

Debugging Quantum Computing Experiments



Martin Tsvetomirov Marinov
s0932707

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

Abstract

The project aims to provide a software solution for design and simulation of Measurement Based Quantum Computing (MBQC) algorithms called patterns. The system is designed to be flexible and platform independent by using Java as an implementation language. It aims to be intuitive and easy to use by providing a graphical user interface which allows users to dynamically build algorithm graphs by dragging and clicking objects on the screen.

ACKNOWLEDGMENTS

I would like to thank my supervisor Elham Kashefi for offering theoretical and practical support and guidance during the course of the project. She was able to provide a comprehensible introduction to the world of Quantum Computing and Measurement Based Calculus. The knowledge received throughout the process assisted the development of the algorithms that the project is using. Elham also pointed out the possible automatic verification methods that were utilized to prove the correctness of the project. I really appreciate the extra time she spend on the regular weekly meetings.

I would also like to acknowledge Einar Pius' contribution to the manual testing process by providing examples and debugging the output data. He also kindly provided his quantum circuit translator program and guidance on how to interface it with a web-based simulator.

I would like to thank Flaviu Cipcigan for providing the theoretical background required to do the analysis on the actual optical quantum computing implementation experiments. His software also aided the verification of the project.

1 TABLE OF CONTENTS

2	Introduction	4
2.1	Goals	4
2.2	Achieved Goals	4
3	Theory	5
3.1	Quantum Mechanics	5
3.2	Quantum Computing Introduction.....	5
3.3	Quantum Computing.....	6
3.3.1	Qubit.....	6
3.3.2	Unitary Operators	7
3.3.3	Entanglement.....	8
3.3.4	Measurements	8
3.4	Measurement Based Quantum Computing Patterns.....	9
3.4.1	Measurement Calculus	10
3.4.2	Graphical Representation	11
4	Implementation of Simulator Engine	12
4.1	Representation	13
4.2	Operators.....	15
4.2.1	Preparation operator N.....	15
4.2.2	Preparation operator I	16
4.2.3	Entanglement operator E.....	16
4.2.4	Measurement operator M	16
4.2.5	Correction operators X and Z.....	16
4.2.6	Adding coefficients together operator	17
4.3	Running Simulations.....	17
4.3.1	Using the Library	17
4.3.2	Using a GUI.....	18
5	Implementation of Graphical Designer	20
5.1	Design	21

5.2	Translation to Mcalc	25
6	Symbolic Algebra System	26
6.1	Motivation	26
6.2	Object Oriented Design	27
6.2.1	MathExpression.....	27
6.2.2	MathNumber and MathSymbol	29
6.2.3	MathFract.....	30
6.2.4	MathFunction	30
6.3	MathsParser.....	30
7	Verification.....	31
7.1	Symbolic Algebra System.....	32
7.2	Experimental Data Analysis	34
7.2.1	Optical Implementation of Quantum Computing	34
7.2.2	Simulation.....	35
7.2.3	Output	36
7.2.4	Results of verification.....	38
7.2.5	Conclusion	39
7.3	Semi-automated Procedure	40
7.3.1	Procedure	40
7.3.2	Implementation.....	41
7.3.3	Usage	42
7.3.4	Results	43
7.4	Manual Testing	44
8	Future Work.....	45
9	Bibliography	47
10	Appendix: User Guide.....	48

2 INTRODUCTION

2.1 GOALS

One of the main goals of the project is to serve as an assisted tool for theoretical research by helping to accelerate the process of running and verifying MBQC patterns. The graphical interface needs to be user friendly and the results produced should be accurate and consistent. It has to report its output in a useful and easy to understand form. It should allow for exploring properties of complex patterns such as whether they are deterministic or not. In case of non-deterministic patterns for example, it could give researchers an opportunity to trace back the place where non-determinism first occurred and explore different outcome branches.

Experimentalists could use it for verifying experimental results against theoretical predictions. MBQC allows for better verification than existing methods like quantum state tomography. The reason is that quantum state tomography can only check the end result while doing an MBQC simulation can yield intermediate results which could be used to “debug” experimental data. The software simulator was eventually used against real data produced in Quantum Computing experiments conducted in Vienna.

The final package should be easy to install and download from the web. It should not be hardware or OS dependent. The usage should be intuitive, requiring only general computer skills from the user.

2.2 ACHIEVED GOALS

The software package produced contains an excellent ecosystem allowing for Full MBQC simulation. It has tools that allow manipulation of system of qubits. It is fully extensible and uses Object-Oriented approach that allows for easy integration of new features and capabilities. The produced library for MBQC manipulation could be also used in 3rd party applications.

The graphical user interface (GUI) covers almost all of the initial set of goals. It is fully interactive and user friendly. The only requirement is that the user has a working installation of the Java Runtime Environment. Once the user downloads the executable jar file, they would only need to double click it to get the software package running.

MBQC patterns could be built graphically or they could be input as MCalc commands. Then they could be simulated with a click of a button. The symbolic mathematics manipulation engine allows for input and output of mathematical expressions, enhancing the user experience and improving the accuracy over using floating point arithmetic.

The system has been proven to produce the expected output consistently and accurately for the designed test procedures. It has been successfully used to verify real experimental data. Furthermore, the analysis of experimental data could be controlled from the GUI allowing for the support of new types of experiments.

The project is available to download from <https://github.com/martinmarinov/HonProj> and is released under the **GPL** license. The binary .jar file could be downloaded pre-compiled from the web-page and run on a user machine by simply double clicking it.

3 THEORY

3.1 QUANTUM MECHANICS

As you might know particles exhibit what is called “wave-particle duality phenomenon”. This means that on macroscopic scale they behave like classical objects. For example we can’t see the wave function of a football bouncing around a pitch. But particles on the really small scale exhibit some weird phenomena.

This was first observed by Thomas Young who performed the famous double-slit experiment and showed that light photons can behave both like particles and waves. Later a separate branch of physics emerged trying to explain this behaviour called Quantum Mechanics. The underlying assumption is that each particle could be represented mathematically as a complex vector that evolves in time according to a wave law. A set of mathematical tools based on matrix calculus could be used to describe interactions between particles as well as changes of the intrinsic properties of the particle itself.

The wave-like nature of such objects means they possess some extraordinary capabilities. They could interfere with each other, they have a finite chance of penetrating through places where classical particles could not possibly exist in (like solid walls) called quantum tunnelling or they could get entangled. When two or more particles are entangled, this means that one cannot tell them apart and they behave like one entity. Therefore when a measurement is done on one of them, all of them are changed. Thus there’s no way to tell exactly what a measurement of a property of a quantum particle would yield. Rather a probability of getting a certain outcome could be mathematically inferred.

3.2 QUANTUM COMPUTING INTRODUCTION

Quantum Computing (QC) exploits Quantum Mechanics phenomena. Quantum systems are represented by complex vectors. Actions on such vectors can be done using matrices.

Since the number of coefficients required to represent a physical quantum system is 2^n where n is the number of qubits in the system, the computation complexity increases exponentially. This also makes simulating bigger quantum systems exponentially harder on classical computers.

Some algorithms that are based on finding optimizations may run faster on a quantum computer. This is because classical computer suffer from the familiar local-minimum problem that makes algorithms get stuck in suboptimal solutions. Quantum particles on the other hand exhibit the quantum tunnelling effect that allows them to penetrate through potential fields and therefore “break free” from such local minima. This has allowed for the development of efficient quantum algorithms like Shor's algorithm which can factorize integers in polynomial time. Other phenomena that don't exist classically like superposition and entanglement could also be utilized. There is still quite a lot of research going into this area to discover other advantages of quantum computing over classical one.

The most common model for Quantum Computing is the Quantum Circuit model. It allows the design of quantum algorithms by using Quantum Gates. They are similar to electronic gates that you would see in classical computers in the sense that they are used to “process” or “mix” signals in a certain way when they pass through them. The circuits actually evolve a system of quantum bits, called qubits in time in order to arrive to a certain state of the system that when measured would yield the desired output of the algorithm.

Measurement Based Quantum Computing (MBQC) is a more recent development. Unlike Quantum Circuit model, the computation is propagated by performing measurements on a group of entangled qubits rather than waiting for the system to evolve in time. This in theory make it more stable and adds the additional measurements to the process of obtaining a result. This means that intermediate results are available and a proper simulation could be able to verify them.

3.3 QUANTUM COMPUTING

3.3.1 Qubit

A quantum unit of information is called a quantum bit or a qubit. It could be measured in two basis vectors $|0\rangle$ and $|1\rangle$. Unlike digital bits which could be either in 0 or 1 state, a quantum bit could be in a linear superposition of the two basis vectors. Therefore a state of a quantum bit could be written as:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

Where α and β are complex coefficients. If the qubit is normalized $|\alpha|^2 + |\beta|^2 = 1$. When a measurement is made the wave function collapses and the qubit ends up in either $|0\rangle$ or $|1\rangle$ state. The probability of the qubit collapsing to the $|0\rangle$ state is $|\alpha|^2$ and the probability of the qubit collapsing to the $|1\rangle$ state is $|\beta|^2$.

A system of two qubits in states $|\psi_1\rangle = \alpha|0\rangle + \beta|1\rangle$ and $|\psi_2\rangle = \gamma|0\rangle + \delta|1\rangle$ could be written as

$$|\psi_1\rangle \otimes |\psi_2\rangle = \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle$$

Where the first element in the bra-ket is for the first qubit and the second one is for the second qubit. The \otimes symbol is called a tensor product. This representation could be generalized for n qubits. Also note that $|00\rangle$ notation is equivalent to the $|0_1 0_2\rangle$ where the subscripts (indicating the id of the qubit) are dropped because they are trivial in this case. For a state $|\psi\rangle = a|0\rangle + b|1\rangle$ to be normalized, it would mean $\langle\psi|\psi\rangle = |a|^2 + |b|^2 = 1$. Since the basis vectors are normalized, the following identities apply $\langle 0_i | 0_i \rangle = 1$ and $\langle 1_i | 1_i \rangle = 1$. Also the basis vectors are orthogonal, therefore $\langle 0_i | 1_i \rangle = 0$ and $\langle 1_i | 0_i \rangle = 0$.

3.3.2 Unitary Operators

Unitary operators are matrices that act on a state vector to produce another state vector. These are used in Quantum Mechanics to drive a computation or do any other changes to states.

3.3.2.1 Single Qubit

Operators that act on a single qubits are 2x2 matrices. Examples of such operators are the Pauli matrices

$$X(|\psi\rangle) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} (\alpha|0\rangle + \beta|1\rangle) = \beta|0\rangle + \alpha|1\rangle$$

$$Z(|\psi\rangle) = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} (\alpha|0\rangle + \beta|1\rangle) = \alpha|0\rangle - \beta|1\rangle$$

3.3.2.2 Two Qubits

Operators that act on two qubits are 4x4 matrices. An example is the ΛZ (controlled-Z) operator:

$$\begin{aligned} \Lambda Z(|\psi_1\rangle \otimes |\psi_2\rangle) &= \Lambda Z(\alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle) = \\ &\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} (\alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle) \\ &= \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle - \beta\delta|11\rangle \end{aligned}$$

3.3.3 Entanglement

Two qubits are in entangled state when they cannot be separated into two states tensored together. A simple example of a system of two entangled qubits has the equation

$$b|00\rangle + c|11\rangle$$

We cannot find two states $|\psi_1\rangle$ and $|\psi_2\rangle$ so that $|\psi_1\rangle \otimes |\psi_2\rangle$ is equal to $b|00\rangle + c|11\rangle$. Furthermore, as a sequence, one cannot measure the two qubits independently. Imagine that a measurement of the first qubit yields the $|0\rangle$ state. This would mean that the second qubit would be in the $|0\rangle$ state as well and vice versa. MBQC uses the entanglement operator $\wedge Z$ (controlled-Z)

3.3.4 Measurements

Measurements are not unitary. They are probabilistic. A measurement is simply a projection to orthogonal basis.

In MBQC, when a measurement is made, the system can take two different branches. The definition of measurement M_i^α onto a qubit i is as used in MBQC is:

$$|+\alpha\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{i\alpha}|1\rangle)$$

$$|-\alpha\rangle = \frac{1}{\sqrt{2}}(|0\rangle - e^{i\alpha}|1\rangle)$$

Where $\alpha \in [0, 2\pi]$ is called the angle of measurement. The output of the measurement is called signal s_i and $s_i = 0$ if the state collapses to the branch $|+\alpha\rangle$ or $s_i = 1$ if the state collapses to the branch $|-\alpha\rangle$.

