background and Related Work

2.1 Quantum Computing

2.1.1 Context

A classical computer is made up of logic gates. These are switches that receive electric currents and let current flow depending on the specific input currents. We interpret the flow and lack of flow of an electric current with the binary values 1 or 0, and the operations a computer can perform are defined through Boolean Algebra.

Quantum computing is not based on binary operations, but it generalises classical physics by harnessing the power of quantum mechanics. This is a probabilistic theory developed in the 20th century thanks to the work done by Planck, Einstein, Bohr, Heisenberg, Schrödinger. . . that provides a precise extension to Newtonian mechanics in the microscopic world. In quantum computing, the analogue of the bit has been identified as the *qubit*; the analogue of a logic gate is any linear operator that can act on the system (also referred to as *quantum gates*) [39]. From the postulates of quantum mechanics, all the information about the physical system and its properties is held by the state vector $|\Psi, t\rangle$ [1]. A quantum computer would modify and evolve the system by manipulating its state vector so that it would extract helpful information for computing purposes [9].

This new field started in 1980 when Paul Benioff proposed a quantum mechanical model for standard Turing machines [3]. Soon after this Richard Feynman showed that it its theoretically possible to create an adiabatic (i.e. heat does not enter or leave the system) reversible computer. In 1982 he argued that an exponential amount of resources were required to simulate quantum mechanical systems, an appreciation that was achieved a bit earlier, in 1980, by Yu Manin in the Soviet Union [4]. It is important to add that not all quantum mechanical systems are hard to simulate; in fact, we have exact solutions of systems such as the hydrogen atom, the Morse potential or the quantum harmonic oscillator. A general quantum system, especially with low temperatures (because of our ability to control and manipulate systems in effective low-temperature settings) [18] or from many-body physics, appears to be hard to simulate [36]. Feynman added the idea of using a quantum computer to solve such computational problems [5].

It has since been realised that the power of quantum computing will have a wide range of applications. Our communication systems will be heavily influenced through protocols that use quantum teleportation (e.g., quantum secure direct communication or remote state preparation [11, 10]) and superdense coding [6]. Cryptography will be greatly influenced from quantum algorithms and protocols such as quantum key distribution. In fact, in 1994 Peter Shor developed a polynomial-time algorithm for prime–factorisation and computing discrete logarithms using [8], providing a dramatic decrease in the required computational time compared with its classical counterparts. A perfect quantum computer would be able to decipher our encryption schemes, such as the RSA encryption system or the Diffie-Hellman key exchange protocol. Nevertheless, recent studies show that we are far from capable to perform these tasks [44] with the available quantum error-correction techniques and hardware.

2.1.2 On “Quantum Supremacy"

The idea of *quantum supremacy* or *quantum advantage* was coined by John Preskill [1] to indicate “the moment that a quantum computer gains the ability to perform a task that a classical computer never could”. This can be demonstrated through problems that do not possess any practical application. Due to the technical difficulties of building a quantum computer, and because qubits are more prone to error than classical bits, researchers are focusing on problems such as sampling random quantum circuits which seem easier to implement in current quantum hardware. Indeed, Google claimed quantum advantage this way, as discussed in the next section.

However, quantum advantage cannot be perceived as a one-shot experimental proof. Because of the improvements in classical simulations, we will require a long-term competition between classical and quantum devices to show such an advantage with random quantum circuits. Researchers are trying other experiments, such as *boson sampling*, proposed by Aaronson and Arkhipov [14]. The hardness of this problem is based on the difficulty of computing Permanents of matrices. Moreover, the Polynomial Hierarchy would collapse at the third level because of an efficient algorithm for boson sampling (or for an approximation, as discussed by García-Patrón and Leverrier in [17]).This also makes boson sampling a strong candidate for demonstrating quantum advantage. A team in China has already claimed quantum supremacy using Gaussian boson sampling [37, 45, 30]. These claims have already been challenged more than once [41, 49] with improved simulation algorithms. Moreover, Gil Kalai [33, 32] argues that it will take several months to see whether we can take such a claim of quantum advantage seriously [2].

[1] There is a very thought-provoking article in `Quanta Magazine` [28] where he comments on the controversy around the naming. I will use the term *quantum advantage* from this report’s section onwards.

[2] He refers to a 2014 paper of his [16] that could be used to test a classical algorithm that “spoofs” the results of [37] to achieve similar results.

2.2 Google's Experiment

2.2.1 Overview

In October 2019, Google published an article titled *Quantum supremacy using a programmable superconducting processor*. In their experiment, they used their Sycamore processor to create quantum states on 53 qubits (hence corresponding to a Hilbert space of dimension 2^{53}) and generate samples from random quantum circuits. According to their results, it took their processor 200 seconds to generate a million samples of a quantum circuit, while –according to their benchmarks– it would take around 10,000 years to perform the same computation on the Summit supercomputer at Oak Ridge National Laboratories [23].

It is essential to highlight the differences in sampling methods between quantum and classical computers. Quantum mechanics defines the collapse of the wavefunction after a measurement [1]. As a consequence, a quantum computer runs the circuit for and records its output for every sample. On the other hand, a classical computer does not collapse the state of a system in a simulation, and there exist different approaches to sampling. One method would be to simulate the state vector, obtain the probability distribution and generate samples with such distribution. From Born's rule, a key postulate of quantum mechanics, the probability of measuring the quantum system in a given state is $p(i) = |\langle i|\Psi, t\rangle|^2$.

For sampling, since the generated events are represented with binary strings, or *bitstrings*, $x \in \{0,1\}^n$, the probability p is a real-valued Boolean function. $p : \{0,1\}^n \to \mathbb{R}$.

2.2.2 Quantum Random Circuits

It was crucial to generate quantum circuits which would be hard for a classical computer to simulate. If the circuits had observable patterns, a classical algorithm could be developed so that it would exploit them to run more efficiently. Moreover, recall the limitations of current quantum processors: generated circuits cannot have a sufficiently large depth. This is due to the logic gates not being perfectly implemented and because the quantum states are sensitive to errors, which could make the output completely random.

Consequently, Google generated **random** quantum circuits where the qubits were entangled and without any observable pattern. This was the foremost candidate for demonstrating quantum advantage, the so-called Random Circuit Sampling (RCS) problem [21]. This states that for any classical computer, it is hard to produce samples from a distribution that is close to the distribution of a local quantum circuit whose local gates are randomly and independently drawn uniformly from the space of all gates.

The quantum circuits, also called *Sycamore circuits*, had $m = 20$ gate cycles of single-qubit and two-qubit logical operations to obtain entanglement between all the qubits in a complex manner. In figure 2.1, they follow an intractable sequence for a classical computer according to their results (repeat ABCDCDAB). In their paper, they also mention "verification circuits" using the ordering EFGHEFGH. As Craig Gidney discusses in this blog post, they were intended to be supremacy circuits, but after collecting data

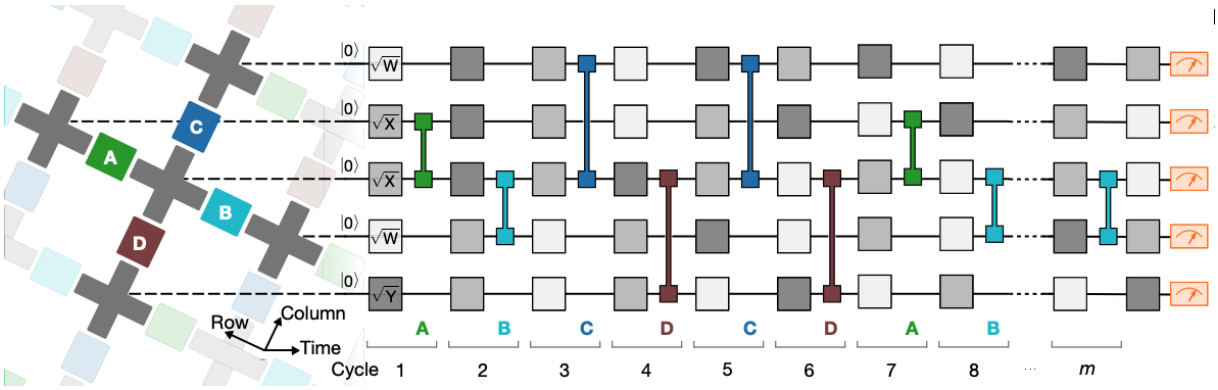

Figure 2.1: **Example quantum circuit instance used in Google's experiment.** Every cycle includes a layer each of single- and two-qubit gates. The single-qubit gates are chosen randomly from $\{\sqrt{X}, \sqrt{Y}, \sqrt{W}\}$, where X and Y correspond to the Pauli-X and Pauli-Y operators, $W = (X+Y)/\sqrt{2}$ and gates do not repeat sequentially. They were chosen for being $\pi/2$ rotations around a specific axes lying on the equator of the Bloch sphere. The sequence of two-qubit gates (also called "couplers") is chosen according to a tiling pattern, coupling each qubit sequentially to its four nearest-neighbour qubits. The couplers are divided into four subsets (ABCD), each of which is executed simultaneously across the entire array corresponding to shaded colours. Figure 3 from [23]

they had to change the ordering which proved to be very important for making the simulation harder.

The $X^{1/2}$ and $Y^{1/2}$ gates belong to the Clifford group, while $W^{1/2}$ does not. This was probably chosen to avoid these circuits being efficiently simulated given the Gottesman-Knill theorem [13]. On the other hand, the two-qubit gate used for quantum advantage was the iSWAP. In the Supplementary Material they describe the "fSim" group (short for fermionic simulation) to which iSWAP belongs, and describe other entangling gates [24, pp 15–16].

2.2.3 XEB Theory

The method used to verify their quantum device worked adequately is known as *cross-entropy benchmarking*. This method analyses the frequency of the possible quantum states measured experimentally, compared to the ideal probability simulated on a classical computer. For a given circuit, they generate a large number of samples $\{x\}$ and compute the *linear cross-entropy benchmarking fidelity*, which is the **mean of the simulated probabilities of the bitstrings they measured**:

$$\mathcal{F}_{\text{XEB}} = 2^n \langle P(x) \rangle_x - 1, \tag{2.1}$$

where n is the number of qubits, $p(x)$ is the probability of bitstring x computed for the ideal quantum circuit, and the average is over the observed bitstrings. $\mathcal{F}_{\text{XEB}}$ is proportional to the mean of the experimentally measured probabilities, also known as High Output Generation (HOG), and intuitively it is correlated with the frequency of sampling high-probability bitstrings. Letting $p(x)$ be the distribution of the random

quantum circuit, and $q(x)$ the one corresponding to the classical simulation of the noisy quantum computer, HOG is defined as:

$$\mathrm{HOG}(p,q) = \mathbb{E}_{q(x)}[p(x)] = \sum_x q(x)p(x). \tag{2.2}$$

We can express the fidelity in terms of the HOG as:

$$\mathcal{F}_{\mathrm{XEB}} = 2^n \mathrm{HOG} - 1. \tag{2.3}$$

For the specific case of random quantum circuits, $\mathcal{F}_{\mathrm{XEB}}$ is **bounded between 0 and 1**. A value of 0 is obtained when the outcomes sampled are noisy so that they follow the uniform distribution and $\mathbb{E}[P(x_i)] = 1/2^n$. On the other hand, if the random quantum circuits include little to zero noise, they will follow the *Porter-Thomas distribution*. This distribution gives $\mathbb{E}[P(x_i)] = 2/2^n$ and a corresponding $\mathcal{F}_{\mathrm{XEB}} = 1$.

In quantum information science, **fidelity** is a term which denotes how close one quantum state is from another. Given a general density operator ρ and a pure state $|\psi\rangle$, it is defined as:

$$F(\rho,\sigma) = |\langle\psi|\rho|\psi\rangle|^2. \tag{2.4}$$

It is a surprising result that, for the specific case of random quantum circuits, XEB is proven to be equivalent to the fidelity:

$$\mathcal{F}_{\mathrm{XEB}} = 2^n \mathbb{E}_{q(x)}[p(x)] - 1 = |\langle\psi|\rho|\psi\rangle|^2 = F(\rho,|\psi\rangle), \tag{2.5}$$

On account of $F \equiv \mathcal{F}_{\mathrm{XEB}}$ for the specific case of RQC, I will use the word fidelity to refer to the *linear cross-entropy benchmarking fidelity*, $\mathcal{F}_{\mathrm{XEB}}$.

2.2.4 Results and criticism

Google performed a large number of experiments to verify their benchmarking methods. They ran a series of experiments with three variations of the circuits that reduced their complexity so that it could be simulated by a classical computer in a reasonable amount of time. From Figure 2.2 **a**, they had:

- **path circuits**, where they reduced the number of two-qubit gates splitting the circuit into two independent patches of qubits.
- **elided circuits**, where a fraction of the initial two-qubit gates is removed.
- **full circuits**, which only modified the sequence ordering to EFGHEFGH.

For each data point they typically collected 5×10^6 samples over ten circuit instances which differed in the choices of single-qubit gates per cycle. The black line corresponds to the predicted $\mathcal{F}_{\mathrm{XEB}}$, and the close correspondence justifies that they used **elided circuits** to estimate the fidelity in the supremacy regime. In Figure 2.1 **b** Google shows their estimates for the $\mathcal{F}_{\mathrm{XEB}}$ in the quantum supremacy regime, where they use the circuits mentioned in section 2.2.2 and the intractable sequence from Figure 2.1.

