UNIVERSITY *of* TASMANIA

School of Natural Sciences
Discipline of Mathematics

QUANTUM COMPUTATION IN MIXED BINARY-TERNARY SYSTEMS

by

Jarvis Carroll, B.Sc.

November 2020

Submitted in partial fulfilment of the requirements for the Degree of Bachelor of Science with Honours

Supervisors: Doctor Michael Cromer and Doctor Jeremy Sumner

I declare that this thesis contains no material which has been accepted for a degree or diploma by the University or any other institution, except by way of background information and duly acknowledged in the thesis, and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due acknowledgement is made in the text of the thesis.

Signed:

Date:

ABSTRACT

Quantum computation is the study of how quantum mechanics can be utilised to reliably compute or simulate results that are significantly harder to compute when done by classical means. Existing quantum computation research focuses overwhelmingly on binary quantum logic, the logic of the 'qubit', due to this being both the simplest and the most familiar number system for performing computation. Quantum computation does not have to operate with binary quantum data, however, and calculation is theoretically possible with ternary 'qutrits', or higher, or even with quantum systems that utilise a mixture of binary and ternary, or other quantum logic systems simultaneously. These systems might store data more efficiently, might scale more rapidly and might present richer or more novel techniques for calculation, or might prove detrimental; we need to develop a theory of non-binary quantum logic in order to resolve these possibilities.

In this thesis we present a detailed description of what a quantum algorithm is and the basic operations available to a qubit computer, before discussing existing literature surrounding proposed devices and algorithms for computation on ternary or 'qutrit' computers. We then convert these existing ternary algorithms into algorithms that act on mixed quantum computers, with access to both qubits and qutrits. After this we describe a basic set of gates already known to be capable of performing generic or universal computation on binary quantum computers, and generalise these to mixed quantum computers as well. After this we present the results of a basic algebraic search implemented in the C programming language, for finding finite and infinite groups of quantum operations, and finally discuss the many novel implications of what was found and the most obvious questions for future research prompted by these findings.

ACKNOWLEDGEMENTS

I would of course like to thank my supervisors Michael Cromer and Jeremy Sumner for their support in this project, who were very generous in taking this project on, especially Michael, who helped me learn everything I know about quantum computation, despite only being a guest at the university.

Further, I would like to thank the friends and peers with whom I have shared an office and classes this year, primarily Mae, Luke, Cassady, Georgia, and Larissa, for sharing in the journey we have just been through, of writing a thesis during a pandemic, making the year much less alienating than it would have been otherwise.

Finally, I would like to thank all of my lecturers from this year and past years for encouraging me and bringing me to this point in my education. This thesis is ultimately the result of five years of interaction with this university and its wonderful staff.

TABLE OF CONTENTS

Introduction

Quantum computation is a contemporary paradigm of computation based on manipulating the quantum behaviour of particles, which leverages phenomena such as superposition and entanglement of 'qubits', quantum analogues to the classical bit, and could theoretically compute useful results that would, for example, take millenia for a classical computer to calculate. Increasingly powerful quantum computers are being realized in practice, and the theory of quantum algorithms and quantum information is growing as well; both of these things are necessary for real life quantum computation to achieve results not already available in classical contexts.

One emerging topic within the field of quantum computation is the exploration of devices and algorithms that utilise logic systems other than the binary system familiar to classical computer science, since these alternative number systems present themselves readily within the quantum mechanics of certain physical systems. A related topic that is newer still is the question of what devices and algorithms might utilise logic systems that *mix* binary, ternary, and higher quantum data, the simplest of which is the mixed binary-ternary quantum system. Mixing binary and ternary is of particular interest since we might be able to combine the strengths of both systems, using binary data and the relatively simple binary algorithms when convenient, but utilising the increased novelty and complexity of ternary logic whenever there is an obvious way of capitalising on this novelty.

In Chapter 1 we go to great lengths to establish the foundational concepts of linear algebra, quantum mechanics, and quantum computation that are already understood and used to construct basic algorithms in qubit contexts, primarily restating the ideas described in the textbook [11]. We then describe the basic notation and behaviour of ternary and higher analogues to these operations, all things which are in common usage in the literature. This discussion is long, and in Appendix A one can find further details and definitions for the basic concepts upon which linear algebra is itself defined. The reader may not need to follow all of this discussion, in which case they can skip fa-

miliar content, until as late as Section 1.4.2, where we describe the specific ways that we need to notate quantum operations and quantum circuits in order to keep track of the many indeces that can vary independently in a mixed quantum computer.