A side effect of performing a measurement onto a qubit is that the qubit is “being destroyed”. Since the basis vectors $|0\rangle$ and $|1\rangle$ are orthogonal and have a length of 1 (see 3.3.1), it follows that a measurement onto a qubit will yield a scalar value. For example, a $|+\alpha\rangle$ measurement onto $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ is

$$\begin{aligned} \frac{1}{\sqrt{2}}(\langle 0| + e^{i\alpha}\langle 1|) \otimes (\alpha|0\rangle + \beta|1\rangle) &= \frac{1}{\sqrt{2}}(\alpha\langle 0|0\rangle + \beta\langle 0|1\rangle + ae^{i\alpha}\langle 1|0\rangle + \beta e^{i\alpha}\langle 1|1\rangle) \\ &= \frac{1}{\sqrt{2}}(\alpha \times 1 + \beta \times 0 + ae^{i\alpha} \times 0 + \beta e^{i\alpha} \times 1) = \frac{1}{\sqrt{2}}(\alpha + \beta e^{i\alpha}) \end{aligned}$$

Which is a scalar value because the qubit has been measured and is not a part of the system anymore. The same principle applies for systems of more than one qubits. For

example, a $|+_0\rangle$ measurement of the first qubit onto a system of two qubits $|0_10_2\rangle + |0_11_2\rangle + |1_10_2\rangle + |1_11_2\rangle$ would yield

$$\begin{aligned} & \frac{1}{\sqrt{2}} (\langle 0_1| + \langle 1_1|) \otimes (|0_10_2\rangle + |0_11_2\rangle + |1_10_2\rangle + |1_11_2\rangle) \\ &= \frac{1}{\sqrt{2}} (\langle 0_1|0_1\rangle \otimes |0_2\rangle + \langle 0_1|1_1\rangle \otimes |0_2\rangle + \langle 1_1|0_1\rangle \otimes |0_2\rangle + \langle 1_1|1_1\rangle \otimes |0_2\rangle \\ &+ \langle 0_1|0_1\rangle \otimes |1_2\rangle + \langle 0_1|1_1\rangle \otimes |1_2\rangle + \langle 1_1|0_1\rangle \otimes |1_2\rangle + \langle 1_1|1_1\rangle \otimes |1_2\rangle) \\ &= \frac{1}{\sqrt{2}} (|0_2\rangle + |0_2\rangle + |1_2\rangle + |1_2\rangle) = \frac{2}{\sqrt{2}} (|0_2\rangle + |1_2\rangle) \end{aligned}$$

Therefore in the end of the measurement only the second qubit “survives”.

3.4 MEASUREMENT BASED QUANTUM COMPUTING PATTERNS

The goal of the project is to simulate MBQC patterns. A pattern is the equivalent to an algorithm in classical programming. Patterns are series of preparation (N), entanglement (E), measurement (M) and correction (X or Z) operations as defined above. Preparation means that we put a qubit in the so called $|+\rangle$ state which is defined as:

$$|+\rangle = \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle)$$

When running a pattern, we may create a certain number of qubits. The pattern will tell us which of those to put in the $|+\rangle$ state, while the remaining ones would be our input qubits (similar to arguments of a method in a programming language). Then some of the qubits are entangled according to the pattern. Afterwards certain qubits are measured again as defined in the pattern. The result is being corrected according to the corrections specified in the pattern.

X and Z Pauli corrections could be applied. They could be used to achieve deterministic patterns. They can depend on signals. If an X correction on qubit i depends on signal s , this is written as X_i^s . Similarly for a Z correction it would look like Z_i^s . This could be interpreted as if the $s = 0$, no correction should be applied or mathematically $Z_i^0 = X_i^0 = I$ (identity).

Measurements could also depend on signals. In particular, a measurement can depend on two signals. Those dependencies change the measurement angle. So a measurement that depends on the signals t , s and α is written as:

$${}^t[M_i^\alpha]^s = M_i^{(-1)^s\alpha+t\pi}$$

Signal addition is performed by taking the modulus of the sum. For example if $s = s_1 + s_2 + s_3$ and $s_1 = 1, s_2 = 1$ and $s_3 = 1$, then $s = (1+1+1) \bmod 2 = 3 \bmod 2 = 1$. Also a measurement is destructive in the sense that a qubit cannot be measured twice.

Measurements create so called “branches”. This is precisely because each measurement can be projected either using $|+\alpha\rangle$ or $|-\alpha\rangle$. Therefore each measurement creates two possible outcomes. For each of those outcomes the next measurement will produce 2 more therefore there would be now 4 possible paths that the computation can take called “branches”. In general if there are n measurements in a pattern, there would be 2^n possible branches. When a pattern is being run, it is usually practically feasible to compute only one of them. A branch is usually labeled as an array of integers b_1, b_2, \dots, b_n . Where b_i means that for the i th measurement you take the projection $|+\alpha\rangle$ if $b_i = 1$, and the projection $|-\alpha\rangle$ if $b_i = 0$.

3.4.1 Measurement Calculus

(a.k.a. MCalc) Formally a pattern may be written as a sequence of commands:

- preparation commands N_i – prepares qubit i in state $|+\rangle$
- entanglement commands E_{ij} – entangles qubits i and j by performing $\wedge Z$ onto i and j
- measurement commands ${}^t[M_i^\alpha]^s$ – performs measurement on qubit i
- correction commands X_i^S and Z_i^S – performs Pauli operators X and Z on qubit i

Qubits that are not prepared in the $|+\rangle$ state are called input qubits. Qubits that are not measured are called output qubits. After the pattern is run, the only remaining qubits in the system are the output qubits (since you can’t measure a qubit twice). They give the output of the pattern.

A famous pattern is the so-called quantum teleportation. It takes the value of an input qubit and “teleports” it to the output qubit.

$$X_3^{S_2} Z_3^{S_1} M_2^0 M_1^0 E_{23} E_{12} N_3 N_2$$

“Running” such a pattern is straightforward (keep in mind that actions are applied from right to left). For simplicity we will compute the 1, 1 branch. This means that both measurements will project to $|+\alpha\rangle$.

- We start with 3 qubits: $|\psi_1\rangle = a|0_1\rangle + b|1_1\rangle$, $|\psi_2\rangle = c|0_2\rangle + d|1_2\rangle$ and $|\psi_3\rangle = f|0_3\rangle + g|1_3\rangle$
- Execute N_2 : put 2nd qubit in state $|+\rangle$ i.e. putting $c = \frac{1}{\sqrt{2}}$ and $d = \frac{1}{\sqrt{2}}$
- Execute N_3 : put 3rd qubit in state $|+\rangle$ i.e. putting $f = \frac{1}{\sqrt{2}}$ and $g = \frac{1}{\sqrt{2}}$
- Now our system is $|\psi_1\rangle \otimes |\psi_2\rangle \otimes |\psi_3\rangle =$

$$\frac{1}{2}(a|000\rangle + a|001\rangle + a|010\rangle + a|011\rangle + b|100\rangle + b|101\rangle + b|110\rangle + b|111\rangle)$$

- Execute E_{12} : Now our system is

$$\frac{1}{2}(a|000\rangle + a|001\rangle + a|010\rangle + a|011\rangle + b|100\rangle + b|101\rangle - b|110\rangle - b|111\rangle)$$

- Execute E_{23} : Now our system is

$$\frac{1}{2}(a|000\rangle + a|001\rangle + a|010\rangle - a|011\rangle + b|100\rangle + b|101\rangle - b|110\rangle + b|111\rangle)$$

- Execute M_1^0 taking the $|+\alpha\rangle$ branch: i.e. multiply by $\frac{1}{\sqrt{2}}(\langle 0_1| + \langle 1_1|)$. Since we took the $|+\alpha\rangle$ branch, $s_1 = 0$. Now our system is

$$\frac{\sqrt{2}}{4}(a|0_2 0_3\rangle + a|0_2 1_3\rangle + a|1_2 0_3\rangle - a|1_2 1_3\rangle + b|0_2 0_3\rangle + b|0_2 1_3\rangle - b|1_2 0_3\rangle + b|1_2 1_3\rangle)$$

- Execute M_2^0 taking the $|+\alpha\rangle$ branch: i.e. multiply by $\frac{1}{\sqrt{2}}(\langle 0_2| + \langle 1_2|)$. Since we took the $|+\alpha\rangle$ branch, $s_2 = 0$. Now our system is

$$\frac{1}{4}(a|0_3\rangle + a|1_3\rangle + a|0_3\rangle - a|1_3\rangle + b|0_3\rangle + b|1_3\rangle - b|0_3\rangle + b|1_3\rangle)$$

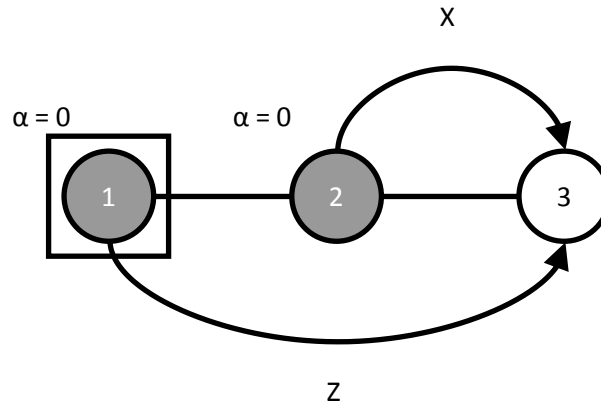
- Since $s_1 = 0$ and $s_2 = 0$ we don't execute $X_3^{s_2}$ or $Z_3^{s_1}$

Therefore after simplification we end up with a value of the output qubit $\frac{1}{2}(a|0_3\rangle + b|1_3\rangle)$, if we normalize the final state to 1, the value would yield $a|0_3\rangle + b|1_3\rangle$. Remember we actually started with a value of the input qubit $a|0_1\rangle + b|1_1\rangle$. Therefore it seems like the state of the first qubit has been “teleported” into the third one after the execution of the pattern.

If we were to compute another branch 0, 0 or 0, 1 or 1, 0 we will see that we will end up with the same result. This means that the pattern is deterministic. The X and Z corrections that are done as last step are important in this case since either s_1 or s_2 (or even both of them) may be 1. They make sure that all branches will give the same result. If the X and Z corrections weren't defined in the pattern, the different branches would yield different results therefore producing a non-deterministic pattern.

3.4.2 Graphical Representation

The teleportation example could be represented graphically like this:



The rules for building and interpreting such a graph are:

- Each qubit is represented as a circle. If the circle is filled up, this means the qubit is being measured and the measurement angle will be indicated on top of it. If there's a square around it, it means that the qubit is an input qubit.
- A line connecting two qubits represents entanglement operation.
- An arrow with X on top pointing to a non-output qubit means that the measurement that is being done onto the qubit at the end of the arrow has a s dependency on the signal of the qubit at the start of the arrow.
- An arrow with Z on top pointing to a non-output qubit means that the measurement that is being done onto the qubit at the end of the arrow has an t dependency on the signal of the qubit at the start of the arrow.
- An arrow with X on top pointing to an output qubit means that the output qubit must undergo an X correction with dependency on the signal of the qubit at the start of the arrow.
- An arrow with Z on top pointing to an output qubit means that the output qubit must undergo a Z correction with dependency on the signal of the qubit at the start of the arrow.

4 IMPLEMENTATION OF SIMULATOR ENGINE

Since this is the first time full MBQC simulator is designed, a new algorithm for representing and storing the qubits needed to be developed. The whole algorithm is based on methods that are used when patterns are evaluated manually with pen and paper. Most of the simplifications, assumptions and optimizations done by hand are used as well.

The classes for representing and analyzing a system of quantum bits are located in the *martin.quantum* package. They utilize the symbolic algebra engine from the *martin.math* package in order to perform simple mathematical operations with coefficients. Quantum bit manipulations inside a system are done by classes in the *martin.operators* package.

4.1 REPRESENTATION

As obvious from 3.3.1 storing a system of n qubits requires storing 2^n coefficients. Therefore the memory requirements increases exponentially. The computational complexity also increases as 2^n . The class *martin.quantum.SystemMatrix* initializes an array of 2^n *martin.math.MathsItem*.

The reason *MathsItems* are used instead of plain floating point numbers is because although it may seem inefficient in first sight, it saves computation. The reason is that once a pattern is evaluated for a generic input, it does not need to be reevaluated again for a specific instance. This means that, if we simulate the pattern for inputs a, b, c and d , later we could manually set the values to, for example, $a = 1, b = 0, c = 0.5$ and $d = 1$ and evaluate the resulting mathematical expression rather than simulate again the whole pattern. In the same time we achieve a greater precision than using plain floating point arithmetic. It is **known** that each operation with floating arithmetic introduces an error. For a pattern, a big number of operations need to be done. Since the mathematical system is being utilized in our case, the simplified final expression would have less mathematical operations to undertake. For example, if we end up with a result \sqrt{a} , and we put $a = 2$, only one mathematical operation would be done to get the real result. If we were simulating the whole pattern with full floating point arithmetic, many multiplications and additions would have been done to the initial value, therefore introducing a bigger error in the output.

In order to access each coefficient in the matrix, one can use the binary representation of the index of the array. For example in a 3 qubit system, an id of 3 would correspond to $|011\rangle$. This representation is key for performance since it presents a sequential access to the data. For example, if we have a system of three qubits

$$a|000\rangle + b|001\rangle + c|010\rangle + d|011\rangle + e|100\rangle + f|101\rangle + g|110\rangle + h|111\rangle$$

It will be represented as an array

0	1	2	3	4	5	6	7
a	b	c	d	e	f	g	h

Because $0 = 000$ in binary, $1 = 001$, $2 = 010$, etc.