They estimated the computational cost in a classical supercomputer by using a hybrid Schrödinger–Feynman algorithm [22]. They argued that such algorithm was the best classical method for circuits with $n > 43$ qubits. This is because it is more memory

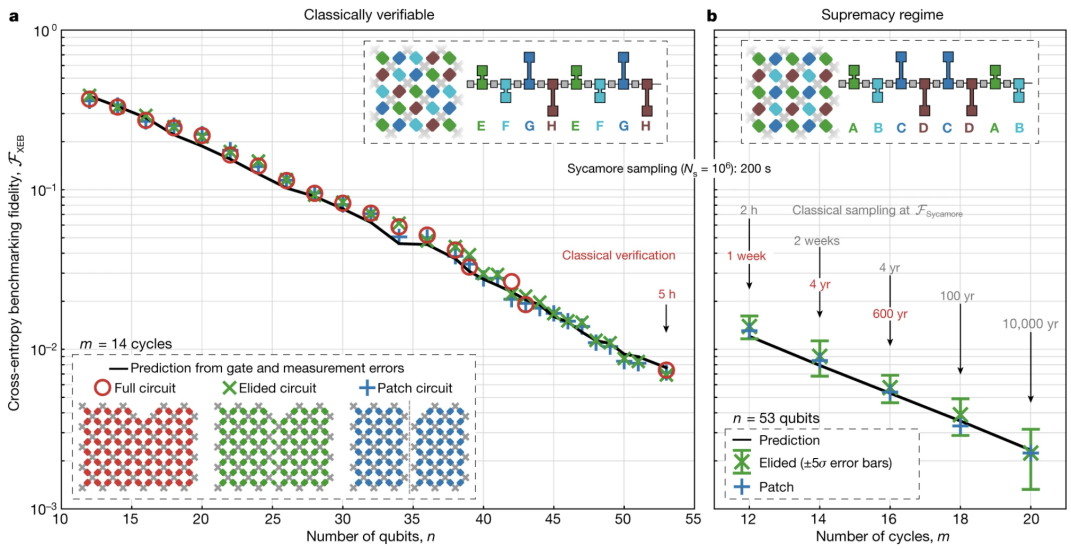

Figure 2.2: **a Google's verification of benchmarking methods** using circuits that can be verified by a classical computer. They used three variations of the circuits to estimate the fidelity in the supremacy regime. **b Estimating XEB in the quantum supremacy regime**. With $m = 20$ cycles and $n = 53$ qubits obtaining a million samples on the quantum processor takes 200 seconds with a $\mathcal{F}_{\text{XEB}} = (2.24 \pm 0.21) \times 10^{-3}$. In contrast a classical sampling with similar fidelity would take 10,000 years on a million cores, and verifying the fidelity would take millions of years. Figure 4 from [23].

efficient than the Schrödinger algorithm which simulates the evolution of the full quantum state, thus requiring a very large RAM to store it. As they show in 2.1 **b**, they estimated it would take the `Summit` supercomputer 10,000 years to obtain a million samples from the 53 qubit, 20 cycle circuits that only takes 200 seconds in their quantum processor. This argument does not seem very convincing because of the small value of $\mathcal{F}_{\text{XEB}}$ obtained. Google's Sycamore processor is not *fault-tolerant* and it also suffers from errors that can be included in classical algorithms to **spoof** a similar fidelity.

In fact, their claim was very soon challenged by IBM [27], arguing that on the Summit supercomputer such circuits can be simulated with the same fidelity to arbitrary depth in a matter of days [26]. They proposed a method which combines in-memory methods with solid-state disk, or more generally secondary storage for storing the large quantum states. Moreover, they model the quantum circuits as tensor networks and use a simulation strategy which relies on *contraction deferral* and *tensor slicing*. Their idea is to partition the tensor network in sub-circuits, with smaller qubits than the original circuit, so that they can perform the "Shcrödinger approach" [19].

2.2.5 Further Experiments

Another team in China presented a more powerful, two-dimensional programmable superconducting quantum processor, *Zuchongzhi*, made up of 66 functional qubits in a coupling architecture. They carried out a similar experiment to Google using random quantum circuit sampling to evaluate the power of their processor. They used circuits

up to 56 qubits and 20 cycles for benchmarking, and measured that for their most complex circuit it took *Zuchongzhi* 1.2 hours to execute. Their estimation for running such computational task is of at least 8 years in the most powerful supercomputer [50]. They recently published their results from a harder experiment using an upgrade to their processor, *Zuchongzhi* 2.1. They sampled random circuits of 60 qubits and 24 cycles, significantly increasing the estimation of a classical simulation to 4.8×10^4 years [54].

Nonetheless, even though these experiments have a larger size, they obtained a lower fidelity value from 7.0×10^7 bitstrings, $\mathcal{F}_{\text{XEB}} = 0.0366\%$. Therefore, it is not obvious if this increase in the circuit's size indicates whether the classical sampling is harder. On the other hand, it is also non trivial to determine the cost of the tensor network contraction that we require to obtain $p(x)$ of the ideal circuit. Even so, we know that the computational state space does increase exponentially, and it could be the case that we achieve a similar value of $\mathcal{F}_{\text{XEB}}$ by using tensor networks and restricting the number of Fourier coefficients used to generate samples.

2.3 Classically simulating quantum circuits

2.3.1 Ideal circuit simulation

Perfect quantum computers are unarguably challenging to simulate because the quantum state-space grows exponentially with the number of qubits N, but also with the depth D of the circuit, i.e. how many layers of gates the quantum circuit has. In current research, scientists come up with algorithms that use tensor network contractions to perform classical simulations that compete with quantum devices.

Tensor network contraction [12] is a mathematical tool that has been undergoing rapid developments for its use in statistical physics, atomic physics, condensed matter physics, quantum information theory... We can think of tensors as multi-linear operators which generalise vectors and matrices, with a tensor network being a collection of possibly connected tensors. A quantum state can be expressed with tensors, and these can be split up iteratively until each corresponds to a single qubit by means of *Schmidt decomposition* [35]. The benefit of using this technique is that we can apply unitaries and measurements fast on states with efficient networks (depending on their rank [25]).

Last year, in 2021, Feng Pan and Pan Zhang proposed a more general tensor network method for simulating quantum circuits; and for the Sycamore circuits (with 53 qubits and 20 cycles) they achieved a linear cross-entropy benchmarking fidelity of $\mathcal{F}_{\text{XEB}} = 0.739$ from one million *correlated* bitstrings [48]. This, however, is different to Google's experiment, where the samples have no correlation. In November 2021, alongside Keyang Chen, they proposed another method using tensor networks for not only passing the XEB test (obtaining a similar or higher value than Google), but also obtaining *uncorrelated* samples, as in Google's 2019 experiment. This time, for the Sycamore circuits, their classical algorithm took 15 hours and achieved an approximate fidelity of $\mathcal{F}_{\text{XEB}} \approx 0.0037$; and they expect that (if implemented efficiently on a modern supercomputer) it could run in a few dozens of seconds [47].

Furthermore, less than two weeks later, a different group developed a different algorithm

to implement in the new Sunway supercomputer, and they reduced the simulation sampling time of Sycamore circuits to 304 seconds [46].

2.3.2 Including noise in the simulation

The problem with these previous experiments on classical simulation do not exploit the fact that current quantum devices are noisy. *Real* quantum devices, those used in current experiments, suffer from quantum imperfections which could potentially make them easier to simulate. They are characterized by an exponentially decaying fidelity $\mathcal{F} \sim (1-\varepsilon)^{ND}$ with an error rate ε per operation is as small as $\approx 1\%$[38].

If the quantum device has many errors, this limits the degree of entanglement that can be achieved, and the output could become random noise: similar to the output of random coin flips. On the other hand, if the quantum device suffers from zero to few errors, it will be very hard to simulate when N and/or D are sufficiently large [38]. Therefore, there exists a transition which lets us cleverly exploit this quantum noise to run classical simulations faster. The true issue is how to clearly identify this transition in order to run more efficient simulations effectively.

2.4 Sampling from Fourier coefficients

One approach which can use noise for helping the classical simulations will be using Fourier coefficients, and this is what I will show in **chapter 4**. In the next section, I will first introduce the mathematics behind Fourier analysis of Boolean functions.

Sampling quantum circuits can be carried out with the Fourier spectrum of the probability distribution from the quantum circuit. As shown by Ashley Montanaro in [20], "in a quite general setting, if we can compute the marginals of an approximation p' to a probability distribution p, we can approximately sample from p". We can use Fourier analysis to obtain the marginal probabilities of the bitstring outcomes. This will be a **crucial step** in my sampling algorithm to generate outcomes from a quantum circuit.

The benefit of using Fourier coefficients is that one can limit how many are included in the computation, thus trading off the fidelity for a reduction in the computational complexity. This greatly depends on the noise model, but if we assume the higher order coefficients are concentrated, we could remove those that are more complicated to compute and still obtain a high value of $\mathcal{F}_{\text{XEB}}$.

Chapter 3

Analysis of Boolean Functions

3.1 Overview

The analysis of Boolean functions refers to studying Boolean functions $f : \{0,1\}^n \to \{0,1\}$ via their Fourier expansion and other analytic means. We are interested on knowing information about the probability distributions of random quantum circuits, which can be represented by the unitaries U acting on an initial quantum state $|\Psi\rangle$. From Born's rule, mentioned in the previous chapter, these belong to the more general case of *real-valued Boolean functions*, $p : \{0,1\}^n \to \mathbb{R}$. From a given bitstring x, these distributions $p_U(x)$ return the probability of it being an output of the circuit. The notation p is used since the functions of interest in this paper are probability distributions.

3.2 Fourier analysis in $\mathbb{Z}_2^n$

We will consider functions defined on the domain $\{0,1\}^n$. The Fourier expansion of a Boolean function can be viewed as its representation as a real multilinear polynomial, meaning that no variable occurs in a power of 2 or higher. Every real-valued Boolean function has its unique expansion as a multilinear polynomial, given by:

$$p(x) = \sum_{s \in \{0,1\}^n} \hat{f}(s)\chi_s(x). \tag{3.1}$$

(Note that the sum is of real numbers, not a sum mod 2)

This expression is called the *Hadamard transform* or *Fourier expansion* of p, and the real number $\hat{p}(s)$ is called the Fourier coefficient of p on s. Altogether, these coefficients are called the Fourier spectrum of p. The functions χ_s are the *parity functions*, because they compute the logical parity (XOR) of the bits from the bitstring x in s, i.e., from $(x_i)_{i \in s}$. The parity functions are defined as:

$$\chi_s(x) = (-1)^{s \cdot x}, \tag{3.2}$$

where $s \cdot x$ corresponds to the mod 2 sum of the bitwise AND between bitstrings s and x:

$$s \cdot x = \bigoplus_{i=1}^{n} x_i s_i \tag{3.3}$$

The parity functions are orthogonal:

$$\sum_{x \in \{0,1\}^n} \chi_s(x)\chi_{s'}(x) = 2^n \delta_{s,s'} \tag{3.4}$$

and also symmetric

$$\sum_{s \in \{0,1\}^n} \chi_s(x)\chi_s(x') = 2^n \delta_{x,x'} \tag{3.5}$$

These properties let us define the inner product on pairs of functions $p, q : \{0,1\}^n \to \mathbb{R}$, which will let us show how to obtain the Fourier coefficients. The usual inner product on $\mathbb{R}^{2^n}$ would correspond to $\sum_{x \in \{0,1\}^n} p(x)q(x)$, but it is convenient to scale this by a factor of 2^{-n}, thus making it an average rather than a sum. Therefore, given two real-valued Boolean functions p and q, its inner product is defined by as:

$$\langle p(x), q(x) \rangle = \frac{1}{2^n} \sum_x p(x)q(x) \tag{3.6}$$

Due to $s \in \{0,1\}^n$, there are 2^n parity functions. The Fourier expansion can then be thought of a linear combination of them and, since they satisfy the orthogonality condition, they constitute a linearly independent basis which justifies the uniqueness of the Fourier expansions. Because the parity functions constitute such linearly independent basis, we can express the Fourier coefficients $\hat{p}(s)$ as the projection of the function $p(x)$ in their corresponding parity functions $\chi_s(x)$; and, similarly to 3.1, its unique multilinear expansion follows from here:

$$\hat{p}(s) = \langle p(x), \chi_s(x) \rangle = \frac{1}{2^n} \sum_{x \in \{0,1\}^n} p(x)\chi_s(x) \tag{3.7}$$

3.2.1 Correlators

Boolean functions have had an effect in statistical physics, where physicists are more used to the notion of *correlators*. They also study the correlation of probability distributions, similar to what I do in the sampling algorithm. In fact, correlation functions are used as a measure of the order in a system. These functions can describe how microscopic variables, such as the magnetic spin quantum number and the density, are related at different positions. Specifically, correlation functions let us quantify how microscopic variables co-vary on average across space and time. Returning to the example of the spin, the parity function $\chi_s(x)$ can be interpreted as the product of the spins in a system, where the bitstring s indicates the spins selected and x their direction.