With this in place we continue in Chapter 2 to describe known results of ternary quantum logic, and 'qutrit' computers, centering on the paper [3] which presents two algorithms for implementing integer addition of trits in a quantum computer. We describe the algorithms, and then present original circuits which perform the same calculations, but using qubits instead of qutrits where possible. We do this without having a particular physical implementation in mind, which blinds us to the performance characteristics that the original algorithms were better able to optimise and discuss. We do find, however, that the resulting circuits are simpler than the original qutrit circuits, suggesting that mixed logic is in fact capable of combining the strengths of individual logic systems. We also discuss analytical techniques presented in [3] for representing ternary gates as polynomials mod 3, which do not easily generalise to mixed systems, suggesting that there is hidden structure exclusive to mixed logic that does not exist in ternary logic, which may either be beneficial or detrimental.

Then, in Chapter 3, we discuss the topic of universal computation, which explores sufficient conditions for a quantum computer to be capable of executing arbitrary quantum algorithms. We follow the results of [11] and its sources [2, 6] for achieving universal computation in a qubit computer with standard gate sets, and provide original but straight-forward generalisations of these results to mixed computers, finding that tightly analogous sets of elementary operations are sufficient in mixed contexts as long as at least one qubit is available.

After this, in Chapter 4 we describe how an important set of operations called the Clifford operations do not provide any way of transmitting data between a qubit and a qutrit, dramatically changing what can be done with such Clifford operations. We discuss how one property of the Clifford operations – that they form a finite group – might apply to other sets of operations as well, including sets that can transmit data between qubits and qutrits, and describe the way that [6] applied algebraic number theory to prove results about operations that we can use to show they form infinite groups.

We then perform a programmatic search for groups of matrices to see which are finite and which are infinite. The search is implemented in the C programming language, without any external dependencies, using integer/rational arithmetic to keep track of the algebraic roots of unity that occur in the binary and ternary Clifford groups. We describe the algorithms used, including

a summary of the open addressed hash map that we implement, before stating the results of the search, including novel operations that appear in groups which we either prove or conjecture to be infinite.

At the end of all of this in Chapter 5 we summarise the key points that we found in our original research, and point to the vast possibilities for future research in mixed quantum systems directly following from the discoveries and literature reviewed in this thesis.

Chapter 1

Preliminaries

1.1 Algebra of Unitary Matrices

In quantum mechanics it will turn out crucial to have a strong theory of unitary operations acting linearly on complex-valued objects, so to that end we shall define these concepts and their notation here.

1.1.1 Hilbert Spaces

Cartesian coordinates provide a powerful abstract way of reasoning about physical space as the combination of three variables, or conversely a way of visualizing combinations of variables as spanning planes or volumes within a physical space. Hilbert spaces are a description which abstracts the Cartesian coordinate system even further, describing a much larger class of mathematical objects with similar geometric properties, including the space of possible states that a quantum object can take.

Hilbert spaces are a kind of Vector space defined explicitly in Section A.3. Specifically a *Hilbert Space* is a vector space with an inner product, that is closed under limits. Naturally the field defining the coordinates of a complete vector space ought to be either $\mathbb{R}$ or $\mathbb{C}$, and in this thesis we will always use $\mathbb{C}$. In Section 1.2 we will see that quantum computation most commonly acts on finite dimensional Hilbert spaces, which are structurally equivalent to $\mathbb{C}^N$,

as outlined in Theorem A.16. $\mathbb{C}^N$ is the set of N-dimensional column vectors

$$\left\{ \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{N-1} \end{bmatrix} \middle| \, a_0, a_1, \ldots a_{N-1} \in \mathbb{C} \right\},$$

equipped with the usual definitions of addition and scaling,

$$c_0 \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{N-1} \end{bmatrix} + c_1 \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{N-1} \end{bmatrix} = \begin{bmatrix} c_0 a_0 + c_1 b_0 \\ c_0 a_1 + c_1 b_1 \\ \vdots \\ c_0 a_{N-1} + c_1 b_{N-1} \end{bmatrix},$$

along with the simplest possible inner product,

$$\left(\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{N-1} \end{bmatrix}, \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{N-1} \end{bmatrix} \right) = \begin{bmatrix} a_0^* & a_1^* & \cdots & a_{N-1}^* \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{N-1} \end{bmatrix} = \sum_{i=0}^{N-1} a_i^* b_i.$$