For example, a system of two qubits $|\psi_1\rangle = \alpha|0\rangle + \beta|1\rangle$ and $|\psi_2\rangle = \gamma|0\rangle + \delta|1\rangle$, which could be also written as $\alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle$ ¹ would have a *SystemMatrix* representation

0	1	2	3
$\alpha\gamma$	$\alpha\delta$	$\beta\gamma$	$\beta\delta$

This representation has a few caveats if the system has qubits that have been already measured. As mentioned before, a measurement “destroys” a qubit². Therefore the number of coefficients drops from 2^n to 2^{n-1} where n is the number of qubits before the measurement. A naïve way of dealing with this problem would be to allocate a new *SystemMatrix* for each measurement done and populate it with the correct coefficients. This would be quite wasteful, since reallocation would be too CPU intensive. Furthermore when the allocation is taking place, along with the 2^n coefficients which have been already allocated in RAM, a new allocation of 2^{n-1} would be made, increasing the total RAM requirement for the simulator dramatically.

The solution to this problem is to preserve the original *SystemMatrix* but keep track of which qubits have been already measured. This is ok as long as extra care is taken when operators are being applied to the whole system. The reason is that now a vector that represents a certain qubit configuration could be split between several coefficients. To illustrate the problem, let’s start with a three qubit system.

$ 000\rangle$	$ 001\rangle$	$ 010\rangle$	$ 011\rangle$	$ 100\rangle$	$ 101\rangle$	$ 110\rangle$	$ 111\rangle$
a	b	c	d	e	f	g	h

Let’s measure the second qubit with $| -_0 \rangle = \frac{1}{\sqrt{2}}(|0_2\rangle - |1_2\rangle)$. The resulting system would be

$ 0_1 0_3\rangle$	$ 0_1 1_3\rangle$	$ 0_1 0_3\rangle$	$ 0_1 1_3\rangle$	$ 1_1 0_3\rangle$	$ 1_1 1_3\rangle$	$ 1_1 0_3\rangle$	$ 1_1 1_3\rangle$
$\frac{1}{\sqrt{2}}\mathbf{a}$	$\frac{1}{\sqrt{2}}\mathbf{b}$	$-\frac{1}{\sqrt{2}}\mathbf{c}$	$-\frac{1}{\sqrt{2}}\mathbf{d}$	$\frac{1}{\sqrt{2}}\mathbf{e}$	$\frac{1}{\sqrt{2}}\mathbf{f}$	$-\frac{1}{\sqrt{2}}\mathbf{g}$	$-\frac{1}{\sqrt{2}}\mathbf{h}$

This is mathematically accurate and could be used for further computation. The caveat is that a vector has been spread into more than one cell. The meaning of the representation above is $\left(\frac{1}{\sqrt{2}}\mathbf{a} - \frac{1}{\sqrt{2}}\mathbf{c}\right)|0_1 0_3\rangle + \left(\frac{1}{\sqrt{2}}\mathbf{b} - \frac{1}{\sqrt{2}}\mathbf{d}\right)|0_1 1_3\rangle + \left(\frac{1}{\sqrt{2}}\mathbf{e} - \frac{1}{\sqrt{2}}\mathbf{g}\right)|1_1 0_3\rangle + \left(\frac{1}{\sqrt{2}}\mathbf{f} - \frac{1}{\sqrt{2}}\mathbf{h}\right)|1_1 1_3\rangle$. If all of the actions that we plan to do on this matrix

¹ See section 3.3.1

² See section 3.3.4

are multiplications, we could carry on as expected (because of the distributive property of multiplication that $a \times (b + c) = a \times b + a \times c$). Therefore if we made another measurement, for example, $|+_0\rangle = \frac{1}{\sqrt{2}}(|0_1\rangle + |1_1\rangle)$ on the first qubit, we would get

$ 0_3\rangle$	$ 1_3\rangle$	$ 0_3\rangle$	$ 1_3\rangle$	$ 0_3\rangle$	$ 1_3\rangle$	$ 0_3\rangle$	$ 1_3\rangle$
$\frac{1}{2}\mathbf{a}$	$\frac{1}{2}\mathbf{b}$	$-\frac{1}{2}\mathbf{c}$	$-\frac{1}{2}\mathbf{d}$	$\frac{1}{2}\mathbf{e}$	$\frac{1}{2}\mathbf{f}$	$-\frac{1}{2}\mathbf{g}$	$-\frac{1}{2}\mathbf{h}$

Extreme caution need to be taken when applying an operation that requires the full vector. An example of such operation is the probability of a state which is defined as the norm of the state vector $P = \langle \psi | \psi^* \rangle$. Such operation would require for the coefficients to be collected together. If we tried finding the norm of each of the coefficients above and adding them together, the result would be incorrect since each of them is meaningless by itself. Therefore before performing the computation, the system above is transformed to

$ 0_3\rangle$	$ 1_3\rangle$	$ 0_3\rangle$	$ 1_3\rangle$	$ 0_3\rangle$	$ 1_3\rangle$	$ 0_3\rangle$	$ 1_3\rangle$
$\frac{1}{2}\mathbf{a} - \frac{1}{2}\mathbf{c}$	$\frac{1}{2}\mathbf{b} - \frac{1}{2}\mathbf{d}$	0	0	$\frac{1}{2}\mathbf{e} - \frac{1}{2}\mathbf{g}$	$\frac{1}{2}\mathbf{f} - \frac{1}{2}\mathbf{h}$	0	0

Now the probability is simply the sum of the norm of all of the elements.

4.2 OPERATORS

All of the operators implement the interface *martin.operators.Operator* which declares a method for an operator to receive a *martin.quantum.SystemMatrix* and act upon it. The operator is responsible of taking into account the state of the system – i.e. which qubits have been already measured. They have direct access to the coefficients and could change them.

Generally an operator would iterate through all of the coefficients and multiply them by a certain factor. If an operator needs perform other operations it must ensure the *SystemMatrix* has all of its coefficients collected together as described in the previous chapter. The iteration through the array is index based and in order for the operator to act on the correct coefficient, the id should be analyzed using binary arithmetic.

4.2.1 Preparation operator N

Prepares a qubit i in a certain state defined by the two input coefficients a and b . The implementation iterates through all of the coefficients in the system. For each of them it multiplies it by either a or b according to whether the binary representation of the coefficient contains 0 or 1 at its i th place.

For example if we have a system of two qubits:

$$x|00\rangle + y|01\rangle + z|10\rangle - w|11\rangle$$

Running a preparation with a, b for the first qubit will give output:

$$ax|00\rangle + ay|01\rangle + bz|10\rangle - bw|11\rangle$$

4.2.2 Preparation operator I

Prepares the first n qubits of the system with the list of coefficients. Basically does the same job as N but applies for a larger number of qubits. The number of coefficients that need to be supplied is of course the number of input qubits squared. The rest of the qubits in the system are put in the $|+\rangle$ state.

This is the operator that initializes a *SystemMatrix* to the correct initial state. Therefore this is the operator that enforces the convention that the first n qubits of a MBQC pattern must be inputs and the rest are initialized in the $|+\rangle$ state. Keep in mind that there is a way of explicitly specifying the initial coefficients of a *SystemMatrix* by just using the appropriate constructor (not an operator).

4.2.3 Entanglement operator E

Entangles qubits i and j . It iterates through all of the coefficients in the system and if the binary representation of the id of the coefficients has a 1 on both the i th and the j th place, it flips the sign of the coefficient. The result of applying an E to a *SystemMatrix* is the $\wedge Z$ operation described in 3.3.3.

4.2.4 Measurement operator M

Performs a measurement with t and s dependency onto a qubit i . For every coefficient referring to the i th qubit it multiplies by the factor described in 3.3.4. It determines which projection to make (either $|+\alpha\rangle$ or $|-\alpha\rangle$) according to a value b which the user defines manually before running the measurement.

Therefore the system can use an array of supplied integers to run a particular branch by supplying different b values to each measurement performed. This allows for full MBQC simulation.

4.2.5 Correction operators X and Z

The correction operator Z changes the sign of coefficients that have i -th bit in the binary representation of their id set to 1. The correction operator X swaps the coefficients that have the binary representation of the form $xxx...1...xxx$ with the ones that have the representation $xxx...0...xxx$ (where the 0 and the 1 are on the i th place of the binary representation, and x stands for either 0 or 1 depending on the coefficient).

4.2.6 Adding coefficients together operator

When a qubit is measured the dimensionality of the matrix is reduced. Then there would be terms that have the same meaning. For example if we have a system

$$x|00\rangle + y|01\rangle + z|10\rangle - w|11\rangle$$

And after measurement of qubit 2 we are left with

$$a|0\rangle + b|0\rangle + c|1\rangle - w|1\rangle$$

Then it is obvious that this system could be rewritten as

$$(a + b)|0\rangle + (c - w)|1\rangle$$

This is the action of *AddCoeffTogether* operator.

4.3 RUNNING SIMULATIONS

4.3.1 Using the Library

A simulation could be run using library calls and the way it is done resembles the description of the pattern itself. For example we could take the quantum teleportation example that has a description

$$X_3^{S_2} Z_3^{S_1} M_2^0 M_1^0 E_{23} E_{12} N_3 N_2$$

In order to run it using the library, we could write a code similar to the one below. Keep in mind that simulation has 0-based indexing.

```
SystemMatrix m = new SystemMatrix(3);

int b[] = new int[]{1, 1}; // which branches to take

m.performReverse(
    new X(2, b[1]), // make X correction
    new Z(2, b[0]), // make Z correction

    // measure qubit 1 with alpha = 0, and t dependency b[0], take
    branch b[1]
    new M(1, b[0], 0, new MathNumber(0), b[1]),
    // measure qubit 0 with alpha = 0, take branch b[0]
    new M(0, 0, 0, new MathNumber(0), b[0]),

    new E(1, 2), // entangle 1 and 2
    new E(0, 1), // entangle 0 and 1

    new N(2), // qubit 2 in state |+>
    new N(1), // qubit 1 in state |+>
```

```
// initialize input qubit in state a|0>+ b|0>
new N(0, new MathSymbol("a"), new MathSymbol("b")));

m.performReverse(new AddCoeffTogether());

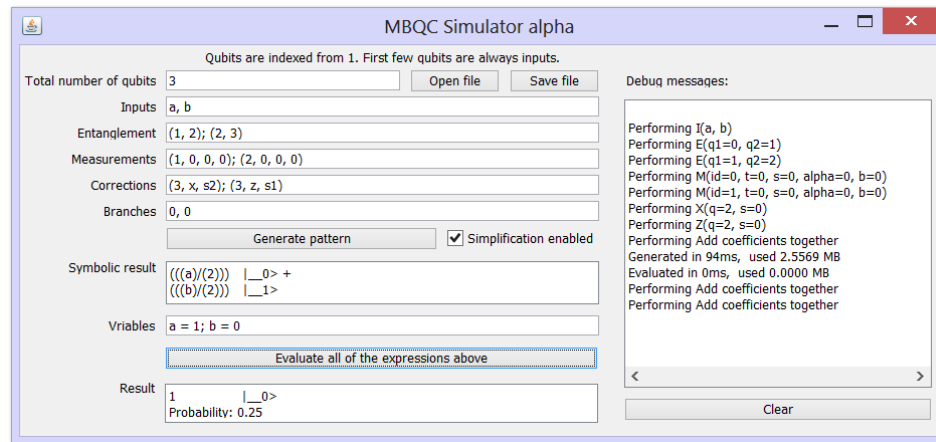
System.out.println(m);
```

The output of running this example is as expected: $(a/2)|_{_0}>+(b/2)|_{_1}>$

4.3.2 Using a GUI

4.3.2.1 User guide

A more user friendly way of running a MBQC pattern is using the graphical interface. The description could be entered in ASCII format allowing for virtually any MCalc pattern to be simulated.



The figure above shows the input required to run the exact same quantum teleportation example as in 4.3.1. An important difference is that the qubit numbering now starts from 1 as it is usually done on pen and paper. You could see the translated output in the Debug message window to the right. Since the symbolic mathematical engine is used, whenever the user is to enter a coefficient, they could use mathematical functions and constants. The format that needs to be followed is:

- Total number of qubits – the number of qubits in the system
- Inputs – the coefficients for the first n qubits. The first n qubits are always assumed to be input qubits. The number of coefficient entered should be 2^n . So an input of a, b, c, d, e, f, g, h would be interpreted as $a|000> + b|001> + c|010> + d|011> + e|100> + f|101> + g|110> + h|111>$ where the qubits in the bra-kets are the first n input qubits. Furthermore, the coefficients entered

are expressions, so they could be either values, characters, mathematical functions or a combination.