Pragmatically, these correlators are used in the project as **rescaled Fourier coefficients** which will help write the marginal probabilities of the sub-bitstrings of a given bitstring x.

$$C(p(x); s) = 2^n \hat{p}(s) = \sum_{x \in \{0,1\}^n} p(x)\ \chi_s(x), \tag{3.8}$$

where the notation $C(p(x);s)$ refers to the correlator of a bitstring s given a Boolean function $p(x)$, which in our case will be the probability distribution. The Fourier expansion with correlators just differs from 3.1 in the scaling factor:

$$p(x) = \frac{1}{2^n} \sum_{s\in\{0,1\}^n} C(p(x);s)\chi_s(x). \tag{3.9}$$

Having introduced the Fourier analysis of Boolean functions (and the notion of correlators) we can show different definitions of the HOG and fidelity. These will be used in **chapter 6** when I discuss the results obtained by the algorithms. Let $p(x)$ be the ideal distribution of the random quantum circuit, while $q(x)$ is the classical simulation of the noisy quantum computer, and where we might limit the order of correlators. For such $q(x)$, the High Output Generation (HOG) is defined as:

$$\begin{aligned} \mathrm{HOG}(p,q) &= \sum_x q(x)p(x) && (3.10)\\ &= \frac{1}{2^{2n}} \sum_s C_p(s)C_q(s) \sum_x \chi_s(x)\chi_s(x) && (3.11)\\ &= \frac{1}{2^n} \sum_s C_p(s)C_q(s), && (3.12) \end{aligned}$$

where I expanded the distributions in terms of their correlators to obtain (3.11) and I used the orthogonality of the parity functions, equation (3.4), for the last equality. Thus the fidelity can be expressed as:

$$\mathcal{F}_{\mathrm{XEB}} = 2^n \mathrm{HOG} - 1 = \sum_{s\neq 0} C_p(s)C_q(s) \tag{3.13}$$

Before moving to the marginal probabilities I would like to mention what the order of the correlator/Fourier coefficient is. As we have seen, we express a probability distribution $p(x)$ as a sum over all possible values x can take. In our specific case, $x \in \{0,1\}^n$ so there are 2^n correlators / Fourier coefficients. Their order corresponds to the *Hamming weight* of their corresponding bitstring s ($|s|$). It is defined as the number of '1's it contains. For example, the correlator of bitstring 1101, $C(1101)$, has order 3.

Overall, from basic combinatorics there are $\binom{n}{k}$ coefficients of order k. The **degree** of p is defined as $\max\{|s| : \hat{p}(s) \neq 0\}$, which is just the degree of p as a real n-variate polynomial. This way we can interpret the order of a correlator/Fourier coefficient as the degree to which it corresponds, or the number of spins which contribute to it.

3.3 Marginal probabilities

In simple terms a marginal probability is the probability of an event irrespective of the outcome of another variable. Let x be expressed as $x = (x_1, x_2, \ldots, x_n)$. Thus, we interpret by x_z as the bitstring that contains the leftmost l bits of x, i.e., $x_z = (x_1, x_2, \ldots, x_l)$ where $l < n$ and n is the size of our intended output x. The marginal $p(y)$ of $p(x)$ over a sub-bitstring x_z of size l is then given by:

$$p(y) = \sum_{x:x_z=y} p(x), \tag{3.14}$$

where the set $\{x : x_z = y\}$ refers to all the bitstrings of size n that begin with sub-bitstring $x_z = y$. We now use the Fourier expansion of $p(x)$ to obtain

$$p(y) = \sum_{x:x_z=y} p(x) \tag{3.15}$$

$$= \sum_{x:x_z=y} \sum_{s} \hat{p}(s)(-1)^{x\cdot s} \tag{3.16}$$

$$= \sum_{s} \hat{p}(s) \sum_{x:x_z=y} (-1)^{x\cdot s} \tag{3.17}$$

Let's focus on the second sum from equation (3.17). We can divide the sum into two parts: one focusing on the first l bits of x, i.e., on sub-bitstring x_z; and the other focusing on the remaining bits, on sub-bitstring $x_{z^*} = (x_{l+1}, x_{l+2}, \ldots, x_n)$, where z^* is the vector of size $n-l$ containing the complementary indices to z. This way we are expressing x as a concatenation of the two bitstrings $x = x_z||x_{z^*}$ (where $||$ denotes concatenation). Expressing s in a similar way, i.e., $s = s_z||s_{z^*}$ we obtain:

$$\sum_{x:x_z=y} (-1)^{x\cdot s} = (-1)^{x_z\cdot s_z} \sum_{x_{z^*}} (-1)^{x_{z^*}\cdot s_{z^*}} \tag{3.18}$$

We can use the orthogonal property of the parity functions (3.4) (with $s = s_{z^*}$ and $s' = \bar{0}_{z^*}$) to obtain an expression for the second term: $\sum_{x_{z^*}} (-1)^{x_{z^*}\cdot s_{z^*}} = 2^{n-l}\delta_{s_{z^*},\bar{0}_{z^*}}$. Substituting into equation 3.18 we obtain:

$$\sum_{x:x_z=y} (-1)^{x\cdot s} = 2^{n-l}(-1)^{y\cdot s_z}\delta_{s_{z^*},\bar{0}_{z^*}}, \tag{3.19}$$

where I used $x_z = y$. Now we can continue equation 3.17 to give

$$p(y) = 2^{n-l} \sum_{s} \hat{p}(s)(-1)^{y\cdot s_z}\delta_{s_{z^*},\bar{0}_{z^*}} \tag{3.20}$$

$$= 2^{n-l} \sum_{s_z} \hat{p}(s_z, \bar{0}_{z^*})(-1)^{y\cdot s_z}, \tag{3.21}$$

where the notation $\bar{0}_{z^*}$ is explicitly used to indicate that vector s is zero at positions z^*. At this point we can use the correlators as scaled Fourier coefficients to give

$$p(y) = \frac{1}{2^l} \sum_{s_z} C(p(x); s_z, \bar{0}_{z^*})(-1)^{y\cdot s_z} \tag{3.22}$$

$$= \frac{1}{2^l} \sum_{s_z} C(p(x); s_z)(-1)^{y\cdot s_z}, \tag{3.23}$$

where the notation was dropped to indicate that the last $n-l$ positions are 0. Notice that **equation (3.24) is equivalent to equation (3.1)** and reducing its size from n to l.

$$p(y) = \sum_{s_z \in \{0,1\}^l} \hat{p}(s_z)\chi_{s_z}(y) \tag{3.24}$$

3.4 Chain rule

It is very useful to obtain an expression for a marginal $p(y, y_w)$ of size $l+1$, where y_w is the single bit being added to bitstring y from which we already know its marginal. This will be the **core idea** for the sampling. The chain rule is:

$$p(x_1, x_2, \ldots, x_n) = p(x_1)p(x_2|x_1)\ldots p(x_n|x_{n-1}, \ldots, x_1). \tag{3.25}$$

To generate a sample, one needs to do a loop of size n, where at each step one decides the value of x_i (ideally with real random numbers, but later I use a pseudorandom number generator for simplicity) according to the conditional probability:

$$p(x_i|x_{i-1}, \ldots, x_1) = \frac{p(x_i, x_{i-1}, \ldots, x_1)}{p(x_{i-1}, \ldots, x_1)}. \tag{3.26}$$

We can modify equation (3.23) by adding a bit s_w so that:

$$p(y, y_w) = \frac{1}{2^{l+1}} \sum_{s_z \in \{0,1\}^l} \sum_{s_w} C(p(x); s_z, s_w)(-1)^{y \cdot s_z + y_w \cdot s_w} \tag{3.27}$$

Since s_w can only take two values, 0 or 1, we can expand the first sum in equation (3.27) to express the marginal of $y||y_w$ as:

$$p(y, y_w) = \frac{1}{2^{l+1}} \left[\sum_{s_z} C(p(x); s_z, 0)(-1)^{y \cdot s_z} + \sum_{s_z} C(p(x); s_z, 1)(-1)^{y \cdot s_z + y_w} \right] \tag{3.28}$$

$$= \frac{1}{2} \left[p(y) + (-1)^{y_w} \frac{1}{2^l} \sum_{s_z} C(p(x); s_z, 1)(-1)^{y \cdot s_z} \right], \tag{3.29}$$

where I used equation (3.23) in the first sum of (3.28). As I will explain in the next section, for my algorithm I computed the marginal probability resulting after adding a 0 to bitstring y of given size l. This is the **most important equation**, and the one that will be implemented at the core of my algorithm:

$$p(y, 0) = \frac{1}{2} \left[p(y) + \frac{1}{2^l} \sum_{s_z} C(p(x); s_z, 1)(-1)^{y \cdot s_z} \right]. \tag{3.30}$$

3.5 Sampling

Having access to the correlators of a distribution p allows us to generate samples by using the marginal probabilities [20]. To begin with, $p(0)$ is computed and used to sample the first bit of my outcome, which is either 0 or 1. Then, if the first bit turned out to be a 0, $p(00)$ is computed; otherwise we compute $p(10)$, and continue subsequently until a bitstring of size n is reached. Note that only equation (3.30) is used to compute the marginals for adding a bit 0 to the current outcome. This is because $p(y, 1)$ can be obtained via subtraction: $p(y, 1) = p(y) - p(y, 0)$. Further, note that I read the notation of $p(10)$ from left to right. That is, $p(10)$ corresponds to the marginal where the first bit is 1, and the second is 0.

In order to sample a bitstring y of size $l < n$ from the ideal distribution $p(x)$, we need to have $\sum_k \binom{l}{k} = 2^l$ correlators. It is straightforward to see that the number of terms grows exponentially with the size of the qubits, which contributes to the fact that simulating random quantum circuits takes exponential resources in space and time. There are three approaches I will consider in the next chapter for generating samples:

1. **Fast sampling** If n is such that one can afford to store the 2^n correlators, it is possible to have a *learning phase* to compute the marginal probabilities in a tree data structure before doing sampling. This will make the sampling exponentially faster, because to obtain an outcome one only needs to traverse the tree from the root to the leaves, going through the already computed marginal probabilities and generating the sample qubit by qubit. Each sample will take $\log_2 2^n = n$ steps
2. **On-the-fly sampling** This method computes the marginal probabilities required for a given sample during its generation. By not storing and reusing the marginals, it will have to compute n different marginals using equation (3.30) for every sample. Note that the sum of equation (3.30) contains 2^l correlators, with l being the size of the marginal, so each sample generated will take an exponential time.
3. **Hybrid** This approach computes the marginal probabilities while it generates a sample, as in **on-the-fly sampling**, but it stores the probabilities in a data structure. By keeping them, when sampling a large number of bitstrings, the marginals that have been already computed can be reused for faster sampling. This avoids having a *learning phase*, as in **fast sampling**, but benefits from accessing the already computed marginals. Nevertheless, there exists the possibility of adding a *partial learning phase*: instead of learning all the marginals, it can learn the marginals up to a given number of qubits l, and then compute and store the rest while sampling.

3.5.1 Restricting Fourier coefficients

The concentration of Fourier coefficient depends on the model of noise. It has been shown that noise on detectors produce concentration [20], while local depolarising noise does not [43]. Due to the connection between XEB and Fourier coefficients, one can decide to "hack" Google's XEB test by suppressing higher-order Fourier coefficients to reach the required $\mathcal{F}_{\text{XEB}}$. It is a known fact that for random quantum circuits,

$$\mathbb{E}_U[C_p(s)C_p(s)] = \frac{2^n - 1}{4^n - 1} = \frac{1}{2^n + 1} \approx \frac{1}{2^n}, \tag{3.31}$$

where the expectation is over the set of unitaries $U(2^n)$, which the random quantum circuits are supposed to mimic. We see that each correlator contributes with a weight $\approx 1/2^n$, as the error is negligible. With the $\mathcal{F}_{\text{XEB}}$ in terms of correlators (c.f. equation (3.13)), the more correlators we make zero the lower the fidelity we will obtain.

This gives way to create an strategy to hack Google's XEB test. Indeed, if one has the goal of achieving a specific value of $\mathcal{F}_{\text{XEB}}$, it is not necessary to simulate the random quantum circuit completely since we can determine which correlators are needed to reach that value. To hack Google's XEB test we could obtain the correlators needed to get $\mathcal{F}_{\text{XEB}} \approx 0.2\%$ for their Sycamore circuits, and only use those while sampling.

Chapter 4

Sampling Algorithm

4.1 Big Picture

As discussed in the introduction, I want to sample using the Fourier coefficients that corresponds to the distribution of the quantum circuit. They could be obtained with TN contractions or other techniques, but I artificially compute all the correlators using a brute-force approach. This is because my project focuses on the sampling procedure, which does not depend on how the correlators are computed.

The three different approaches for the sampling algorithm **follow the same set up**, which is executed by the `get_sampling_algorithm` method in `src/algorithms.py`. However, the **on-the-fly** and **hybrid** approaches are discussed before I mention the **fast** algorithm. This is because the fast approach can be thought of as an extreme case of the hybrid one, and the implementation has been designed so that the fast algorithm comes up this way. The logic flow of the 3 approaches to sampling is shown in figure 4.1 below. Moreover, there are instructions in the README.md file to try the methods discussed in this section. Before moving on to the main procedures for sampling, the programming language and the set up required to start generating samples are discussed.

4.2 Programming Language

For the implementation of the algorithm, the programming language chosen was Python 3.9.7. Python is a free and open-source programming language that can be used for web development, AI, and so on. It can also be used for quantum computing because companies such as IBM, Google or Microsoft have developed their own quantum libraries in Python (Qiskit, Cirq and Q#, respectively). Due to its object-oriented features, the code for the simulation can be organised in objects for a better representation and understanding.

several different libraries have been used throughout the whole project. The main libraries for the quantum computing operations were **cirq** and **cirq_google**. **Numpy** was very important to make the algorithm more efficient. It is based on well-optimized C code and provides powerful methods for operating with large n-dimensional arrays.

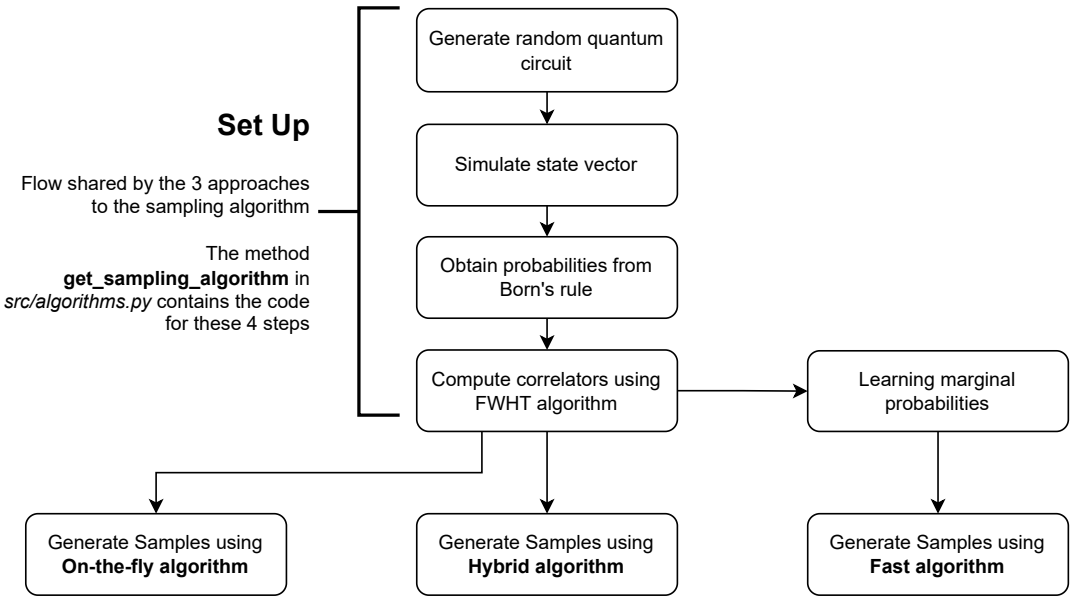

Figure 4.1: Flow chart containing the logic of the algorithms

Moreover, **numpy** has a very good random library, which was used through their generator `numpy.random.default_rng()`. The generator corresponds to PCG64, which has a period of 2^{128} so during sampling a number is not generated twice. The results of the simulations run were stored in .csv files with **pandas**. On the other hand, **matplotlib**, specifically a sub-module called **pyplot** was used to do the graphs, whose appearance was modified with **seaborn**. Lastly, the built-in Python libraries **time** and **timeit** were used to obtain the running times of the algorithms and benchmark them.

4.2.1 Google Cirq's `.sample()` method

The reader may ask what is the point on developing a sampling algorithm when cirq already provides a `sample` method from its `cirq.sim` module. The reason behind is obviously the implementation. After looking at Google's documentation in their github, `cirq.sim.sample` produces samples by calling `numpy.random.choice`. This function takes an array and a probability distribution. It generates a given number of samples from the array using the provided distribution. It is inefficient when the number of qubits n and/or depth m increases because it needs the probabilities of all outcomes. With large circuits this is not feasible. It would be more efficient to compute the marginals and sample from the noisy distribution by, for example, obtaining the Fourier spectrum without the most-expensive correlators and using TN contractions.

4.3 Set up

4.3.1 Generating random quantum circuits

Before starting the simulation we need to have the quantum random circuit for n qubits. Google Cirq was chosen in order to perform similar experiments to those

```
cirq.experiments.random_rotations_between_grid_interaction_layers_circuit(
      qubits = qubits,
      depth = depth,
      two_qubit_op_factory = (lambda a, b, _: cirq.ops.ISwapPowGate()(a, b)),
      pattern = cirq.experiments.GRID_STAGGERED_PATTERN,
      single_qubit_gates = (
          cirq.ops.X ** 0.5, cirq.ops.Y ** 0.5,
          cirq.ops.PhasedXPowGate(phase_exponent=0.25, exponent=0.5)
      ),
      add_final_single_qubit_layer = True,
      seed = None
  )

```

Figure 4.2: Method declaration for creating random quantum circuit

Google did, and reproduce the Sycamore circuits with a smaller number of qubits, $n < 23$. In the beginning, I thought that I would have to create the circuits manually because their documentation guides were very limited to simple circuits. I checked the module they use for testing and found methods that simulated random circuits such as `cirq.testing.random_circuit`. However, after a long time I decided to ask a question in "quantumcomputing.stackexchange.com" and a Google software engineer recommended to use the method from Figure 4.2.

He added, however, that it might not be the circuits that one would expect. I therefore proceeded to check examples with a small number of qubits and found that it produced circuits similar to the ones Google used in 2019, except that they did not follow any particular sequence as discussed in 2.2.2. As a default, it was using Control-Z gates as the two-qubit gates. I modified it to use the ISWAP gate, which belongs to the fSim gate family as Google's team discussed in their Supplementary Material. This can be done by using `cirq.ops.ISwapPowerGate`, whose default arguments turn it into a normal ISWAP gate, as the method's `two_qubit_op_factory`.

4.3.2 Simulating the circuit

In order to simulate the circuit, I use my method `simulate_sycamore_circuit(N, depth, num_extra_qubits)` defined in `src/simulations.py`, where N is the number of qubits the circuit has. Because Google's experiment is 2-dimensional, I need to have a staggered grid for the qubits. Google provides the grids they use in different devices through `cirq_google`. Specifically, I fetch the grid of their `Sycamore23` device, which I can then truncate depending on the specific size N of my circuit. Then, it generates a circuit following 4.3.1. Lastly, it simulates the circuit using a sparse matrix simulator `cirq.Simulator` and returns a `StateVectorTrialResult` object.

4.3.3 Obtaining probabilities

Having the result from `simulate_sycamore_circuit`, the state vector can be obtained by calling its `.final_state_vector` attribute. Expanding it in the computational basis:

$$|\Psi,t\rangle = \sum_i c_i(t)\,|i\rangle\,, \tag{4.1}$$

where $c_i(t)$ are the coefficients of the complete set of eigenstates $\{|i\rangle\}$, also known as *probability amplitudes*. If we use the representation with column vectors we will have:

$$|\Psi,t\rangle \to \begin{pmatrix} c_1(t) \\ c_2(t) \\ \vdots \\ c_n(t) \end{pmatrix} = c_1(t) \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} + c_2(t) \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix} + \cdots + c_n(t) \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}$$

Following Born's rule [39], the absolute value of the wavefunction squared is the probability density function. This means $|c_i(t)|^2$ is the probability of eigenstate $|i\rangle$, and it corresponds to the ideal distribution of the quantum device. Therefore, I obtain the probability array by taking the square modulus. To obtain the probability of a bitstring we need the coefficient to which it corresponds. For example, if we have a system of 3 qubits and want to know $p(101)$, we have $p(101) = |c_5|^2$ because $101_2 = 5_{10}$.

4.3.4 Obtaining correlators

Having obtained the probability array by squaring the absolute value of the state vector, we can now use our knowledge of Fourier analysis. It turns out that the *Hadamard transform* is equivalent to a multidimensional Discrete Fourier Transform of size $2 \times 2 \times \cdots \times 2 \times 2$ [2]. It decomposes an arbitrary input vector into a superposition of Walsh functions. Therefore, we will use such transform to obtain the Fourier spectrum of our distribution $p : \{0,1\}^n \to \mathbb{R}$.

It is important to mention the implementation of such transform. A naive one will require computing the matrix $H \otimes H \otimes \cdots \otimes H \otimes H = H^{\otimes n}$ of size $2^n \times 2^n$. Having n qubits and defining $N = 2^n$, the naive method will have a computational complexity of $O(N^2)$. There are more efficient ways of acting with $H^{\otimes n}$ in a vector of size 2^n. Similar to the Fast Fourier Transform algorithm, there exists a divide-and-conquer Fast Walsch-Hadamard algorithm that has the same recurrence as Merge Sort, with only $O(N \log N)$ additions or subtractions [29, Section 2.2].

I used Python example code from this Wikipedia's article which implements the FWHT algorithm in-place and without adding the $1/\sqrt{2}$ normalization factors. Therefore, after applying such algorithm to my probability array, I obtain my array of correlators and not of Fourier coefficients. This is a **better choice to reduce the number of multiplications and decrese the numerical error** of the computations. I store these values in a numpy array called `correlators`.

Notation aside

Before going on explaining the algorithms, I want to discuss the notation used for the correlators. When we developed the mathematical tools for the Fourier analysis of Boolean functions, I added the definition of marginal probabilities and expressed them in terms of correlators. Looking back at equation (3.17), it is very important to note that if we are working with bitstrings of n qubits and want to obtain the marginal probability of a bitstring with l qubits (where $l < n$), the remaining qubits are set to zero. For example, imagine we have a circuit of $n = 3$ qubits and we want to obtain the marginal probability of bitstring 11, hence $l = 2$. From equation (3.18) we will have:

$$\begin{aligned} p(11) &= \frac{1}{2^2}\left[C(p(x);00,0) - C(p(x);01,0) - C(p(x);10,0) + C(p(x);11,0)\right] \\ &= \frac{1}{2^2}\left[C(000) - C(010) - C(100) + C(110)\right], \end{aligned}$$

where I dropped the notation in the second equation to explicitly show that **the last $n-l=1$ qubits are set to 0**. This is very important because the way of obtaining these correlators is by indexing them from the array `correlators`. For example, the correlators $C(01)$ and $C(010)$ are not the same. The former corresponds to the 1^{st} Fourier coefficient (because $01 = 1_{10}$) while the latter corresponds to the $010_2 = 2^{\text{nd}}$.

4.4 On-the-fly algorithm

4.4.1 Overview

The main idea behind the sampling algorithm is to obtain outcome x by computing the values of its positions from left to right. That is, given that bitstring x is made up of n bits, i.e., $x = (x_1, x_2, \ldots, x_n)$, we would start computing the value of x_1, x_2 and successively until we reach x_n. To do so, we calculate the marginal probabilities with the chain rule and use them to set x_i equal to 0 or 1 in each step.

This will be similar to traversing a decision tree which contains the marginals in its nodes. Figure 4.3 shows an example tree for a circuit with 3 qubits. An important thing to note is that the probabilities of each level must add up to 1. The root node is the base case and we let the probability of $\emptyset$ be $p(\emptyset) = 1$. Moving down to level 1 we must have $p(0) + p(1) = 1$ and similarly until level n.

In order to obtain a sample, we will follow this procedure:

1. Initialise `outcome` to the empty string. This is the variable where we will store the sample step by step. Set current position to node $\emptyset$.
2. Compute marginal of `outcome` + 0. In the first iteration, we will compute $p(0)$ because `outcome` will be the empty string.
3. Use a pseudo random number generator to obtain a number between 0 and the marginal of our current position. For the first iteration, since we start in node $\emptyset$, we obtain a number between 0 and $p(\emptyset) = 1$.
4. If the generated number $< p(0)$ we add a 0 to `outcome`. Otherwise we add a 1.

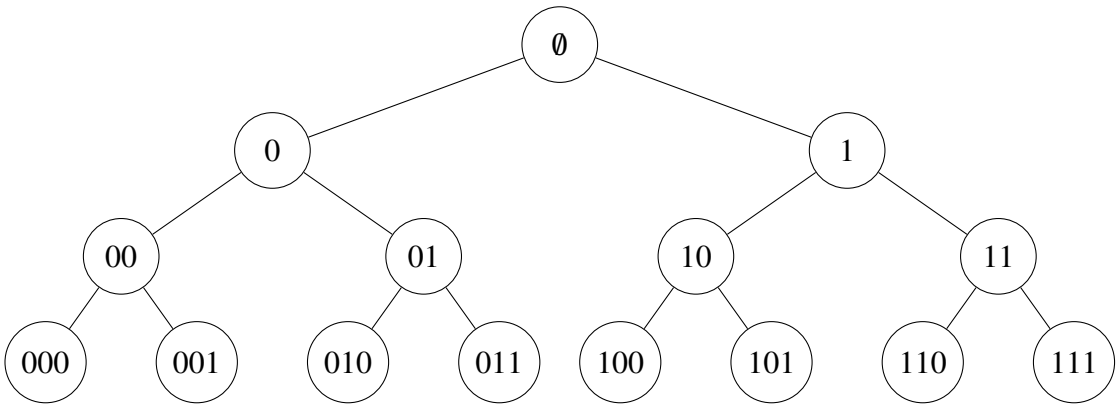

Figure 4.3: Binary tree whose nodes contain the marginals of a circuit with 3 qubits.

5. Move to a child node depending on the outcome from the previous step. Repeat this process from 2. until we reach the end of the tree.

An example with bitstrings of size $n = 2$ will be the following:

- Initialise outcome to the empty string: `outcome` = ''
- compute marginal of `outcome` + 0, i.e., compute marginal $p(0)$. For this example let's set $p(0) = 0.34$.
- call pseudorandom number generator between 0 and $p(\emptyset) = 1$. Assume it gives 0.45. Because $0.45 > p(0)$, we add a '1' to outcome so that know `outcome` = 1. Now we move our current position from node $\emptyset$ to node 1.
- compute marginal of `outcome` + 0, i.e., compute marginal $p(10)$. Let's set $p(10) = 0.4$ for this example.
- call pseudorandom number generator between 0 and $p(1) = 1 - p(0) = 0.66$. Assume it gives 0.2. Since $0.2 < 0.66$, we add a '0' to outcome. `outcome` = 10.
- We have reached the end of our tree (for $n = 2$). Our generated sample is 10.

4.4.2 Procedures

There are three main procedures used by the **on-the-fly** approach. Note that they have the word *slow* to distinguish them from the –faster– ones described in the next section:

- Algorithm 1: `get_correlators_for_marginal_slow(y)`
- Algorithm 2: `get_prob_add_zero_slow(y, prob_limit)`
- Algorithm 3: `sample_random_circuit_slow()`

These methods belong to a Python class named `OnTheFlySampler`. An object of such class is created by `get_sampling_algorithm(num_qubits, seed, slow, VERBOSE)` given that `slow = True`. **This class has** `num_qubits` **and** `correlators` **as class variables, which will be used in the pseudocode**.