- Entanglement operations – the operations are input in the format (*qubit_id1*, *qubit_id2*) and are interpreted as $E_{qubit_id1\ qubit_id2}$. A ; is used to separate the entanglements.
- Measurement operations – the measurements are specified in the format (*qubit_id*, *s*, *t*, *alpha*). This is interpreted as $^t[M_{qubit_id}^\alpha]^s$. *Alpha* is an expression and could be written in terms of mathematical functions. Measurements could be also separated using ;. *S* and *t* are the signals generated from previous measurements. If, for example, the measurement has an *s* dependence on the 2nd and the 4th measurement, then *s* should be written as *s2* + *s4*.
- Corrections – the format is (*qubit_id*, *m*, *s*). *S* is a signal and obeys the same rules as the corrections for the measurements. *M* could be either *x* or *z* depending on whether or not an *x* or a *z* correction is required.
- Branches – the current branch that needs to be computed. There are two possible ways of representation. The first one is inputting *b1*, *b2*, *b3*, ... which would be interpreted as for measurement of qubit 1, take branch *b1*, for measurement of qubit 2, take branch *b2*, etc. If you only have measurement for the first and third qubits, you will need to use the alternative syntax which is (*qubit_id*, *b*) separated by ; where *qubit_id* is the qubit whose measurement should take the *b* branch. Keep in mind that the only allowed values for the *bs* are either 0 or 1.

Once a pattern has been computed, its symbolic representation will appear in the box below. In order to get a concrete result, the constants need to be substituted with the desired value. The format is *const_name* = *mathematical expression* and the different constants should be separated by a ;. The evaluated result would be normalized (so make sure the input is also normalized in order to get physically meaningful results). The probability for obtaining the specified branch would be also output in the result.

4.3.2.2 Implementation

The classes responsible for conducting a simulation are in the *martin.quantum* package. The GUI just populates a *McalcDescription* object which is just a container that contains the strings as they are input in the GUI. Therefore *McalcDescription* becomes a universal representation instance of an MBQC pattern. It supports saving and loading the instance to and from a file on the local hard drive.

In order to run the simulation, the *McalcDescription* needs to be translated to a series of commands like in 4.3.1. This is done by the *SimulationRunner* class. It takes a *McalcDescription*, parses it and runs the produced commands. Extra care is taken with branches and measurement dependencies so that the constructors of the operators take

the correct parameters from the branch array (again very similar to the way branches are handled in 4.3.1). Also the s and t signals are calculated accordingly. The $\text{mod } 2^3$ rule is used.

When the *SimulatorRunner* returns a *SystemMatrix*, it is displayed on the screen. When the user decides evaluate it, an evaluation rule is generated. The values for each constant the user supplies is put into a *HashMap*. It is then used to calculate and output the value for each coefficient in the *SystemMatrix*. In the same time a value for the probability of obtaining that state is calculated by taking the norm of each coefficient and adding it together (done internally inside the *SystemMatrix*).

5 IMPLEMENTATION OF GRAPHICAL DESIGNER

The graphical user interface is an interactive tool for building Mcalc pattern graphs⁴. It provides an intuitive way of describing complicated algorithms and running them. It takes care of the translation to an ASCII Mcalc description automatically so the MBQC Simulator could run the graphical pattern correctly. All of the entities responsible for rendering the GUI are located into the *martin.gui* and *martin.gui.quantum* packages.

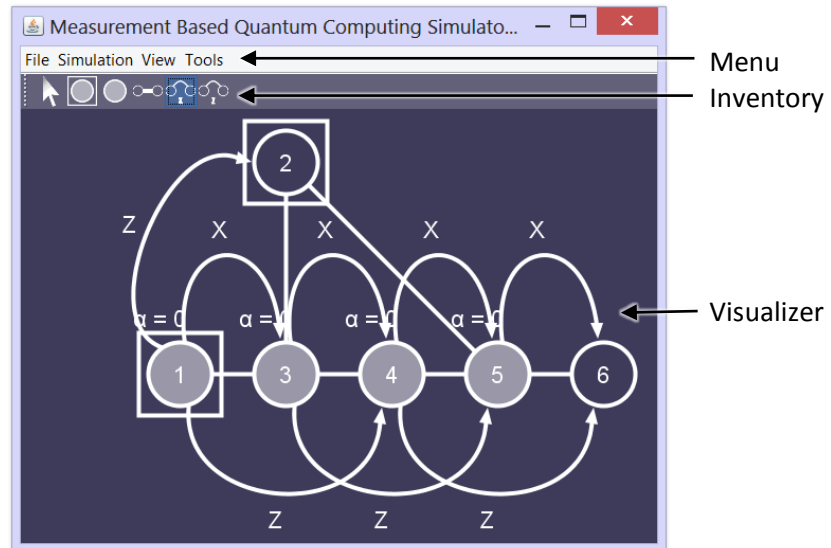


Figure 1 The Graphical Designer GUI

³ See signal addition paragraph in 3.4

⁴ The graphical representation is described in 3.4.2

5.1 DESIGN

5.1.1.1 *Item*

The interface design is object oriented and is made to be fully extensible beyond the current features. All that is required is that a new class extending *martin.gui.quantum.Item* is developed that implements the desired new functionality. The engine will take care properly of displaying this *Item* onto the screen and the interactivity associated with it. This is currently how all existing tools are implemented.

From user point of view an *Item* is conceptually a tool. It is represented as an icon in the *Inventory*. The users click on *Items* and use them to build and interact with the MCalc graph. *Items* handle everything – from the actions taken when mouse is dragged, moved or click to the rendering of different components on the screen and their appearance during modelling.

An *Item* has two modes of functioning. In the first mode, the item is used as a tool for creating a part of the graph. The tool icon moves with the mouse as the user selects it from the inventory. After the user is satisfied and decides to place the current object onto the screen, the second mode of the *Item* is activated which renders the object on the *Visualizer* and makes it interact with other objects on the screen.

Every *Item* needs to:

- In “tool” mode
 - Have an icon image and provide the filename of that image.
 - React to mouse actions like drag, move, click, release.
 - Create an instance of itself and put it into the *Visualizer* in “render” mode if required
- In “render” mode
 - Store its screen coordinates and report whether the mouse is on top of it
 - Provide with a mouse pointer when the mouse is on top of it
 - Render itself on the screen
 - React to movement requests by changing its position accordingly
 - Generate context menu options when required and react to clicks on the context menu
 - Tell whether it needs to be deleted given a list of *Items* that the user has requested to be removed. This is done in cases in which the current item depend on other Items. If they are deleted, then the current *Item* must also inform that it is no longer valid.
 - React to events that tell the *Item* that the state of the visualization has changed (and the Item might need to react, change its id, etc)

- Save itself and loads itself to a *String*. This is used for saving and loading the graph that is presented on the screen.

Implementing a new tool would require an implementation of the abstract methods in the *Item* class which correspond to the above actions. This is enough to make it behave correctly when the user wants to use it. Currently there are 4 *Items* that are implemented.

5.1.1.1.1 Arrow



This is the tool that allows the user to move and change items on the screen. It does not have “render” mode and you cannot place a cursor on the screen. It only has “tool” mode. The *Visualizer* is queried on mouse move whether there’s something below the mouse. If there is, the cursor of the mouse is changed to the default cursor dictated by the object underneath. If a click is performed, a request for movement is sent to the object making it change its position or other properties it decides to change. The arrow also helps build the context menu for each item.

5.1.1.1.2 Qubit



There are two types of qubits – *input* qubits and *normal* ones (defined in *Qubit.type*). The type of the qubit is defined in the constructor. The two types have different icons and render differently on screen. The input qubit has a square surrounding the circle. The type could be changed at runtime via the context menu that the qubit defines.

Qubits are movable. When they are being moved around, a snap-to-grid feature is implemented. A square grid is being drawn on top of the surface and whenever the mouse approaches an edge, the qubit “sticks” to the centre. This allows for better visual representation of the pattern that is being produced.

The non-trivial part is the way qubit removal is undertaken. If a qubit is removed from the visualizer, there would be a gap in the numbering of the qubits. Therefore the remaining qubits need to change their identification numbers so that the gap is filled. For example if we have a pattern consisting of qubits 1, 2, 3 and 4 and the user decides to delete 3, we would be left with qubits 1, 2 and 4. This description of a quantum circuit violates the norm. Therefore the last qubit, qubit 4 needs its number changed from 4 to 3. So we end up having qubits 1, 2 and 3 in the whole pattern.

When the *Visualizer* notifies the qubits that a removal has been done somewhere in the pattern, they need to figure out how to rearrange themselves. The algorithm is recursive. It finds the highest qubit in the pattern and it recursively changes the id of the previous qubit until no more changes are possible. Implementing something like a gravitational “fall”:

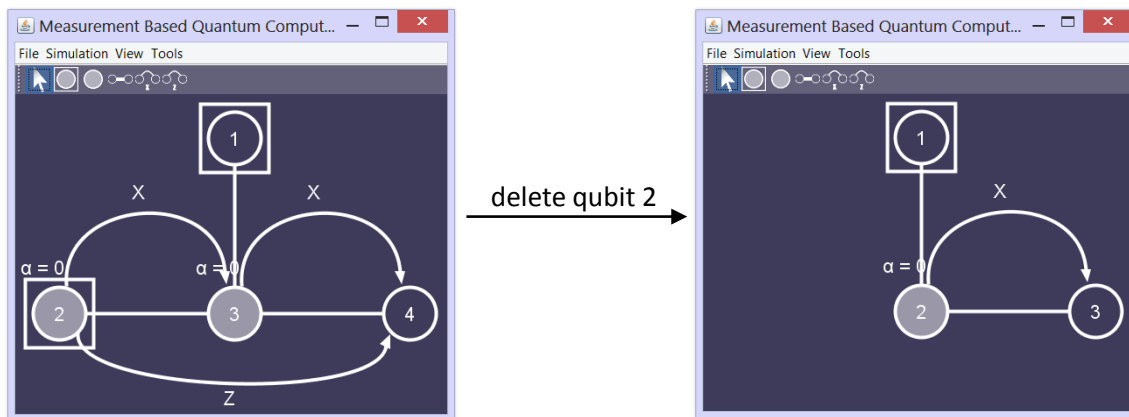
```

// This is called by the Visualizer when something has been deleted
Function changedCallback() {
    qid = qubit with highest id;
    if (qid.id == my_id)
        fix();
    else
        qid.changedCallback(); // recursively find qubit with highest id
}

Function fix() {
    qid = get highest qubit less than me;
    if (qid does not exist)
        my_id = 1; // if there's no one in front of me, I'm 1!
    else {
        qid.fix(); // put the right number on the qubit in front of me
        my_id = qid.id + 1; // I'm 1 above that!
    }
}


```

When running this algorithm, this renumbering is possible:



Also note because *Entanglers* and *Correctors* depend on the qubit that has been deleted, they have disappeared as well.

5.1.1.1.3 Entangler

 The entanglement is represented as a line joining two qubits. It depends on two *Qubit* objects. It is rendered as a simple line that spawns between them. The location of the computed by taking into account the locations of the two *Qubits*. If the mouse is a certain distance (6 pixels) from the centre of the line, it is assumed that the mouse is above the line. *Entanglers* cannot be moved around. When a *Qubit* that the *Entangler* depend on is deleted, the *Entangler* reports it is no longer valid and the *Visualizer* disposes it as well.

The entanglement has two actions – selecting first qubit and selecting the second one. This is handled internally and the icon is moved slightly to show the user which action is being performed.

5.1.1.1.4 Corrector



A *Corrector* is very similar to the *Entangler* because it depends on two *Qubits*. If one of them is deleted, the *Corrector* is no longer needed and is removed as well. It also has two actions similar to the *Entangler*. There are two types of corrections defined in *Corrector.type* – *x* and *z*.

A challenge in representing the *Corrector* was rendering the curved line that connects the two *Qubits*. Java's *CubicCurve2D* is used to generate an array of points that span across the two qubits that cover a **Bezier** curve. Intersection with the curve and the circles surrounding the qubits is taken. Then a new Bezier curve spanning between the two points on the perimeter of the qubits is generated. It is drawn on the screen. The mouse is considered on the curve if it is 6 pixels away from a segment. When the *Corrector* is moved, the height of the curve is adjusted interactively. The end of the arrow is a triangle which is rotated to face away from the final segment of the curve.

5.1.1.2 Inventory

The *Inventory* is a class that extends *JToolBar*. It stores a list of *Items* and manages which one of them is selected by the user. A call-back interface *inventoryClickListener* is used by the *Visualizer* to obtain the *Item* that the user wants to use as a tool to build the MCalc pattern graph. The physical buttons are stored as *JToggleButton*.

The current inventory implementation is initialized as

```
private Inventory toolBar = new Inventory(  
    new Arrow(),  
    new Qubit(type.input),  
    new Qubit(type.normal),  
    new Entangler(),  
    new Corrector(corrtype.X),  
    new Corrector(corrtype.Z));
```

5.1.1.3 Visualizer

The *Visualizer* is the main component that displays the graph and allows for interactive manipulation. It extends *JPanel* and takes care of rendering an array of *Items* in “render” mode and an *Item* in “tool” mode. The item that is in “tool” mode is the currently selected icon from the *Inventory*.