First of all we describe `get_correlators_for_marginal_slow(y)`, where y is a bitstring $y \in \{0,1\}^l$ with $l < n$. It returns an array with the necessary correlators to compute the marginal $p(y,0)$ following equation (3.24), which I show again to stress its

importance. It is necessary to note that I am not retrieving the same correlators multiple times, so avoiding being inefficient here. However, there is an implementation change of Algorithm 1 that I show in section 4.5.

$$p(y,0) = \frac{1}{2}\left[p(y) + \frac{1}{2^l}\sum_{s_z} C(p(x);s_z,1)(-1)^{y\cdot s_z}\right],$$

The second procedure is `get_prob_add_zero_slow(y, prob_limit)`. It calls `get_correlators_for_marginal(y)` to get the needed correlators and signs from equation (3.24), and it returns $p(y,0)$. Note that `prob_limit` is the marginal $p(y)$. Equation (3.24) is divided into a sum of marginal $p(y)$ and the correlators from Algorithm 1. Lastly, `sample_random_circuit_slow()` returns a sample **on-the-fly** using the previous two procedures.

Algorithm 1 Obtaining correlators to compute marginal $(y,0)$

```
1:  procedure GET_CORRELATORS_FOR_MARGINAL_SLOW(y)
2:      l ← length of y
3:      y ← Convert bitstring y to integer
4:      marginal_correlators ← new array
5:      for i ← 0 to 2^l − 1 do
6:          s_z ← obtain bitstring from integer i
7:          index ← s_z with 1 appended to the end
8:          correlator ← correlators[index]
9:          sign ← (−1)^(s_z·y)
10:         marginal_correlators.append(correlator × sign)
11:     end for
12:     return marginal_correlators
13: end procedure
```

Algorithm 2 Get marginal of appending a 0 to bitstring y

```
1:  procedure GET_PROB_ADD_ZERO_SLOW(y, prob_limit)
2:      l ← length of y + 1
3:      marginal_correlators ← get_correlators_for_marginal_slow(y)
4:      return 1/2 (prob_limit + 1/2^l sum(marginal_correlators))
5:  end procedure
```

There are a few things from Algorithm 3 that might not seem obvious:

- In line 3 we initialise `prob_limit` to 1 because we start at the root node, so $p(\emptyset) = 1$. We will have to keep updating this value to generate random numbers in the correct range. This is very important because the **sum of marginals in every level has to be equal to 1**, therefore the value of the children will be getting smaller, and we cannot keep generating random numbers in the [0,1) range.
- In line 5 we initialise `prob_add_zero` to keep track of the marginal probabilities. Further, depending on whether x_i is 0 or 1 , we update the `prob_limit` accordingly, which contains $p(y)$ and is used to compute $p(y,0)$ following equation

Algorithm 3 Generating one sample of the random quantum circuit

```
1: procedure SAMPLE_RANDOM_CIRCUIT_SLOW()
2:     result ← empty string
3:     prob_limit ← 1
4:     idx0, idx1 ← indexes for correlators C(0) and C(1)
5:     prob_add_zero ← 1/2 × (correlators[idx0] + correlators[idx1])
6:     for step ← 0 to num_qubits −1 do
7:         flipped_coin ← random number between 0 and prob_limit
8:         if flipped_coin ≤ prob_add_zero then
9:             outcome + = 0
10:            prob_limit = prob_add_zero
11:        else
12:            outcome + = 1
13:            prob_limit = prob_limit − prob_add_zero
14:        end if
15:        if step ≠ num_qubits −1 then
16:            prob_add_zero ← get_prob_add_zero_slow(outcome,
   prob_limit)
17:        end if
18:    end for
19:    return (outcome)
20: end procedure
```

(3.24). That is, if we have bitstring y and we add a 1, then $p(y,1) = p(y) - p(y,0)$ which is done in line 13. Otherwise, if we add a 0, we just set `prob_limit` as $p(y,0)$, i.e., `prob_add_zero`.

- We need the if statement from line 15 to avoid an `IndexError`: when we reach the end we cannot compute the marginal probability of adding an extra qubit: we do not have the correlators for `num_qubits`+1 qubits.

The complexity of this last procedure is mainly determined by line 21 when we compute the marginal probabilities. This is run `num_qubits`−1 times. Analysing `get_prob_add_zero_slow` we see that for an outcome y of size l it takes $O(2^l)$ to run `get_correlators_for_marginal_slow(y)`. We call this method after obtaining the value of the first bit until outcome has a size of $n-1$, i.e., for $k \leftarrow 1$ to $n-1$. It takes

$$\sum_{k=1}^{n-1} 2^k = \sum_{k=0}^{n-1} 2^k - 1 = \frac{1-2^n}{1-2} - 1 = \frac{-2^n+2}{-1} = 2^n - 2.$$

steps, hence the complexity scales exponentially with the number of qubits, $O(2^n)$.

4.5 Hybrid algorithm

When we compute the marginals **on-the-fly**, we do not store them. Hence, if we want to generate another sample we need to compute them again. In this section I explain how I

reused the marginals, modified the implementation so that we can **limit the order of the correlators** used and made the **code more efficient using numpy**.

4.5.1 Storing the marginals

I add a class variable `marginals` to store the marginals after computing them during the generation of a sample, as can be seen below in line 18 of Algorithm 5. `marginals` is a `defaultdict` whose keys are bitstrings and its values are the probabilities. For example, to retrieve the stored marginal $p(011)$, I would call `marginals['011']`. Once the marginal has been computed, it will be reused for the next samples if they need it.

4.5.2 Limiting the order of correlators

Retrieving the correlators of a specific order is more involved. The first thing I do is create an array, called `order_arr`, which contains the orders of the correlators depending on their position. For example, if we have a circuit of 4 qubits, the correlator for bitstring 1100 will be in position $1100_2 = 12$ and we would extract it by calling `correlators[12]`. Recalling our previous definition of order in section (3.2.1), $C(1100)$ has order 2. I obtain `order_arr` by mapping the function `gmpy2.popcount`, which efficiently counts the number of 1s a number in binary has, to an array containing numbers from 0 to $2^n - 1$. For $n = 2$ it would do:

$$\begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 00_2 \\ 01_2 \\ 10_2 \\ 11_2 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 1 \\ 1 \\ 2 \end{pmatrix}$$

Having this `order_arr`, now I need some way of accessing the correlators with a specific order. For this reason, I use `zip` in Python to create a 2-dimensional array, `orders_and_correlators`, where every row has tuple containing its order (in the 1st column) and its value (in the 2nd column). For an example with $n = 2$ qubits:

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 2 \end{bmatrix} + \begin{bmatrix} C(00) \\ C(01) \\ C(10) \\ C(11) \end{bmatrix} \rightarrow \begin{bmatrix} [0, C(00)] \\ [1, C(01)] \\ [1, C(10)] \\ [2, C(11)] \end{bmatrix}$$

With this 2-dimensional array I can extract a 1-dimensional boolean array that contains `True` if the correlator's order is smaller than k, and `False` otherwise. This is done in line 6 from Algorithm 4. I proceed to use it in line 7 for efficiently indexing the array with masking.

4.5.3 Procedures

The **hybrid** approach is implemented in the `HybridSampler` class, and is based on three methods and four class variables: `marginals`, `num_qubits`, `correlators` and

`orders_and_correlators`. An object of such class is created by `get_sampling_algorithm(num_qubits,seed,slow,VERBOSE)` given that slow = False.

- Algorithm 4: `get_correlators_for_marginal_to_order(y, order)`,
- `get_prob_add_zero(y, prob_limit, order)`
- Algorithm 5: `sample_random_circuit_hybrid(order)`

Similarly to the **on-the-fly** approach, the first procedure obtains the correlators required to compute $p(y,0)$. The implementation here lets the user choose the maximum order of the correlators used, and is coded more efficiently to **avoid the for loop** from lines $5 \rightarrow 11$ in Algorithm 1.

Algorithm 4 Obtaining correlators to compute marginal $(y,0)$

```
1:  procedure GET_CORRELATORS_FOR_MARGINAL_TO_ORDER(y, order)
2:      check order is ≤ the number of qubits
3:      l ← length of y
4:      idxs ← array of binary strings for integers from 0 to 2^l − 1
5:          ▷ Obtain Boolean array. Positions of correlators we want have value True:
6:      mask ← ordes_and_correlators[:, 0] ≤ order
7:      corr_idxs ← idxs[mask]
8:      parity_exp_products ← array with product of exponent s_z · y as in (3.3)
9:      parity ← Array of parities.  ▷ If exponent is 0, parity is 1; if exponent is 1, -1.
10:     marginal_correlators ← element-wise multiplication of parity and correlators.
11:     return marginal_correlators
12: end procedure
```

The second procedure is `get_prob_add_zero(y, prob_limit, order)`. This method computes $p(y,0)$ as Algorithm 2, but modifies its line 3 to use `get_correlators_for_marginal_to_order(y,order)`, instead of the previous `get_correlators_for_marginal_slow(y)`, for a shorter runtime.

Lastly, `sample_random_circuit_hybrid(order)` is the procedure which generates samples. While generating a sample, it computes the marginals **on-the-fly**, but it stores them so that they can be reused for the next samples. Note that we only compute the marginals of adding a '0' to our bitstring `outcome` since I implement $p(y,0)$. We can then obtain $p(y,1)$ simply by $p(y,1) = p(y) - p(y,0)$.

On the other hand, as I mentioned in the last part of **chapter 2** we can have a *partial learning phase* with the hybrid algorithm. I show the procedures for learning marginals in the next section, namely with Algorithm 7. The idea behind this is that, before moving on generating samples with Algorithm 5, we can perform a small learning phase where we compute a subset of the marginals. Due to my implementation, I can choose the level of the tree (c.f. Figure 4.3 for the marginal tree corresponding to a circuit of 3 qubits) at which I want to stop learning, thereby allowing for a fast learning phase that helps during sampling.

Algorithm 5 Generating one sample of the random quantum circuit

```
1: procedure SAMPLE_RANDOM_CIRCUIT_HYBRID(ORDER)
2:     outcome ← empty string
3:     prob_limit ← 1
4:     idx0, idx1 ← indexes for correlators C(0) and C(1)
5:     marginals['0'] = 1/2 × (correlators[idx0] + correlators[idx1])
6:     for step ← 0 to num_qubits −1 do
7:         prob_add_zero ← marginals[outcome + '0']
8:         flipped_coin ← random number between 0 and prob_limit
9:         if flipped_coin ≤ prob_add_zero then
10:            outcome + = 0
11:            prob_limit = prob_add_zero
12:        else
13:            outcome + = 1
14:            prob_limit = prob_limit - prob_add_zero
15:        end if
16:        if step ≠ num_qubits −1 then
17:            if marginals[outcome + '0'] not computed before then
18:                marginals[outcome + '0'] ← get_prob_add_zero(outcome,
   prob_limit, order)
19:                if marginals[outcome + '0'] > prob_limit then
20:                    marginals[outcome + '0'] ← prob_limit
21:                end if
22:            end if
23:        end if
24:    end for
25:    if last value of outcome is '1' then
26:        marginals[outcome] ← prob_limit
27:    end if
28:    return (outcome)
29: end procedure
```

4.6 Fast algorithm

The **fast algorithm** begins with *learning phase* where it computes and stores all the marginal probabilities so they can be reused. It is also implemented in the subclass of `Sampler` called `HybridSampler`. This is because it uses the same methods as the **hybrid** approach for computing $p(y,0)$ (`get_prob_add_zero` and `get_correlators_for_marginal_to_order`), but it does so during the learning phase. The **fast** approach can be interpreted as an extreme case of the **hybrid** one, where we do not store, but reuse the marginals because they have been computed before the sampling started. In total, we have the following:

- Algorithm 6, `add_marginal(outcome,pruning_depth,order,prob_limit)`: Helper method to recursively store all the marginals of bitstrings that start with `outcome`. If `pruning_depth` is smaller than `num_qubits`, the method stops after it has achieved level `pruning_depth` in the tree.
- Algorithm 7, `learning_marginals(pruning_depth, order)`: Carries out the learning of the marginals. Due to the implementation, one can choose a `pruning_depth` to stop the learning at some level of the tree (prune the tree).
- `sample_random_circuit_fast`: It is identical to Algorithm 5, but is modified to assume all the marginals have been stored by removing lines $16 \rightarrow 27$ from such Algorithm. This way, it avoids storing or checking whether they have been computed, because this has been done in the learning phase.