Repainting of the whole surface is done only when something has been changed on the screen. If such an event occurs, all *Items* from the pattern shown on the screen are iterated and are presented with a surface to paint themselves on. In addition mouse actions are handled and passed to the *Items*. Also a responsibility of the *Visualizer* is displaying and populating the context menu with the means of a *JPopupMenu*. Also the *Visualizer* does the translation from the graphical representation to ASCII MCalc description of the pattern that is being built.

In order to load and save the current visual pattern to a file, a strings are taken from all *Items* and saved to a file. The reverse process happens when a file is being loaded. A special care is taken when loading *Items* that depend on other *Items* (like *Corrector* and *Entangler*) that their dependencies are already in memory.

5.2 TRANSLATION TO MCALC

The translation process is very important for the functioning of the system as a whole. The graphical pattern must be correctly interpreted and translated to a *McalcDescription* in order for the simulation to produce the expected results. This is done inside the *Visualizer*.

The procedure for producing a *McalcDescription* out of an array of *Items* is as follows:

- Number of qubits – number of *Qubit* instances in the *Items* array
- Inputs – the first 2^n letters of the alphabet separated by a comma. Where n is the number of *Qubits* with type *input* in the *Items* array.
- Entanglements – the *Entangler* objects in the *Items* array. The qubit ids are taken from the *Qubit* instances that the *Entangler* depends on. The procedure is


```

      For each Entangler e
        append "(e.qubit1.id, e.qubit2.id)"
      
```
- Measurements – the *Qubits* that are being measured will hold the angle of measurement. In order to get the s and t dependencies, we need to trace which *Correctors* end up pointing at the qubit that is being measured and add them as dependencies. The translation relies on the fact that a correction onto a measured qubit results in an s or t dependency (depending on the type of the correction)⁵. This could be more formally written as ${}^t[M_i^\alpha]^s X_i^r = {}^t[M_i^\alpha]^{s+r}$ and ${}^t[M_i^\alpha]^s Z_i^r = {}^{t+r}[M_i^\alpha]^s$. The procedure is


```

      For each Qubit q that is measured
        xs = get x Correctors that point at q
        zs = get z Correctors that point at q
        append "(q.id, xs, zs, q.angle)" to list of measurements
      
```
- Corrections – the *Correctors* applied to qubits that are not being measured. The procedure is


```

      For each Corrector c
        if (c.end_qubit is not measured)
          append "(c.end_qubit.id, c.type, c.start_qubit)"
      
```

All of the above are done by multiple iterations through the *Items* array. After the required information is obtained, the result is then used to populate the MBQC Simulator in order for a simulation to be ran.

⁵ The Measurement Calculus, Danos, Kashefi & Panangaden – Equations (13) and (14)

6 SYMBOLIC ALGEBRA SYSTEM

6.1 MOTIVATION

In general this project could have used plain floating point arithmetic as a system for algebraic manipulation. This approach would have been certainly easier to implement, but would fail to give the flexibility that a symbolic manipulation algebra system would give. For one thing, the error introduced by the floating point rounding would increase with the number of operations undertaken. On the other hand, having a human readable algebraic representation is certainly more helpful than having an answer of a plain floating point number. Furthermore computation could be saved by evaluating the whole algorithm symbolically and then substituting with different input values rather than having to re run the whole simulation for each set of input values.

Of course there are a few disadvantages which have been addressed. Firstly, any error in the system would propagate through the whole simulator, therefore the mathematical system must be fully tested and working. Secondly, using objects as coefficients increases significantly the RAM requirements of the project. This is actually not a big issue considering that the typical MBQC patterns that are currently being ran have quite a small number of qubits in them. If RAM was an issue, the system could be easily modified to use the hard disk as a buffer by modifying the *SystemMatrix* class to store its array on the hard drive, although this particular feature has not been implemented in the project.

Therefore there was a need for a robust, flexible and lightweight system that could manipulate algebraic expressions symbolically. It needed to use as less storage as possible. It needed to be accurate, fast and allow for automatic simplifications. Parsing expressions from and to strings was essential to provide a smooth user experience in order to allow input to be entered as mathematical expressions rather than just plain constants.

Several already available libraries were considered as an option. Libraries like **symja**, **JAS** and **j'Abr** would have done the job but there were some drawbacks. Since RAM memory usage is a big issue when dealing with such a big number of coefficients, it was vital for the system to conserve as much of it as possible. It is not entirely clear how those system manage RAM memory. Also it is not entirely clear how they handle the computation internally and the efficiency of the algorithms used. Another issue is that they are general purpose – have too many features that won't be useful for this project making them too clumsy for the purpose.

A decision was taken to build a completely custom algebraic manipulation system for parsing and evaluating expressions from scratch. The goals were – it needs to have as few memory allocations as possible and needs to have a way of simplifying complex expressions by removing objects and therefore reducing RAM usage. This is done by only cloning entities when long term storage is desired. It also needs to natively work with complex numbers. Also it has to be custom built for the task it's going to serve – which is mainly multiplications and additions of expressions. As a result a fast and reliable custom system for algebraic manipulation was developed from scratch.

6.2 OBJECT ORIENTED DESIGN

The base unit of the system (which is situated in the *marto.math* package) is the *MathsItem*. All of the entities implement a *MathsItem* which defines the base interface that any mathematical object should have:

- It could be negative
- It could be a complex conjugate
- It should be possible to evaluate it to a *Complex* number given a set of constants and their corresponding value
- You should be able to simplify it, reducing memory usage
- You should be able to multiply it with another *MathsItem*
- You should be able to add to it another *MathsItem*
- You should be able to divide it by a *MathsItem*
- You should be able to get a clone of it which has the same mathematical meaning (may not be instance of the same class, though)
- You should be able to parse it from a plain string

Using those rules, the different entities were built.

There are few assumptions that the whole system uses. Additions and multiplications are destructive – they either succeed and change the original values or fail and don't change anything. Therefore if multiplication occurred, the object that we are multiplying with will no longer be required – therefore we would free memory.

6.2.1 MathExpression

This is the a placeholder for mathematical expressions of the form $a_1*b_1*c_1*... + a_2*b_2*c_2*... + ...$ where each term is an actual *MathsItem*.

In order to explain the multiplication algorithm, it would be best to approach it with a simple example. Multiplication of $\exp(a) * \sqrt{b} * c$ with \sqrt{c} would be executed like:

- Try multiplying $\exp(a)$ and \sqrt{c} . They are incompatible, skip.

- Try multiplying $\text{sqrt}(b)$ and $\text{sqrt}(c)$. Multiplication succeeds. The original $\text{sqrt}(b)$ has turned into $\text{sqrt}(b + c)$.
- No need to continue - multiplication succeeded, $\text{sqrt}(c)$ is no longer required.

Therefore this algorithm runs without taking more memory. It may actually free memory if no other entity is using the $\text{sqrt}(c)$. Now the expression reads $\text{exp}(a) * \text{sqrt}(b+c) * c$.

If we rather tried multiplying $\text{exp}(a) * \sin(b) * c$ with $\text{sqrt}(c)$, all multiplications will fail since all of them are different types of objects. In this case a copy of $\text{sqrt}(c)$ will be concatenated to the sequence resulting in $\text{exp}(a) * \sin(b) * c * \text{sqrt}(c)$. The same exact rules are followed when additions are made. For example if we have $a + 5 + \text{sqrt}(c)$ and try to add 6 to it, it will turn into $a + 11 + \text{sqrt}(c)$, disposing of the 6 (a note: the 6 is an object of type *MathNumber*).

Extra care need to be taken when multiplying two expressions in the form of $(a_1 * b_1 * c_1 * \dots + a_2 * b_2 * c_2 * \dots + \dots) * (x_1 * y_1 * z_1 * \dots + x_2 * y_2 * z_2 * \dots + \dots)$. In such a case the whole thing is calculated as $a_1 * b_1 * c_1 * \dots * x_1 * y_1 * z_1 * \dots + a_1 * b_1 * c_1 * \dots * x_2 * y_2 * z_2 * \dots + \dots$ which is simple bracket expansion. Then the addition and multiplications rules are applied. You could clearly see that such expansions could result in quite a lot of coefficients. Hence simplifications are needed.

The concept of hard simplification is quite time consuming. If enabled, it will clone every single object in the expression. Some object may decide to clone themselves into simpler ones. For example, $\text{sqrt}(4)$ (which are two objects, an *MathSqrt* that contains a *MathNumber*) may decide to turn into 2 (which is one *MathNumber*). No significant overhead occurs for number of qubits less than 10, but by default this is disabled due to performance concerns. Keep in mind that turning on this option does not increase RAM memory since whenever an object is cloned, the older one is no longer needed and is deleted by the garbage collector. The deep simplification could be enabled by changing the *DEEP_SIMPLIFY* flag in the *MathsExpression* class. A benefit of this type of simplifications is also more human readable form. It could also lead to further simplifications because new types of objects may interact with other objects of the same type.

A more common type of simplification is applying some algebraic tricks to get rid of additional coefficients. The procedure is – repeat the following sequence until no changes in the expression occur any more:

- Recursively call *simplify* on all nested expressions. This would more or less guarantee that all of the nested expressions are simplified.
- If a series of multiplications has a zero in it, turn the whole thing into a zero and delete the coefficients. If there is a zero in an addition, delete it.

- If there is a one in a series of multiplications, delete the 1.
- If deep simplify was ran or nested expressions were simplified, group similar multiplications together. This means that if we had $4*\text{sqrt}(4)$ before deep simplification and after it turned into $4*2$, the whole thing would be evaluated to 8 which would mean storing 1 coefficient rather than 3 to represent the same concept.
- Remove nested expressions. For example if we had, $[2*[[3]]]$ this would be changed to $[2*3]$ (we have used $[x]$ as meaning a *MathsExpression* containing an element x). The group similar expressions together will make sure on the next run the multiplication would be converted into one result.
- Perform additions. If we originally had $\text{sqrt}(4) + [2*[[3]]]$ which after a deep simplification, removal of nested expression and grouping multiplications was turned into $2 + 6$, this method will add the two coefficients together to result in a 8.
- Common factors. If we have an expression $a*\text{sqrt}(5) + a*\text{exp}(2)$, it will be turned into $a*(\text{sqrt}(5)+\text{exp}(2))$, therefore one of the a s is no longer needed.

Those rules are enough to produce good quality results and use little memory and CPU. Evaluation of the results is done recursively as well. Each of the sub expression is evaluated and the results are multiplied and added together accordingly to produce the final value.

Division of *MathExpression* has not been yet implemented. However division of other *MathItems* is implemented. Also the concept of division can only be introduced via *MathFract* which represent a fraction. This is because divisions are not very common and are usually done only on simple numbers rather than full expressions during the evaluation of a typical Mcalc pattern.

6.2.2 MathNumber and MathSymbol

MathNumbers are containers for a simple real number. No further simplifications possible. *MathNumbers* are divisible by *MathNumbers* only if they are integers and are multiples of each other, therefore avoiding floating point arithmetic as much as possible.

MathSymbols are the variables you see. Each of them has a string as a name. They cannot be multiplied together and can only be added together if they are the same with opposite signs, meaning a result of 0 is obtained. When evaluated, *MathSymbols* get their values from the variable – value pairs provided. Since they are complex numbers, complex conjugation is taken into account.

6.2.3 MathFract

Contains two *MathItems* – a numerator and denominator. Addition and multiplication is implemented as addition and multiplication of two fractions. In addition, a common denominator is achieved by multiplying the two numerators and then adding them together. For multiplication, the numerators and denominators of the two fractions are multiplied together.

The interesting part is the simplification which only works on denominators and numerators that contain integer multiplication terms and uses the greatest common divider in order to simplify the coefficients. Also requests for simplifications are passed to the denominators and numerators as well.

6.2.4 MathFunction

It is just an abstract representation of a mathematical function that specific functions extend and implement. It has a container for a *MathsItem* which is the argument of the function. Therefore *MathFunction* cannot be instantiated directly. It provides general facilities for parsing functions from *String*.

MathItems that extend *MathFunction* are:

- *MathSqrt* – implements a *sqrt* onto the complex argument. *MathSqrts* could be multiplied together by simply multiplying their arguments (extra care is taken for non-real arguments).
- *MathExp* – implements an *exp* onto a complex argument. *MathExps* could be multiplied together by adding their complex coefficients together.
- *MathIm* – is interpreted as “multiply by *i*”. Therefore *MathIm(2)* has the mathematical meaning $2*i$. Multiplications and additions are correctly handled.

6.3 MATHSPARSER

Parses an arbitrary string containing supported mathematical expressions into a tree of *MathItems* that could be used for easy manipulation. Each *MathsItem* is required to implement a static *fromString* method that parses it from an input *String*. The classes that are available for parsing need to be registered in the *CLASSES* array in the *MathsParser* class. The exact order of parsing would be the order in which those classes are declared. Java Reflection API is used to invoke the *fromString* method of each class. If a *MathsItems* class reports that it can't read an instance from the *String* provided, the next *MathsItems* in the *CLASSES* array is queried. The current definition of the *CLASSES* array is

```
private static final Class<?>[] CLASSES = {MathExpression.class,  
MathFract.class, MathNumber.class, MathExp.class, MathIm.class, MathSqrt.class,  
MathSymbol.class};
```

For example if we had the string “**a + b * sqrt(2+3)**”, this is the recursive algorithm that would be executed in order to parse it:

- *MathExpression* determines it could parse it, so it asks *MathsParser*
 - **parse “a”**
 - *MathSymbol* determines it could parse “a” and returns a new *MathSymbol* object that contains “a”
 - **parse “b”**
 - *MathSymbol* returns “b”
 - **parse “sqrt(2 + 3)”**
 - *MathSqrt* determines it could parse it, so it asks *MathsParser* in order to resolve its argument
 - **parse “2 + 3”**
 - *MathExpression* determines it could parse it so it asks *MathsParser*
 - **parse “2”**
 - *MathsNumber* returns “2”
 - **parse “3”**
 - *MathsNumber* returns “3”

Therefore we end up with a structure like

```
Expression[ Symbol[a] + Symbol[b] * Sqrt[ Expression[ Number[2] + Number[3] ] ] ]
```

Which could be used for further computation.

7 VERIFICATION

Verification of the core engine is an important factor in ensuring the quality of the implementation. Although some parts of the system have undergone automated self-testing based on unit tests, the overall performance needs to be benchmarked against well-established results. Quantitative results should objectively show the fitness of the system.

This being said, verification of the current system as a whole is a quite challenging task. The reason is that this is the first full MBQC simulator that allows simulation of MCalc patterns. The lack of such software means that only parts of the whole could be automatically tested. Nevertheless several solutions exist that could aid the testing process.

Some of the methods that are presented in this section use already existing software that has been independently verified. They are based on the assumption that the third

party software is producing correct results. Those methods could be in theory automated to produce consistent outputs and reliable benchmark for accuracy. This would guarantee that the components that are undergoing the aforementioned procedures are free from bugs in the context of the data that is being analysed. No testing could possibly guarantee absolutely bug free performance unless all possible inputs/output pairs of the algorithm are known in advance. This is impractical in the context of a simulator.

7.1 SYMBOLIC ALGEBRA SYSTEM

The mathematical system for symbolically manipulating algebraic expressions lies at the core of the whole project. All of the components rely on its correct behaviour. The way simplifications are done could introduce a large number of bugs. Luckily, testing of this system is simple enough and is quite reliable to rule out any possible causes of error.

A unit test **MathAutoTest** was specifically designed to perform an automated test onto the whole of the existing system, making sure that algebraic simplifications do not result in unexpected results. The idea is based on the fact that there is a simple way to perform the same computation that the algebraic system is supposed to undertake by explicitly calculating the results in parallel using ordinary floating point arithmetic. The result of the algebraic system is then compared with the expected result and any large discrepancies will make the automated unit test to fail.

In order to achieve this there is an array of **MathsItems**. The *generateComplexMathItem* method creates a random **MathsItem** and puts it into a random place in the array. The way it is done is, it picks at random a mathematical function – *exp*, *sqrt*, *fraction*, *Im* or a *mathematical symbol* or a *number* and uses the classes in the system to create a **MathsItem** object that represents it. If the resulting **MathsItem** is a mathematical function, a randomly chosen element from the array is used as an argument. If instead it is a symbol, it is picked from a predefined pool of available symbols that have some initial values (so that evaluation is possible). The newly formed **MathsItem** is modified by adding or multiplying it with a randomly chosen element of the array. The value of the function is compared before and after the simplification. If any differences are found, an error is thrown.

You could find below a sample pseudo code of the verification process:

```

Array of MathsItems items;
Repeat N times
    Mathematical function f = random between (exp, fract, sqrt, im, number, symbol)
    If (f is exp, fract, sqrt or im)
        argument of f = random element from items
    If (random number between [0, 2) == 0)
        f = f + random element from items;
    else
        f = f * random element from items;

```

```

put f in a random place in items;
Complex result1 = f.evaluate();
f.simplify();
Complex result2 = f.evaluate();
If (result1 != result2)
    throw an Exception("TEST FAILED");

```

Where **N** is the number of iterations to be run. This algorithm can generate arbitrary mathematical expressions because of the recursive nature of the generation process. To demonstrate its working, a sample possible output for the algorithm for **items** = {1, 1}, **N** = 5 and pool of symbols {*a*, *b*} would be:

1. **items** = {exp(1)+1, 1} // exp is randomly generated, 1 is randomly picked for argument, it is then added to 1 and is put as first element of **items**
2. **items** = {exp(1)+1, a} // a is randomly generated, it is multiplied by 1 and is put as second element of **items**. Note the simplification has turned $a * 1$ into simply *a*
3. **items** = {b * (exp(1) + 1), a} // b is generated and multiplied by the first element and set as first element
4. **items** = {b * (exp(1) + 1), a + sqrt(b * (exp(1) + 1))} // sqrt is generated, b * (exp(1) + 1) is picked as an argument. It is then added to a and put as second element
5. **items** = {Im(b * (exp(1) + 1)) * (a + sqrt(b * (exp(1) + 1))), a + sqrt(b * (exp(1) + 1))} // Im is generated, it got its argument from the first element and it was multiplied by the second element and put as first element of **items**.

You could actually see that even for a small number of elements and iterations the algorithm produces a set of complicated random expressions. This means that the test is being conducted with completely arbitrary inputs. The Unit test has an **N** = 10 and number of elements in **items** = 20. The unit test actually runs the whole algorithm 10 additional times producing 100 different expressions which are being tested.

The core of the test is comparing the value of a random expression before and after simplification. This will pick up all possible errors and bugs that could exist in the symbolic algebra manipulation library. This is because the bugs could be divided into two subcategories.

- Simplification bugs – the resulting expression after simplification is not mathematically equivalent to the initial expression. This would yield a difference in the evaluated output of the expressions and will be detected as an assertion error and the test will fail.
- Evaluation bugs – assuming the simplification is correct, the two resulting expressions would be mathematically equivalent. It will be very likely, that they will have different representations. If the evaluation system is malfunctioning the resulting expression after simplification will not yield the same result as the original one therefore the test will fail.

Since the unit test is not failing for any number of runs, it could be assumed that the mathematical system can handle an arbitrary expression and simplify it correctly. Therefore the correctness of the algebraic system could be proven for the random examples that have been generated during testing.

7.2 EXPERIMENTAL DATA ANALYSIS

The first approach to verifying the fitness of the simulation itself is to actually use real experimental data and compare it with the predictions done by the simulator. The experimental data that is currently available is taken from an experiment on optical implementation of quantum computing. The choice of data source is rather useful since there already exists a third party software that is specifically developed to analyse this particular experimental setup. Furthermore, the available software has been tested and provides a good benchmark for performance and accuracy.

7.2.1 Optical Implementation of Quantum Computing

The experimental setup is described in details in *Diagnosing Optical Implementations of Quantum Computing* by Flaviu Cipcigan. In a nutshell, a laser produces photons which pass through a series of optical filters. Each filter applies a certain quantum operation to the photons. The original stream is split into several photon beams. Detectors at the end of the optical paths measure the arrival of individual photons. Most of the events are discarded, since the probability of entangling photons is quite low. When photons arrive at the same time in all of the detectors, a certain assumption could be made about the particles and their quantum states. Events that have such coincidences are recorded and the number of photons arriving at those states is stored into a csv file.

The quantum “algorithm” implemented could be neatly described in terms of MCalc operations onto an initial system called a “lab cluster”. A simulation of this lab cluster with a series of measurements could be done. This will change the state of the cluster. The probability of obtaining this particular state could be calculated and it directly relates with the relative number of photons that are expected to trigger the simultaneous events at the detector. The predicted probability and the actually obtained one could be compared and the quality of the comparison will prove the quality of the simulation algorithm.

The lab cluster contains 4 qubits. They start in an initial state

$$\frac{1}{2} [|0000\rangle + e^{i\frac{n\pi}{4}} |0011\rangle + i |1100\rangle - ie^{i\frac{n\pi}{4}} |1111\rangle]$$

Where n depends on the type of experiment that is being performed and is recorder in the filename of the csv file.

A series of 4 measurements are made, which could be either an X, Y, M, P or Z measurement. Each set of measurement corresponds to a row in the csv file. For each of them, events are recorded for all of the possible 4 branches thus generating 16 columns for each row.

In order to obtain the probability of a particle ending up in a certain branch when a certain sequence of measurements is applied, the state of the lab cluster is used. As mentioned previously in the report, the probability of a branch is the norm squared of the resulting vector.

7.2.2 Simulation

In order to implement a simulator tailored for the optical quantum computation experiment, several additional entities needed to be added to the already existing system.

LabCluster – a class that extends SystemMatrix and allows for initialization of the pool of bits to a certain lab cluster. Once an instance of this class is created, measurements could be directly applied to it and the probability could be extracted using the inherited methods from SystemMatrix.

ZM – a Z measurement in the basis of $\{|0\rangle, |1\rangle\}$. This literally means that a Z measurement in branch 0 will take only the coefficients of the current qubit that are in the $|0\rangle$ state, setting the others to 0. A measurement in branch 1 will take only the coefficients in the $|1\rangle$ state.

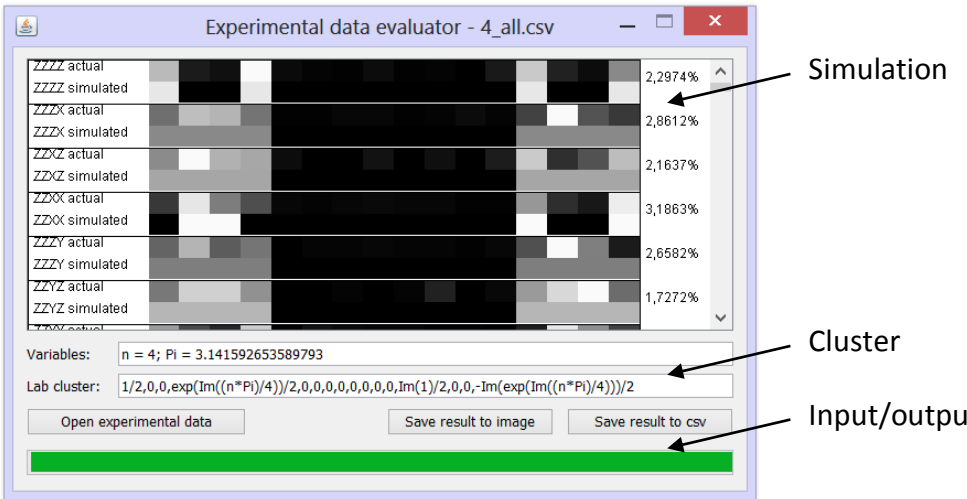
WorkSheet, TableStorage – for parsing the csv and storing the original experimental probabilities for each pair of measurements and branches.

ExperimentVisualizer, CSVExporter – for converting predicted output to images and csv files for further analysis and comparison with the original data.

ExperiTest – performs the tests by loading a file. It takes an ASCII version of the initial cluster state and runs an experiment. Some sanity checks of the results and inputs is also done and warnings are issued in cases in which total probability does not add up to 1 or input is not normalized. A callback ensures that updates are being passed to the original invoker, allowing for user friendly features like a progress bar to be rendered on the screen. It then returns a result that contains a CSV file and an image as a result of the simulation.

A GUI interface was developed for intuitively interacting with the simulation. It allows for defining a custom lab cluster and setting values for variables. It then shows the visual representation of the analysis on screen and gives option to save it as an image file or as

a csv file. Also a progress bar is shown while the analysis is taking place so the user is aware of the status of the computation.



This configuration allows for flexibility in the simulation. If a new set of experiments is generated (that follows the same input csv format), the analysis could be easily adapted by changing the initial state of the lab cluster. The system can handle an arbitrary number of qubits as well (i.e. the lab cluster could in theory hold more than 4 qubits) as long as the number of qubits in the csv file matches with the number of qubits in the lab cluster.

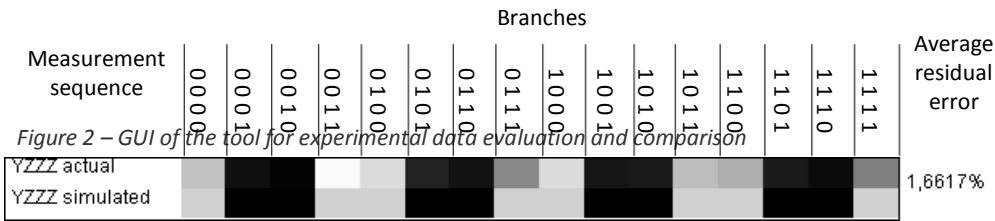
7.2.3 Output

The output from the simulator could be best visualized using a grayscale mapping⁶. The mapping is between the minimum (black) and maximum (white) probability in each row



between the actual and the simulated data.