Algorithm 6 Adds marginals recursively

```
1:  procedure ADD_MARGINAL(outcome, pruning_depth, order, prob_limit)
2:      if length of outcome > pruning_depth then return
3:      end if
4:      marginal ← get_prob_add_zero(outcome, prob_limit, order)
5:      if marginal > prob_limit then
6:          marginal ← prob_limit
7:      end if
8:      marginals[outcome + '1'] ← prob_limt - marginals[outcome + '0']
9:      add_marginal(outcome + '0', pruning_depth, order, marginal)
10:     add_marginal(outcome + '1', pruning_depth, order, marginals[outcome + '1'])
11: end procedure
```

Algorithm 7 Learns marginals up to a given level of the tree

```
1: procedure ADD_MARGINAL(outcome, pruning_depth, order, prob_limit)
2:     marginals['0'] ← 1/2 × (correlators[idx0] + correlators[idx1])
3:     marginals['1'] ← 1 − marginals['0']
4:     add_marginal('0', pruning_depth, order, marginals['0'])
5:     add_marginal('1', pruning_depth, order, marginals['1'])
6: end procedure
```

Chapter 5

Testing

5.1 Correlators and negative probabilities

Obtaining the precise correlators for a marginal $p(y)$ was more difficult than I thought at first. In 4.3.4 I mention that we have to be careful indexing the correlators, and my efficient implementation for obtaining the correlators to compute $p(y,0)$ was indexing the wrong correlators. Specifically, `get_correlators_for_marginal_to_order` was indexing the correlators from bitstrings of size $l \neq n$ and thus retrieving correlators that did not correspond to the marginal I wanted to compute. I needed to pad the bitstrings with zeros in the end to make them of size n,

I did not realised about this problem until my implementation of the learning algorithm. With the *learning phase*, I could compute all the marginals and see whether there was any value that did not make sense. Indeed, I found out that some of the **probabilities had negative values**. Therefore, I decided to check with a hardcoded example whether the correlators that I was using to compute such marginals were correct; and I found that neither the values nor the parities were the appropriate ones. I also obtained **negative probabilities by limiting the order of the correlators**, but I discuss that in 5.2.

Fixing the issue

To fix the indexing problem, I defined a method `get_index(y, x, num_qubits)` which gives the correct index for a specific correlator. It takes a bitstring y of size $l < n$ from which we want to compute $p(y,0)$; x is an integer index $x \in [0, 2^l)$ and `num_qubits` is the number of qubits of the circuit, n.

Assume we have a circuit of 6 qubits and we want to compute the probability of adding a 0 to our bitstring y, $p(y,0)$, where $y = 101$. From equation (3.24) we need correlators $C(s_z, 1)$ with $s_z \in \{0,1\}^3$. However, we cannot index our class variable `correlators` with the corresponding integer values $(s_z, 1)$. The reason for this is that our `correlators` array has size 2^n and not 2^l, so those indexes would not correspond to the values we need. For instance, say we want to access correlator C(011, 1), i.e., when $s_z = 011$. We would need to call `correlators` with the value $011100_2 = 28$, `correlators[12]`. Revisiting equation (3.18), we know that we need to pad the

bitstring with $n-l$ zeros to the right:

$$p(y) = \frac{1}{2^l} \sum_{s_z} C(p(x); s_z, \bar{0}_{z^*})(-1)^{y \cdot s_z}$$

On the other hand, I also implemented more efficiently the calculation of the parity functions in `get_correlators_for_marginal_to_order` to avoid looping over the bitstrings y and s_z. My idea was to compute the parities of all correlators at once, $(-1)^{s_z \cdot y}$, and store them in a single array `parity`. The problem with numpy is that one cannot perform operations in arrays of characters, so I had to look for a different solution. I created a 2d array whose rows contain the bitstrings s_z & y, where & is the bitwise AND operator. This way it is possible to do a mod 2 sum over all the rows at once, and turn the resulting values (either 0 or 1) into 1 or -1, which are the results of the parity function. With such an array, I can return the Hadamard product `correlators` $\odot$ `parity` so that the parities are included with the correlators.

Testing

The best way I found to test whether the correlators were computed correctly was to do an example of 3 qubits by hand, and verify whether I obtain the same results. I generated a random quantum circuit and with the resulting state vector I obtained the following:

$$\texttt{probabilities} = [0.19, 0.07, 0.09, 0.23, 0.06, 0.13, 0.22, 0.01]$$

$$\texttt{correlators} = [1, 0.12, -0.1, -0.02, 0.16, -0.16, -0.02, 0.54]$$

We can easily check that `probabilities` sum up to 1. `correlators` was obtained by applying the FWHT algorithm. Using equation (3.23), the marginals of size 1 are:

$$p(0) = \hat{p}(0) + \hat{p}(1) = \frac{1}{2}(C(000) + C(100)) = \frac{1}{2}(1 + 0.16) = 0.58 \tag{5.1}$$

$$p(1) = \hat{p}(0) - \hat{p}(1) = \frac{1}{2}(C(000) - C(100)) = \frac{1}{2}(1 - 0.16) = 0.42 \tag{5.2}$$

And it is easily verifiable that $p(0) + p(1) = p(\emptyset) = 1$. Moving on to the marginals of 2 qubits we have:

$$p(00) = \hat{p}(00) + \hat{p}(01) + \hat{p}(10) + \hat{p}(11) \tag{5.3}$$

$$= \frac{1}{2^2}\left[C(000) + C(010) + C(100) + C(110)\right] \tag{5.4}$$

$$= (1 - 0.1 + 0.16 - 0.02)/4 = 0.26 \tag{5.5}$$

Similarly, we obtain $p(01) = 0.32$, $p(10) = 0.19$ and $p(11) = 0.23$. We can verify again that $\sum_{y \in \{0,1\}^2} p(y) = 1$. Following the same procedure with bitstrings of size 3 we will obtain the same exact values as vector `probabilities`, see appendix A. This is what we expect, since the example is for a circuit of 3 qubits.

I also checked that this method was working by comparing it against `get_correlators_for_marginal_slow(y)` since that method was working correctly. After I fixed the problem before, both methods agreed with their outputs.

5.2 Changing the order of correlators

In 4.5.2 I explain how I use the array `order_and_correlators` to efficiently retrieve the correlators of a specific order. I do so with numpy by indexing the correlators with a Boolean array that tells me which satisfy the condition `order` $\leq k$. This can be done in every step of the sampling process so the correlators always have order $\leq k$.

Similarly as in section 5.1, I sometimes found negative marginals when I limited the order of the correlators, which I noticed thanks to using my learning algorithm while setting different maximum orders k. However, after my fix from the previous section I found that I was indeed computing the correlators correctly. In fact, I checked my algorithm by learning the marginals for different arrays of correlators. I obtained those correlators by applying the FWHT procedure to several arrays of probabilities (which summed up to one) that I came up with. I could verify that the algorithm was limiting the order of the correlators correctly, because I obtained the same results by hand.

Nevertheless, I realized that only when I computed the marginals with the distribution from the quantum circuits I was obtaining negative probabilities. By examining the correlators, I understood that it could be a problem of numerical error, because I am computing an **approximation to the correlators**, and not the exact values. Specifically, from Fourier analysis we know the correlator of order 0 is $C(\bar{0}) = \sum_s p(s)(-1)^{s\cdot\bar{0}} = \sum_s p(s) = 1$, but it always has small fluctuations that make it a bit bigger or smaller. This variation, and errors in other correlators seem to be what caused the issue.

This is a **standard problem** with a defined solution, and not because of my code. I solved it by following the solution discussed by Ashley in [20, Section 3.2]. To avoid having negative marginals I modified the sampling algorithm through lines 19 and 20 of Algorithm 5. If I compute $p(y,0)$ and it turns out to be $p(y,0) > p(y)$, then I set $p(y,0) = p(y)$ and $p(y,1) = 0$ so that $p(y,0) + p(y,1) = p(y)$ is still satisfied.

5.3 Other tests

There have been more parts of the projects that I tested differently. I checked the circuits which the laptop was generating through the method recommended by the Google engineer (c.f. Figure 4.2). It turned out that they were using CZ gates as two-qubit gates, so I modified them to use the ISWAP gate. Moreover, I found out that they do not take the sequence ABCDCDAB into account, as Google does for their intractable circuits. Even without using the intractable sequence, my laptop struggles to simulate circuits with $n > 23$. Because the sampling algorithm is the same irrespective of this ordering, I decided to keep generating the circuits with that method.

Other parts of the code such as the FWHT algorithm, obtaining the `order_arr` and `order_and_correlators`, checking whether sampling worked and so forth were easier to test because I had the specific outputs/results of what I needed to obtain. For example, applying the FWHT algorithm to $[1,0,1,0,0,1,1,0]$ should give me $[4,2,0,-2,0,2,0,2]$. Similarly for the class variables, I could check they behaved as I expected. Lastly, I kept track of the random numbers by using the generator PCG64, whose period is of 2^{128} so it did not repeat the same numbers during sampling.

Chapter 6

Experiments and results

6.1 Fidelity decrease with order of correlators

6.1.1 Ideal distribution p

In **chapter 3**, I discussed different noise models one can consider in experiments with a quantum device. If there is noise during the measurement in a quantum circuit, we would expect the fidelity (and therefore XEB) of the quantum state to decay exponentially with the number of gates m, i.e., $(1-p)^m$ with p being the probability of error in a gate [20]. This happens because measurement noise concentrates the Fourier coefficients. If there is not a great deal of such noise, it could be possible to obtain high fidelity values when the higher-order correlators are suppressed. The number of gates is linearly related to the number of qubits n and depth of circuit D, hence the exponential decay of XEB can be related to the size and depth of the quantum device.

On the other hand, there are other types of noise where the concentration of the Fourier coefficients is not found. This can happen, for example, when one encounters local depolarising noise in the experiment with the quantum device. For these cases, however, it is not obvious how the XEB decays with the size of the circuits.

In Figure 6.1 I show a plot of the $\mathcal{F}_{\text{XEB}}$ versus the maximum order of correlators that were used in the calculation. To compute such values of the fidelity, I obtained the ideal probability distribution from the state vector of a random quantum circuit with n qubits. Then, I obtained the correlators with the FWHT algorithm which I used to compute the $\mathcal{F}_{\text{XEB}}$. In fact, since we are using the ideal probability distribution, we can reduce equation (3.13) to:

$$\mathcal{F}_{\text{XEB}} = \sum_{s\neq 0} C_p(s)C_q(s) = \sum_{s\neq 0} |C_p(s)|^2. \tag{6.1}$$

As we can see in Figure 6.1, $\mathcal{F}_{\text{XEB}}$ increases from a value of 0, when I only use the first correlator, $C(\bar{0})$, up to a value of 1 when considering correlators of all possible orders. This is expected because we are computing the fidelity using the ideal distribution p, and not a distribution q obtained experimentally on my laptop.

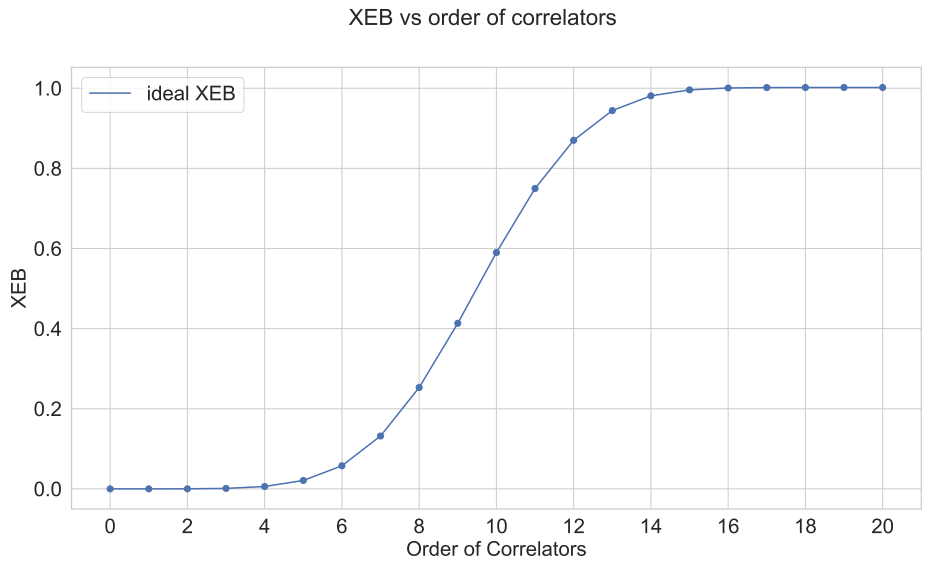

Figure 6.1: **Ideal XEB vs maximum order of correlators**. The graph shows the different XEB values computed for a random quantum circuit of 20 qubits, where the correlators from its ideal probability distribution have been limited to have a maximum order k, which is shown in the x-axis. The blue line corresponds to an interpolation of the XEB values. We can see how the XEB is bounded by 0 and 1, as discussed before in section 2.2.3, and there is a very small variation in its value when we consider the higher-order correlators. Remark that, from combinatorics, when the maximum order k is closer to the number of qubits n we add very few correlators, so these could be avoided to increase the performance of the computation. It is important to notice the steep decrease in XEB when the order of the correlators is close to half of the number of qubits.

6.1.2 Results with experimental distributions q

I designed a more interesting experiment by computing the XEB not with the ideal probability distribution, but with a distribution $q(x)$ which corresponds to the classical simulation of the quantum device. I obtained the XEB values by generating a large number of samples, and using the probability of each generated sample. From our first definition of $\mathcal{F}_{\text{XEB}}$ in **chapter 2**, and by using the definition of HOG(p,q) in equation (2.2), we express the fidelity as:

$$\mathcal{F}_{\text{XEB}} = 2^n \text{HOG} - 1 = 2^n \left(\frac{1}{n} \sum_x q(x) \right), \tag{6.2}$$

where n is the number of samples generated, and in the last equality I substituted the expected value $\mathbb{E}_p[q(x)]$ of the HOG, to the mean of the batch of generated samples $\{x\}$, which is what we measure experimentally.

The method `get_experimental_Hog(order, events=int(1e4))` returns the experimental HOG obtained after sampling the number of events given as an argument. After generating a single outcome, I store its probability in an array that gets returned after all the samples have been generated. This is a good way to obtain the HOG since we only have to take the mean value of such an array.