How to interpret a “row” from the generated result image:



⁶ Flaviu Cipcigan, Diagnosing Optical Implementations of Quantum Computing

The row above means that for each cell, which is illuminated, the lab cluster needs to be initialized and 4 measurements must be done (in this case a Y measurement, followed by three consecutive Z measurements). The first column represents all measurements resulting in a projection to the 0 branch. The second column means the final Z measurement is resulting in a projection to the 1 branch while the other 3 result to the 0 branch.

Therefore for each row like this, 16 simulations are done using the following algorithm:

```

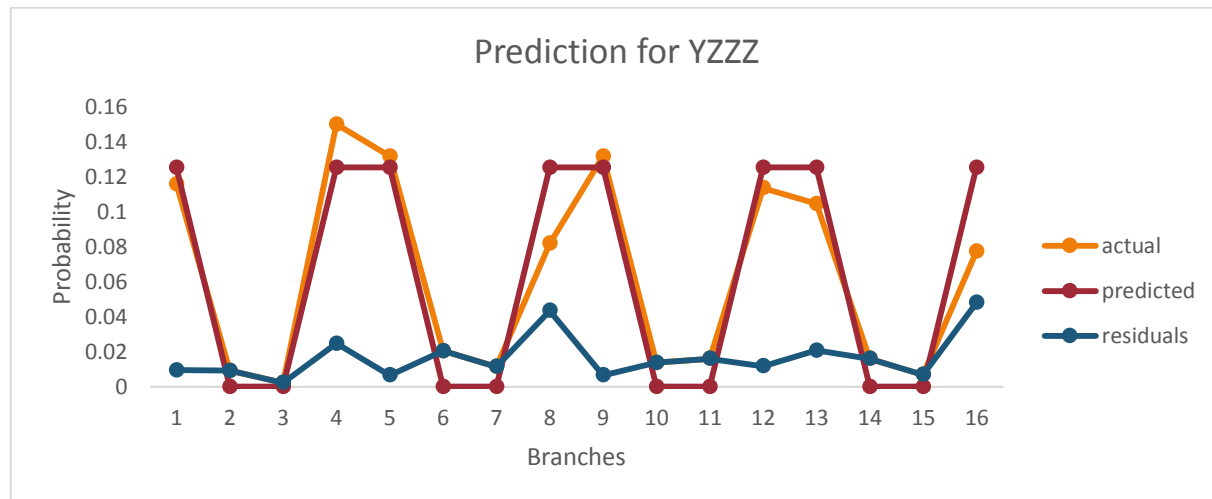
For each branch combination B in bs do:
    Initialize new lab cluster
    For i from 0 to size of ms do:
        Perform measurement ms[i] taking a branch B[i]
    Calculate probability of the cluster

```

For the example above, **ms** = {Y, Z, Z, Z}, **bs** = { {0,0,0,0}, {0,0,0,1}, {0,0,1,0} ... {1,1,1,1} }

If we assume that we have the original probabilities in an array **orig** and the simulated probabilities in an array **sim**, then we could take the values **max_val** = $\max(\max(\mathbf{orig}), \max(\mathbf{sim}))$ and **min_val** = $\min(\min(\mathbf{orig}), \min(\mathbf{sim}))$. The colour for each cell is calculated as **colour** = $(\text{current_value} - \text{min_val}) / (\text{max_val} - \text{min_val})$. The resulting value is mapped as **colour** = 0.0 means *black* and **colour** = 1.0 means *white*.

The average residual error is the average absolute difference between results obtained in the experiment and the simulated output for a given row. It gives a quantitative measure on the accuracy of the theoretical fit. It is reported in percent, which is the actual value multiplied by 100.

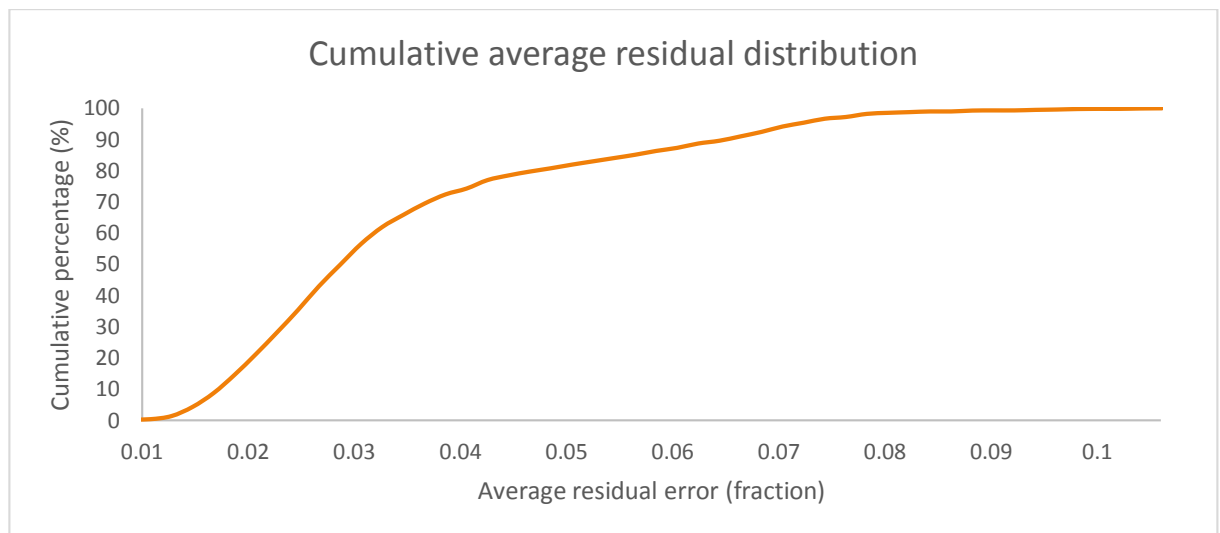


The relationship of actual values vs predicted vs residuals should be clear from clear from the figure above. The average value of the residual line is 0.0166 therefore the reported value as an average residual error for the row shown in Figure 2 – GUI of the tool for experimental data evaluation and comparison is 1.6617%. You could also see that in this case **max_val** = 0.149 and **min_val** = 0.

7.2.4 Results of verification

The verification has been done on 8 lab clusters, each with 225 measurement tuples, each with 16 possible branches. This totals to 28800 simulations for the available data. The average residual error is 3.68%. The best simulated experiment has a residual error of 0.8% while the worst one has an error of 10.8%.

49.83% of the rows have a value of the average residual error less than 2.96%. Only 27.61% of the data has an average residual error bigger than 3.95%. This could be clearly seen on the cumulative average residual distribution profile below. The plot shows the fraction of data rows which have an average residual error less than a certain value.



This clearly show that majority of the data could be explained using the simulation correctly. Here are two examples of a good and a bad theoretical prediction.



Figure 3 Good agreement with simulation



Figure 4 Disagreement with simulation

It could be seen that the actual data is quite noisy but the pattern that it follows could be observed. The reason for disagreement is the Y measurement and it is explained in detail in *Diagnosing Optical Implementations of Quantum Computing* by Flaviu Cipcigan.

A result of the *Diagnosing Optical Implementations of Quantum Computing*, a software was developed for specifically verifying the results of the above experiment. The software was run on the exact data set input files and the predicted

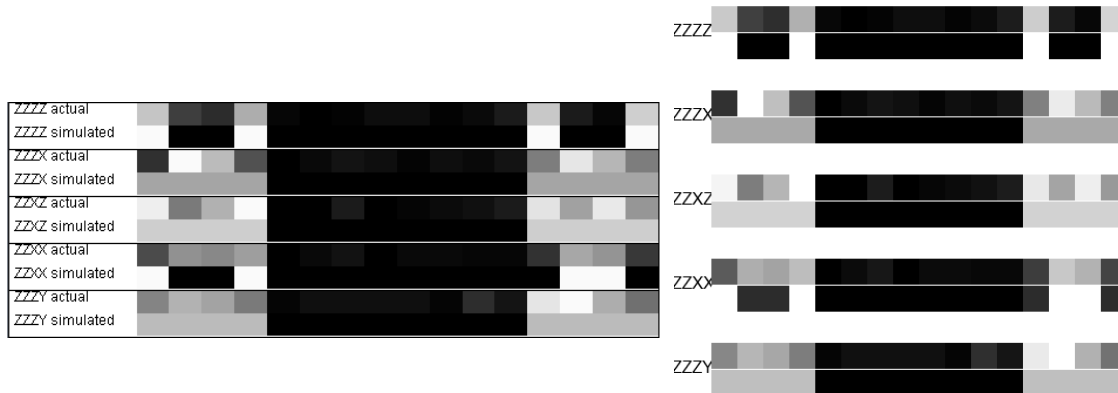


Figure 5 Comparison between output of the current project (on the left) and Mr Cipcigan's project (on the right)

The resulting data was compared and the two applications agreed completely on the predicted values. This could be seen on Figure 5 Comparison between output of the current project (on the left) and Mr Cipcigan's project (on the right). These are the first 5 measurements of the 7th lab cluster. They match completely as do all of the other predictions.

7.2.5 Conclusion

It should be made clear that Mr Cipcigan's software uses a completely different algorithm for generating the theoretical predictions. His software works exclusively with probabilities. The core of the current project is a full MBQC simulation. Furthermore, the current project uses symbolic algebra which could make printing the exact probability and lab cluster more human readable.

It is apparent from the results presented in the previous subsection that the output from the current simulator completely matches with the output from an already existing software. Furthermore, it could be successfully used to explain and analyze experimental data. Therefore this should guarantee the correctness of the implementation of several of the core systems – the symbolic mathematics manipulation and the MCalc and cluster operation. This is the essentially all of the required components for a full MBQC simulator.

It should be noted that the test above does not test the user-interaction part of the system. Systems that are not being used by the above test are the ASCII to MCalc

translation and the graphical pattern to MCalc translation. Those systems would be targeted with the next verification technique.

7.3 SEMI-AUTOMATED PROCEDURE

Simulation of an arbitrary MBQC pattern was not possible before the development of this software. Therefore directly comparing the output of a random MBQC pattern with an already existing software solution is impossible. This makes the task of evaluating the accuracy of the ASCII and graphical to MCalc translation challenging.

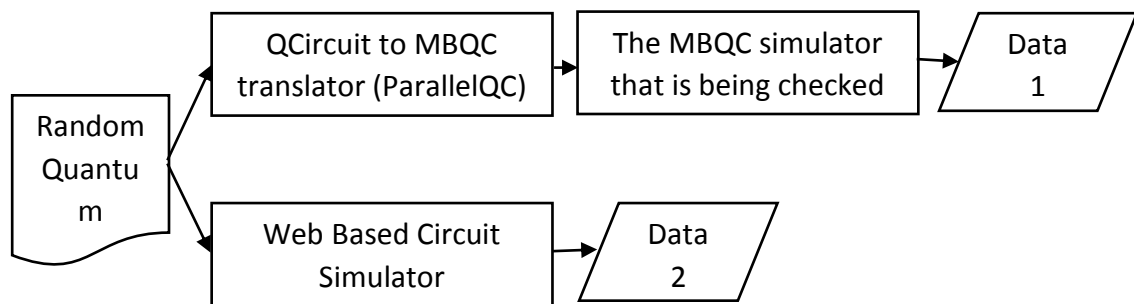
7.3.1 Procedure

A possible approach is to use more than one already existing software packages and pipe their inputs and outputs together. The method proposed below, in theory, could be fully automated to generate a good test. The reason the automation hasn't been done at this point is that one of the third party software does not have an automated input system but rather relies on a GUI.

The software packages that have been used are:

- Software⁷ developed by Einar Pius that translates quantum circuits into MCalc description. The software is written in C++.
- Quantum Circuit Simulator⁸ by Davy Wybiral that simulates quantum circuits. The software is web based and written in JavaScript.

The way the procedure works:



The Random Quantum Circuit could be literally a random generated one. It does not need to represent anything physically meaningful. The verification method will produce two sets of data **Data 1** and **Data 2** for each Quantum Circuit. Those two sets could be compared in order to evaluate the correctness of the implementation.

⁷ Einar Pius, Automatic Parallelisation of Quantum Circuits Using the Measurement Based Quantum Computing Model

⁸ <http://www.davyw.com/quantum>

7.3.2 Implementation

There is a user interface built in the GUI that automates the translation from a quantum circuit to MCalc and evaluates the pattern using the MBQC simulator engine for all possible inputs. It also generates a nice ASCII image analogous to the way the web-based quantum simulator would display the quantum circuit.