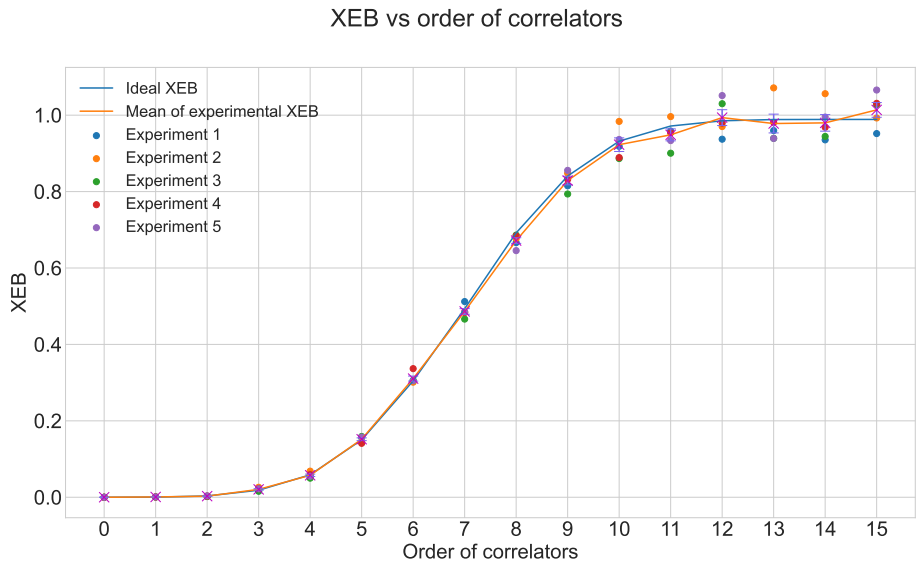

Figure 6.2: **Experimental XEB versus maximum order of correlators.** The blue line shows the ideal XEB for a random quantum circuit for $n = 15$ qubits. On the other hand, the plotted orange line is the mean of the experimental XEB obtained in 5 different experiments. Each experiment was run with a different random quantum circuit of 15 qubits, and their corresponding XEB values can be identified as the colour dots (green, red, orange, blue, purple). The magenta cross shows the mean of these 5 different experiments, with an error bar showing the standard error on the mean. We can see that there are fluctuations when increasing the number of correlators, as the standard errors increase

In order to obtain a good estimate, I computed the experimental XEB for 5 different random quantum circuits of 15 qubits and limited the order of correlators from 0 to 15, as can be seen in the x-axis of Figure 6.2. I obtained 1000 different samples whose probabilities were used to compute the corresponding XEB. I show plotted the mean of the experimental values and the standard error on the mean. Moreover, I plot in blue the fidelity values of a different random quantum circuit using its ideal distribution for comparison. We can see that the errors increase with the maximum order of the correlators, because there are more values from which to choose from, and I only ran 5 different experiments.

It can be seen, however, that the fidelity sometimes surpasses the value of 1 when we use the correlators of all orders. The expectation value of the fidelity is 1 because random quantum circuits follow the Porter-Thomas distribution very closely when the number of qubits is $n > 5$ [34]. The fluctuations are not very big and they could come from small error in the quantum circuits, or from statistical fluctuations. The proof of the bound of 1 is only on the limit of infinite generated samples, when $\text{HOG} = 2/2^n$ so that $\mathcal{F}_{\text{XEB}} = 1$.

There is a very valuable observation that can be retrieved from a detailed analysis of the graph. Due to our assumption of the noise model (measurement noise), the high-order coefficients could be exponentially suppressed by the noise. Therefore, we can avoid

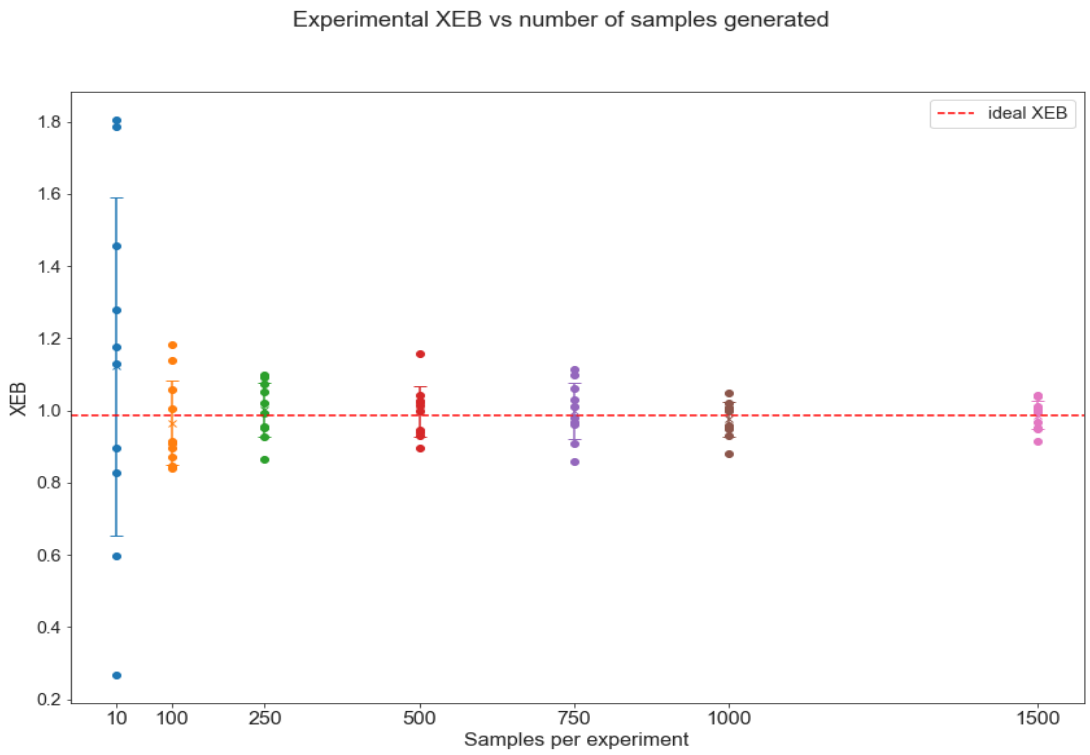

Figure 6.3: **Experimental XEB vs number of samples generated per experiment.** This figure shows the experimental XEB obtained for a quantum quantum circuit of 15 qubits with different number of samples. For each number of samples per experiment, N, I computed the XEB 10 different times. The values plotted are grouped in different colours depending on the number of outcomes generated to obtain them: from left to right I used 10, 100, 250, 500, 750, 1000 and 1500 samples to obtain the XEB. They are plotted together with their mean and the standard error on the mean. On the other hand, the red dotted line indicates the XEB's expected value using the ideal distribution p. The standard error on the mean and the dispersion of the points decreases very rapidly with the increase of samples taken per XEB, and we get a good estimate when the number of experiments surpasses the 1000 samples.

computing these higher order coefficients and make faster simulations with a reasonable fidelity. Looking at Figure 6.2 we can see that restricting the correlators to an order of $k = 9$ will give us a fidelity of ≈ 0.8, still a high value.

6.2 Increased accuracy with the number of samples

The value of the experimental XEB fluctuates with the number of outcomes sampled, because the ideal XEB corresponds to the expected value over all possible random quantum circuits. It is interesting to see how the experimental fidelity fluctuates with the number of samples, so that we can suggest how many samples are required to obtain a good estimate.

To compute the XEB values from Figure 6.3, I first obtain the probability distribution

Number of qubits	Fast sampling	Hybrid sampling	On-the-fly sampling
3	2.39 ms ± 215 µs	2.22 ms ± 369 µs	8.11 ms ± 980 µs
6	3.8 ms ± 333 µs	7.68 ms ± 555 µs	53.9 ms ± 16.4 ms
9	5.6 ms ± 1.45 ms	51.3 ms ± 1.42 ms	302 ms ± 40.9 ms
12	6.87 ms ± 722 µs	415 ms ± 5.16 ms	2.5 s ± 802 ms
15	7.75 ms ± 968 µs	3.32 s ± 136 ms	27.4 s ± 8.14 s

Table 6.1: Running time of the different implementations of the sampling algorithm

and the correlators from the simulation of a random quantum circuit of 15 qubits. Having this computed, I proceed to obtain the experimental XEB value N different generated samples, variable that changes in the x-axis. For each N, I generate N samples using the hybrid sampling algorithm, I get the experimental XEB and then I show a scatter plot with the different XEBs obtained. I also show the mean of the experimental XEB for the number of samples and the standard error on the mean. The red dotted line shows the XEB obtained from the ideal distribution, i.e., computed with the correlators from the simulation of another random quantum circuit of 15 qubits.

As Figure 6.3 shows, when we have less than 100 samples per computed XEB the values seem very dispersed, and the fluctuation is very high compared to the expected value. This is because in total we can have 2^n samples, but we will be generating a very small subset of those. Therefore, the events generated with the sampling algorithm will mainly be those with higher probability and we will be missing most of the other possible outcomes. On the other hand, when the number of samples N becomes greater than 1000, the experimental XEB becomes much closer to its true value.

6.3 Running time of the algorithms

This last section will be devoted to a comparison between the different implementations, which I performed on my laptop. I am using a Macbook Pro (13 inch, 2018) with a 2.7 GHz Quad-Core Intel Core i7 processor.

In table 6.1 I show the running time that it takes the different sampling approaches discussed in **chapter 4**. For the **fast sampling**, I did a learning phase to store all the 2^n different marginal probabilities, and then I generated samples. The results shown come from generating 100 samples 100 different times for a total of 10000 samples. I then repeated the experiment 7 times and obtained the mean and standard deviation of those executions, which is what I show in the table, with the format mean $\pm$ std. dev.

For the times that correspond to the **hybrid** and **on-the-fly** I ran the experiment a bit differently. First of all, I did not do a learning phase before generating samples, because this is part of what I consider the **fast** approach. Furthermore, after generating 100 samples for a given circuit, I cleared the marginals so that there are no values cached which will make the next sampling faster. I repeated this 100 different times, as I did for the **fast** approach, and repeated the experiment 7 times to obtain the mean and std. dev.

I did some of the experiments using the longjob command in DICE, so that I can run

the code for a long period of time without it stopping executing. Nevertheless, because more people have started to put jobs in the queue, the experiments took longer than usual and I had to repeat them in my laptop to be consistent and obtain the values above. In the future it would be useful to set up the experiments in a cloud platform such as Google Cloud. This way the values obtained could be easier to benchmark and more reliable than performing them on my own laptop. Moreover, I must say that to obtaining all the values using the **on-the-fly** approach took a whooping 906 minutes and 41.8 seconds. The **hybrid** sampling managed to give all the values from the table in only 44 minutes and 16.1 seconds, and the **fast** sampling took only 20 seconds to generate all the samples (without considering the learning phase). If we consider the time it took for executing the learning phase, the fast approach performed the experiment in 6 minutes and 54 seconds.

In conclusion, we easily see that the **fast** sampling algorithm is much faster than the other two approaches, and that the **on-the-fly** approach is the least performing one, which is also due to the fact that `get_correlators_for_marginal_slow` does not retrieve the correlators through indexing. Nevertheless, the learning phase in the **fast** sampling takes a sheer amount of time with increasing number of qubits. For 12 qubits it could learn the whole marginal tree in around 7 seconds. For 15 qubits, however, it took almost 6 minutes. Consequently, the **fast** algorithm should be used when the circuits do not have many qubits. In case the circuits are bigger, it seems a better option to consider the **hybrid** approach. In fact, we can do as discussed in **chapter 4** and incorporate a partial learning phase to this approach where we do not learn all the marginal probabilities, but a fraction of them.

Chapter 7

Conclusion

7.1 Discussion

I considered three different approaches to generate samples from a quantum circuit. All three require knowledge of the Fourier spectrum, or correlators, of the corresponding distribution to the quantum circuit. As explained, I used the chain rule to obtain the outcomes bit by bit using the corresponding conditional probabilities. To simplify the project, I considered the correlators as given by a "black- box", so that the report can focus entirely on the sampling algorithm. Specifically, I obtained all the correlators at once through the Fast Walsh-Hadamard transform, letting me access them on demand.

I gave the notion of sampling **on-the-fly**, where the required marginal probabilities were computed whenever needed for choosing a bit x_i. Later, I showed in the **hybrid** approach how reusing the marginals, and potentially adding a partial learning phase, would make the simulation faster. Lastly, with the **fast** approach I showed that generating a sample takes linear time in the number of qubits, with all the marginals pre-computed.

From the analysis and experiments of the algorithms, it was easy to see that pre-computing the marginals gave a significant speedup to generate samples. Nevertheless, due to the exponential time required for computing such values, it was concluded that a complete learning phase would not be feasible with bigger-size circuits. Instead, the learning could be performed partially, and the sampling executed with the hybrid algorithm.

The sampling algorithm, however, was more complicated to test. After finding a bug to obtain the necessary correlators for the marginals, I changed the implementation to index the correct values. The problem persisted when the order of these coefficients was limited, but this was a standard issue whose solution I implemented.

On balance, **the three different approaches of the sampling algorithm were successfully implemented using Fourier decomposition**. This allowed to simulate quantum circuits on a laptop, and to limit the order of the coefficients. Because of how the correlators were obtained, the algorithm cannot simulate bigger ($n > 23$), more complex circuits. Nevertheless, it can be adapted for computing the Fourier coefficients with TN contractions, and to execute in parallel to run the Sycamore circuits. With those

improvements, the sampling algorithm will be able to show whether Google's XEB test can be "hacked" by computing, in a timely manner, a fidelity $\mathcal{F}_{\text{XEB}} \approx 0.2\%$ through limiting the order of the correlators.