The input quantum circuit can contain the following gates ZZ, H, $J(\pi)$, $J(\pi/2)$, $J(\pi/4)$ and $J(\pi/8)$. The input is compatible with the input of Einar Pius' quantum circuit to MCalc pattern translator. When the "Evaluate" button is pressed, the input circuit is parsed and transformed into an array of **QCGate** objects. Each of those objects represents a quantum circuit gate and the order of the elements in the array is the order in which the gates act on the input qubits.

This representation is important so that the input quantum circuit could be presented into a compatible format for the web-based quantum circuit simulator. The simulator does not use the same format as Einar's translator. Furthermore, it uses a different set of gates. Luckily the six possible gates in the input that we already mentioned could be translated into compatible sequence of gates that the web-based translator supports. Each of the six gates has a unique equivalent representation. The representations of those gates from the input **QCGate** array is concatenated together in order to form the ASCII web-friendly representation of the input quantum circuit.

The input is saved to a temporary file and the **ParallelQC** executable is invoked with the file as an argument. The verbose output is analyzed and the resulting MCalc description is stored. It is then parsed into a **MCalcDescription** object. This object is used to run a simulation using the symbolic algebra system. A **SystemMatrix** is generated that contains the output of the MBQC simulation.

All possible inputs (for the specific number of qubits) are generated, and the **SystemMatrix** is reevaluated for each of the possible input values. A note should be made that since symbolic algebra is involved, once the **SystemMatrix** is obtained in the previous step, a new simulation is not necessary for each input. The outputs are then shown on the screen.

There are, though, some incompatibilities in some MCalc descriptions that are generated by the **ParallelQC** translator. The MCalc model limits the **n** input qubits to always be the first **n** qubits in an MBQC pattern. This is not always the case in **ParallelQC** since the Circuit to MBQC translation is not what the software was originally designed for, and this translation is used internally (in order for the MBQC pattern to be obtained, a verbose mode should be enabled for **ParallelQC**). Therefore some patterns generated by **ParallelQC** might be incompatible with the MBQC simulator. Furthermore, some

patterns outputs could contain a large number of qubits which may be bigger than what is practical to simulate (as discussed in the beginning of the report). Therefore if an incompatible pattern is introduced, the test suit will try its best to give a meaningful error message to the user.

The automatic quantum circuit pattern generation tries to generate completely random patterns that are always compatible with the MBQC simulator. As a start, two random numbers are picked – the number of gates and the number of qubits in the circuit. In order to ensure that the generated MBQC pattern is possible to simulate, the maximum number of qubits is limited to 3 and the maximum number of gates is limited to 5. The gates are generated using a completely random distribution (using the Java **Random** class). The qubits they act on are also randomly generated with the only restriction being that the qubit ids must be getting bigger in ids, starting from the 1st one. A note should be made that this decision does not mean that only a specific subset of MBQC patterns are isolated. All of the resulting MBQC patterns would be still random. The only limit this restriction poses is that the input qubits in the resulting MBQC pattern are always the first *n* qubits (so the convention is followed).

7.3.3 Usage

From the Tools menu, the user could open the Quantum Circuit Evaluator. They need to choose the path to the ParallelQC executable which is Einar Pius' quantum circuit to MCalc pattern translator. For each test the user would like to run, they need to follow the simple procedure:

- Generate a random Quantum Circuit with the “Generate random” button (or alternatively, load it from file or manually type it in)
- Press “Evaluate”
- Copy the web-compatible description of the quantum circuit into the web-based quantum circuit simulator.
- Use the web-based quantum circuit simulator to evaluate the circuit for a sample input.
- Compare the output of the web-based simulator with the output generated from the same input in the MBQC simulator.

The procedure is semi-automatic because there is no easy way of interfacing the web-based quantum simulator automatically. Therefore the user would need to manually input the circuit into the web-based simulator. It is worth noticing that the user is a passive element in the testing procedure; they do not do any computation, but rather act as a link that merely transfers output from one component to the other component. If there was an easy API to access the web-based simulator, the whole procedure could have been fully automated.

Figure 6 The MBQC simulator (on the left) and the QC Web simulator (on right) result comparison shows a sample quantum circuit that has been using the procedure above. The similarity between the ASCII based web-friendly circuit description and the corresponding representation inside the web simulator is clearly visible. You could also notice that the outputs from both simulators for the $|001\rangle$ input completely match for the provided 9 significant figures. The user could utilize the web-based simulator to change the input (this is done by clicking on the beginning of the line representing the input qubits) and compare the generated output with the output from the MBQC simulator. In this case the verifications showed a complete agreement between the results obtained by the two simulators.

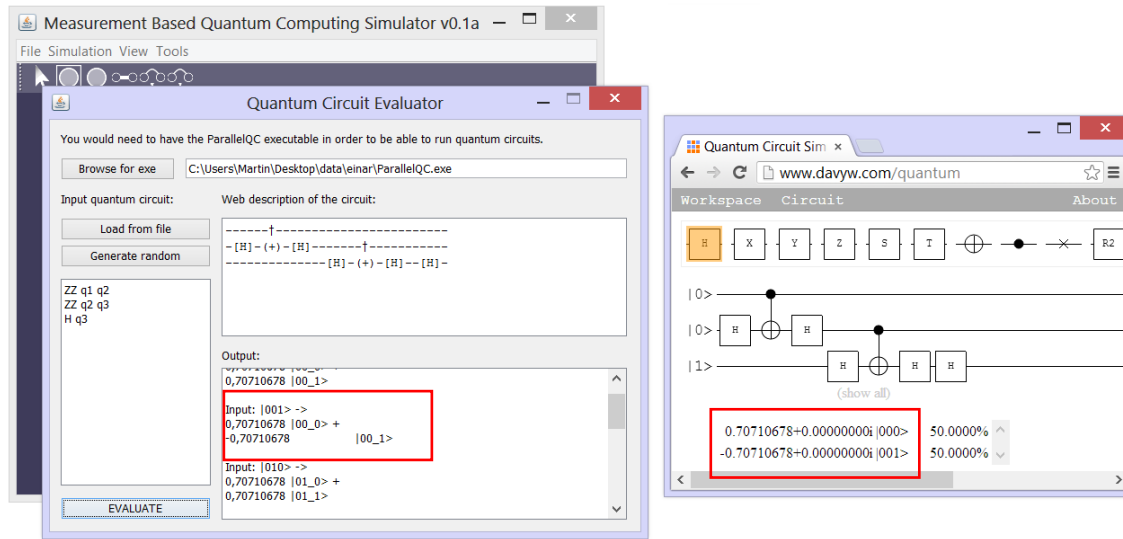


Figure 6 The MBQC simulator (on the left) and the QC Web simulator (on right) result comparison

7.3.4 Results

50 random quantum circuits were generated. 9 of them contained 1 gate, 11 of them – 2, 12 – 3, 8 – 4 and 10 – 5 gates. Of them 13 had 1 qubit, 19 had 2 qubits and 18 had 3 qubits in their quantum circuit representation. This means that $2 \times 13 + 4 \times 19 + 8 \times 18 = 246$ outputs were obtained from both simulators and compared. All of them completely agreed up to the 9th decimal digit, which is the limiting precision of the web-based simulator.

The procedure uses software packages which were already properly tested. Although the verification is semi-automatic, the user does not take part in any data processing therefore the user is not a factor in the results obtained. The agreement in the outputs can prove the reliability of the MBQC simulator that is undergoing the testing. Furthermore, the results show clearly that at least 50 pseudo-randomly generated MBQC patterns could be simulated correctly.

The components that have been tested using the procedure are the ASCII MCalc parser, the symbolic mathematics engine and the quantum operator system. The evidence show that under the testing conditions, they behave as expected and produce consistent results. The component that has not been tested using this procedure is the Graphical GUI to ASCII MCalc translation process.

7.4 MANUAL TESTING

Some components that comprise the human – computer interaction part of the project could not be tested procedurally. For this testing, special patterns were input manually to verify the correctness of the implementation. This was the main method that was used during the early stages of the development since most major bugs could be detected this way. It does not prove completeness, nor quality but it gives certain guarantees that widely known cases are being simulated correctly.

Other aspects, specifically the quality and the intuitiveness of the GUI could be also evaluated. Feedback was collected from people already familiar with the concept of MBQC patterns. They were able to use the graphical interface to set the input for the simulation and obtained results which were consistent with the manual theoretical predictions.

Some of the patterns that were tested are quite computationally intensive. They are extremely tedious to be simulated with pen and paper. A good example is the Figure 8 Swap example which if simulated using pen and paper would mean manipulation of 256 coefficients and keeping track of 256 bra-kets and performing 6 measurements with the corresponding corrections. Nevertheless the outputs of the circuits that have been tested is known. The simulator is successfully able to deliver the expected output for all of them. Below are examples of some of the MBQC patterns that were manually simulated and verified.

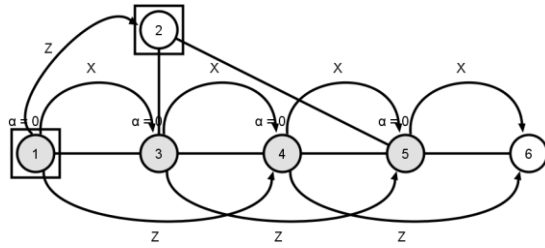


Figure 7 Identity

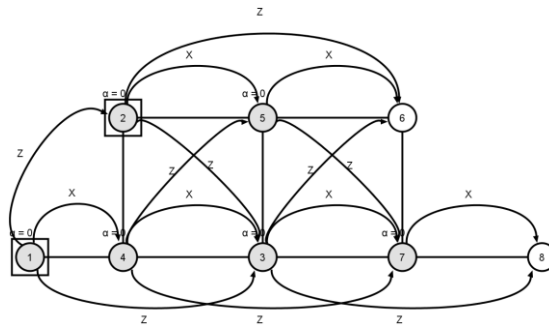


Figure 8 Swap

Other patterns that have been verified using this method are two versions of CNOT, CNOT+Hadamards and the quantum teleportation example.

An advice was made by Einar Pius to mention this small note. A special care should be made to the way outputs are numbered. This could be a source of confusion since the resulting bra-ket is reported in ascending order of the qubit id. When handling the examples manually, this is usually not noted and if not enough care was taken it may look like the output from the simulator has its inputs swapped. In such a case take a closer look in the input MBQC pattern and check whether the ids of the output qubits are correct.

Overall the manual testing resulted in those famous quantum computing patterns being run correctly. This would mean that the system as a whole is working as expected for the provided examples. The testing has included all of the components of the simulator (symbolic algebra, ASCII Mcalc parser and quantum operations), including the graphical to MBQC translator.

8 FUTURE WORK

The works so far has introduced a base framework for running MBQC patterns and creating them with the means of a graphical user interface. The system is usable and is proven to be accurate. The main goals of the project have been achieved, but the purpose of the project is to be the foundation of a comprehensive and thorough toolkit for Measurement Based Quantum Computing simulation and evaluation.

Several ideas were discussed as features that could be developed in future. One idea was to develop a tool that would allow for detailed step – by – step simulation execution. Knowing intermediate answers could allow theoretical researchers to explore non-deterministic patterns which are currently an active area of interest. It could also allow for exploring different branches and identifying at which point the non-deterministic

behavior starts to occur. Unfortunately, the development of such tools was not able to fit into the schedule of the current project so it is left as a possibility for future implementation.

Some graphical tweaks could be implemented in future – like zoom feature or feature that hides certain objects on the screen so the visibility could be improved. Other user accessibility features could be added such as changing the font size and font color according to user preferences. Exporting the visualizer content to an image is also an easy-to-implement feature that could be introduced.

Another possibility would be to allow for a big number of qubits to be simulated by modifying the *SystemMatrix* class so it buffers its coefficients on the hard drive. The infrastructure is flexible enough to allow for such a change without the rest of the system needing to be modified. Good practices like sequential data access would insure good performance even on such physical media.

Parallelization of the computation could be also done. The operators act on each coefficient at a time and do not depend on other parts of the coefficient matrix. This means that every action of an operator to a coefficient is independent and could be parallelized extremely well. This could allow for offloading the computation to a cloud computing resource. In order to achieve this, a modification of the quantum operators and the system matrix will need to be done.

To wrap it up, this project is an excellent base project for developing a comprehensive toolkit for further research in MBQC. Since the conceptual design and engine is already implemented, it is straightforward to add functionality that could fit within the provided framework. This would mean that the possibilities of extending the library are broad and very little time would be needed to develop new features because of the already existing infrastructure. The development of the underlying code was designed with extensibility in mind which should further ease the task.

9 BIBLIOGRAPHY

TODO

10 APPENDIX: USER GUIDE

This could serve as a short user guide. It is not exhaustive but will show what a user unfamiliar with the simulator will encounter.

Make sure you have Java Runtime Environment installed. JRE could be obtained from Java's website⁹. Download the executable **mbqsim.jar**¹⁰. Double click to launch it. You will be presented with the home screen.

⁹ <http://www.java.com/getjava/>

¹⁰ <https://github.com/martinmarinov/HonProj/blob/master/release/mbqcsim.jar>