7.2 Future work

There are numerous ways in which the ideas discussed in this report can be modified and improved. Undoubtedly, the most critical change that can be performed is the computation of the correlators. The current method requires us to store all the correlators, which will not be possible as the space complexity increases exponentially with the number of qubits. Moreover, the time complexity of computing a marginal does not change, even if the order of the correlators is limited. There are several techniques to obtain these correlators more efficiently, but using tensor network contractions will probably be the best approach. TN contractions let us compute only the correlators that we are interested in, and it will also be possible to limit their order. The next step would be to determine the correlators needed to reach a certain fidelity with such implementation. Then, it would be exciting to see whether we can "hack" the sampling of the intractable Sycamore circuits.

This is incredibly non-trivial, and the implementation will have to change to perform computations on circuits with n > 50. Ideally, after implementing TN contractions, it should be modified so that it can be executed in the EPCC, the Edinburgh Parallel Computing Centre. This will require parallelising the code and migrating it from Python to C++. Moreover, the generation of Sycamore circuits should be adapted to include the intractable sequence ABCDCDAB, as discussed in **chapter 2**. With the computational power of the EPCC, it could be possible to simulate these Sycamore circuits and try to hack Google's XEB test.

Last but not least, it would be interesting to see a detailed analysis of how a partial learning phase affects the running time of the sampling algorithm. Specifically, the level at which the marginal tree can be pruned so that sampling is faster, but the learning phase does not require a large amount of time.

Bibliography

[1] Claude Cohen-Tannoudji, Bernard. Diu, and Franck. Laloe. *Quantum mechanics Vol.1*. Wiley, 1978.

[2] Henry O. Kunz. "On the Equivalence Between One-Dimensional Discrete Walsh-Hadamard and Multidimensional Discrete Fourier Transforms". In: *IEEE Transactions on Computers* C-28 (1979), pp. 267–268.

[3] Paul Benioff. "The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines". In: *Journal of Statistical Physics* 22.5 (1980), pp. 563–591.

[4] Yuri Manin. "Computable and Uncomputable". In: *Sovetskoye Radio* (1980).

[5] "Simulating physics with computers". In: *International Journal of Theoretical Physics* 21.6/7 (1981).

[6] Charles H. Bennett and Stephen J. Wiesner. "Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states". In: *Phys. Rev. Lett.* 69 (20 Nov. 1992), pp. 2881–2884.

[7] Lov K. Grover. "A fast quantum mechanical algorithm for database search". In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing*. Association for Computing Machinery, July 1, 1996.

[8] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM Journal on Computing* 26.5 (Oct. 1997), pp. 1484–1509.

[9] Michael A. Nielsen and Isaac L. Chuang. "Quantum Computation and Quantum Information". In: Cambridge University Press, Oct. 2000, p. 13.

[10] Charles H. Bennett et al. "Remote State Preparation". In: *Phys. Rev. Lett.* 87 (7 July 2001), p. 077902.

[11] F. L. Yan and X. Q. Zhang. "A scheme for secure direct communication using EPR pairs and teleportation". In: *The European Physical Journal B - Condensed Matter and Complex Systems* 41.1 (Sept. 1, 2004), pp. 75–78.

[12] Igor L. Markov and Yaoyun Shi. "Simulating Quantum Computation by Contracting Tensor Networks". In: *SIAM Journal on Computing* 38.3 (2008), pp. 963–981.

[13] M. Van den Nest. *Classical simulation of quantum computation, the Gottesman-Knill theorem, and slightly beyond*. 2009.

[14] S Aaronson and A Arkhipov. *Proceedings of the forty-third annual ACM symposium on Theory of computing*. 2011.

[15] Charles H. Bennett and Gilles Brassard. "Quantum cryptography: Public key distribution and coin tossing". In: *Theoretical Computer Science* 560 (Dec. 2014), pp. 7–11.

[16] Gil Kalai and Guy Kindler. "Gaussian Noise Sensitivity and BosonSampling". In: *arXiv:1409.3093 [quant-ph]* (Nov. 8, 2014).

[17] Anthony Leverrier and Raúl García-Patrón. "Analysis of circuit imperfections in BosonSampling". In: *arXiv:1309.4687 [quant-ph]* (Nov. 5, 2014).

[18] Varun Narasimhachar and Gilad Gour. "Low-temperature thermodynamics with quantum coherence". In: *Nature Communications* 6.1 (July 2015), p. 7689.

[19] Scott Aaronson and Lijie Chen. "Complexity-Theoretic Foundations of Quantum Supremacy Experiments". In: *arXiv:1612.05903 [quant-ph]* (Dec. 26, 2016).

[20] Michael J. Bremner, Ashley Montanaro, and Dan J. Shepherd. "Achieving quantum supremacy with sparse and noisy commuting quantum computations". In: *Quantum* 1 (Apr. 2017), p. 8.

[21] Sergio Boixo et al. "Characterizing quantum supremacy in near-term devices". In: *Nature Physics* 14.6 (Apr. 2018), pp. 595–600.

[22] Igor L. Markov et al. *Quantum Supremacy Is Both Closer and Farther than It Appears*. 2018.

[23] Frank Arute et al. "Quantum supremacy using a programmable superconducting processor". In: *Nature* 574.7779 (Oct. 24, 2019), pp. 505–510.

[24] Frank Arute et al. "Quantum supremacy using a programmable superconducting processor". In: *Nature* 574.7779 (Oct. 2019), pp. 505–510.

[25] Román Orús. "Tensor networks for complex quantum systems". In: *Nature Reviews Physics* 1.9 (Sept. 2019), pp. 538–550.

[26] Edwin Pednault et al. "Leveraging Secondary Storage to Simulate Deep 54-qubit Sycamore Circuits". In: *arXiv:1910.09534 [quant-ph]* (Oct. 22, 2019).

[27] Edwin Pednault et al. *On "Quantum Supremacy"*. IBM. Oct. 2019.

[28] John Preskill. *Why I Called It 'Quantum Supremacy'*. Quanta Magazine. Feb. 10, 2019.

[29] Emanuele Viola. "Algorithms and Complexity". In: (2019), p. 20.

[30] Philip Ball. "Physicists in China challenge Google's 'quantum advantage'". In: *Nature* 588.7838 (Dec. 3, 2020), pp. 380–380.

[31] Ivan B. Djordjevic. “Secure, Global Quantum Communications Networks”. In: *2020 22nd International Conference on Transparent Optical Networks (ICTON)*. 2020, pp. 1–5.

[32] Gil Kalai. *Photonic Huge Quantum Advantage ???* Dec. 5, 2020. (Visited on 10/22/2021).

[33] Gil Kalai. “The argument against quantum computers, a very short introduction”. In: (2020).

[34] Sean Mullane. *Sampling random quantum circuits: a pedestrian’s guide*. 2020.

[35] Shi-Ju Ran et al. “Tensor Network Contractions”. In: *Lecture Notes in Physics* (2020).

[36] J. Tangpanitanon and D. G. Angelakis. “Many-body physics and quantum simulations with strongly interacting photons”. In: (2020), pp. 169–215.

[37] Han-Sen Zhong et al. “Quantum computational advantage using photons”. In: *Science* 370.6523 (2020), pp. 1460–1463.

[38] Yiqing Zhou, E. Miles Stoudenmire, and Xavier Waintal. “What limits the simulation of quantum computers?” In: *Physical Review X* 10.4 (Nov. 23, 2020).

[39] Arjun Berera and Luigi Del Debbio. *Quantum Mechanics*. Cambridge University Press, 2021.

[40] Francesco Bova, Avi Goldfarb, and Roger Melko. “Quantum Computing Is Coming. What Can It Do?” In: (July 2021).

[41] Jacob Bulmer et al. “The Boundary for Quantum Advantage in Gaussian Boson Sampling”. In: (Aug. 2021).

[42] Song Cheng et al. “Simulating noisy quantum circuits with matrix product density operators”. In: *Physical Review Research* 3.2 (Apr. 2021).

[43] Alexander M. Dalzell, Nicholas Hunter-Jones, and Fernando G. S. L. Brandão. “Random quantum circuits transform local noise into global white noise”. In: (2021).

[44] Craig Gidney and Martin Ekerå. “How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits”. In: *Quantum* 5 (Apr. 15, 2021), p. 433.

[45] “Light on quantum advantage”. In: *Nature Materials* 20.3 (Mar. 2021), pp. 273–273.

[46] Yong (Alexander) Liu et al. “Closing the "Quantum Supremacy" Gap: Achieving Real-Time Simulation of a Random Quantum Circuit Using a New Sunway Supercomputer”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’21. New York, NY, USA, 2021.

[47] Feng Pan, Keyang Chen, and Pan Zhang. *Solving the sampling problem of the Sycamore quantum supremacy circuits*. 2021.

[48] Feng Pan and Pan Zhang. *Simulating the Sycamore quantum supremacy circuits*. 2021.

[49] A. S. Popova and A. N. Rubtsov. "Cracking the Quantum Advantage threshold for Gaussian Boson Sampling". In: *arXiv:2106.01445 [quant-ph]* (Dec. 21, 2021).

[50] Yulin Wu et al. "Strong quantum computational advantage using a superconducting quantum processor". In: *arXiv:2106.14734 [quant-ph]* (June 28, 2021).

[51] David Amaro et al. "Filtering variational quantum algorithms for combinatorial optimization". In: *Quantum Science and Technology* 7.1 (Jan. 2022), p. 015021.

[52] Dylan Herman et al. "A Survey of Quantum Computing for Finance". In: *arXiv:2201.02773 [quant-ph, q-fin]* (Jan. 18, 2022).

[53] Sanjay Rout. *12 Ways Quantum Computing Can Radically Change The World*. Openexo. Mar. 2022.

[54] Qingling Zhu et al. "Quantum computational advantage via 60-qubit 24-cycle random circuit sampling". In: *Science Bulletin* 67.3 (2022), pp. 240–245.

Appendix A

Hardcoded example of 3 qubits

Following section 5.1, I showed the calculation of the marginal probabilities up to 2 qubits for an example with:

$$\texttt{probabilities} = [0.19, 0.07, 0.09, 0.23, 0.06, 0.13, 0.22, 0.01]$$

$$\texttt{correlators} = [1, 0.12, -0.1, -0.02, 0.16, -0.16, -0.02, 0.54]\,.$$

In this appendix, I continue the computation of the marginals for 3 qubits. We will see that they are equivalent to the `probabilities` array because I am not limiting the order of the correlators.

$$\begin{aligned}
p(000) &= \hat{p}(000) + \hat{p}(001) + \hat{p}(010) + \hat{p}(011) \\
&\quad + \hat{p}(100) + \hat{p}(101) + \hat{p}(110) + \hat{p}(111) \\
&= \frac{1}{8}[C(000) + C(001) + C(010) + C(011) \\
&\quad + C(100) + C(101) + C(110) + C(111)] \\
&= \frac{1}{8}[1 + 0.12 - 0.1 - 0.02 + 0.16 - 0.16 - 0.02 + 0.54] \\
&= 0.19
\end{aligned}$$

$$\begin{aligned}
p(001) &= \frac{1}{8}[C(000) - C(001) + C(010) - C(011) \\
&\quad + C(100) - C(101) + C(110) - C(111)] \\
&= \frac{1}{8}[1 - 0.12 - 0.1 + 0.02 + 0.16 + 0.16 - 0.02 - 0.54] \\
&= 0.07
\end{aligned}$$

$$\begin{aligned}
p(010) &= \frac{1}{8}[C(000) + C(001) - C(010) - C(011) \\
&\quad + C(100) + C(101) - C(110) - C(111)] \\
&= \frac{1}{8}[1 + 0.12 + 0.1 + 0.02 + 0.16 - 0.16 + 0.02 - 0.54] \\
&= 0.09
\end{aligned}$$

$$
\begin{aligned}
p(011) &= \frac{1}{8}[C(000) - C(001) - C(010) + C(011) \\
&\quad + C(100) - C(101) - C(110) + C(111)] \\
&= \frac{1}{8}[1 - 0.12 + 0.1 - 0.02 + 0.16 + 0.16 + 0.02 + 0.54] \\
&= 0.23
\end{aligned}
$$

$$
\begin{aligned}
p(100) &= \frac{1}{8}[C(000) + C(001) + C(010) + C(011) \\
&\quad - C(100) - C(101) - C(110) - C(111)] \\
&= \frac{1}{8}[1 + 0.12 - 0.1 - 0.02 - 0.16 + 0.16 + 0.02 - 0.54] \\
&= 0.06
\end{aligned}
$$

$$
\begin{aligned}
p(101) &= \frac{1}{8}[C(000) - C(001) + C(010) - C(011) \\
&\quad - C(100) + C(101) - C(110) + C(111)] \\
&= \frac{1}{8}[1 - 0.12 - 0.1 + 0.02 - 0.16 - 0.16 + 0.02 + 0.54] \\
&= 0.13
\end{aligned}
$$

$$
\begin{aligned}
p(110) &= \frac{1}{8}[C(000) + C(001) - C(010) - C(011) \\
&\quad - C(100) - C(101) + C(110) + C(111)] \\
&= \frac{1}{8}[1 + 0.12 + 0.1 + 0.02 - 0.16 + 0.16 - 0.02 + 0.54] \\
&= 0.22
\end{aligned}
$$

$$
\begin{aligned}
p(111) &= \frac{1}{8}[C(000) - C(001) - C(010) + C(011) \\
&\quad - C(100) + C(101) + C(110) - C(111)] \\
&= \frac{1}{8}[1 - 0.12 + 0.1 - 0.02 - 0.16 - 0.16 - 0.02 - 0.54] \\
&= 0.01
\end{aligned}
$$

Now we can easily see that the values computed are the same as with the initial distribution.