One caveat to this equivalence is the concept of basis-independence, that many concepts in linear algebra are made useful by the fact that they give the same result regardless of which basis is used for the space in question. To show that something is basis independent, it is sufficient and often convenient to define it directly in terms of the inner product and vector operations, rather than the coordinate system induced by a given basis. As an example the ℓ^2 norm in $\mathbb{C}^N$ does not depend on the basis used,

$$\|v\|_2 = \sqrt{(v, v)} = \sqrt{\sum_{i=0}^{N-1} |v_i|^2}.$$

1.1.2 Dirac Notation

There are a number of notations available for reasoning about linear algebra and vector spaces, but we shall see that in order to reason about quantum algorithms we will rely heavily on the following techniques:

- linear operators defined in terms of inner products

- change of basis via unitary operators
- change of index set when summing vectors

For this collection of techniques the Dirac notation for vectors and linear operators is particularly well suited.

The main feature of Dirac notation is the ket, where a bar and angle bracket are used to distinguish a symbol as representing a vector: $|v\rangle$, $|*\rangle$, $|0\rangle$, $|i\rangle$, etc.

The most common vectors used are the canonical basis vectors, and in Dirac notation it is natural to use the index of each basis vector as the symbol *for the vector itself*, e.g.

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}, \qquad |1\rangle = \begin{bmatrix}0\\1\end{bmatrix}.$$

The inner product of two vectors ($|u\rangle, |v\rangle$) is normally written $\langle u|v\rangle$, called a 'bra-ket'. This is inspired by the notation $\langle u, v\rangle$, used before Dirac, but allows an elegant representation of co-vectors. *Co-vectors* linearly map a vector to a scalar; in an inner product space every vector has a natural co-vector defined using the inner product. In Dirac notation we can simply omit the ket from an inner product to notate a co-vector,

$$\langle u| : \mathbb{C}^N \to \mathbb{C}$$
$$\langle u| \equiv |v\rangle \mapsto \langle u|v\rangle .$$

1.1.3 Linear Operators

If U and V are Hilbert spaces, (or generally vector spaces) then a map $A : U \to V$ is a linear operator if it satisfies

$$A(a|u\rangle + b|v\rangle) = aA|u\rangle + bA|v\rangle$$

for arbitrary $|u\rangle \in U$, $|v\rangle \in V$, $a, b \in \mathbb{C}$.

If we have vectors $|u\rangle \in U$ and $|v\rangle \in V$ then we can use the inner product to define a simple linear operator $|v\rangle\langle u|$ equal to the map

$$\langle u'| \mapsto \langle u|u'\rangle |v\rangle .$$

We call this map the *outer product* of $|v\rangle$ and $|u\rangle$, since in this notation it appears to be the opposite of an inner product. The fact that any outer

product is a linear operator is a consequence of the inner product being linear in its second argument, one of the defining properties of Hilbert spaces.

Note that if we reverse the notation for scaling a vector we get the expression $|v\rangle \langle u|u'\rangle$, which looks very similar to the application $|v\rangle \langle u| (|u'\rangle)$, again identifying the map with the object it produces, just as was done in defining co-vectors.

As we have discussed, when dealing with any finite dimensional Hilbert space we can assume the space is equivalent to some complex vector space $\mathbb{C}^N$, in which case matrix representations of a linear operator become available. We derive matrix representation explicitly, since the relevant technology is readily available from Dirac notation. First observe that any linear operator A say, is determined uniquely by the image of each basis vector $|j\rangle$, say $A|j\rangle = |v_j\rangle$. Consider an arbitrary application of A,

$$\begin{aligned} A\left(\sum_{j=0}^{N-1} u_j |j\rangle\right) &= \sum_{j=0}^{N-1} u_j A|j\rangle \\ &= \sum_{j=0}^{N-1} u_j |v_j\rangle . \end{aligned}$$

This turns out to be the same result that we get from applying the sum of outer products

$$\begin{aligned} \left(\sum_{j=0}^{N-1} |v_j\rangle \langle j|\right)\left(\sum_{i=0}^{N-1} u_i |i\rangle\right) &= \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} u_i \langle j|i\rangle |v_j\rangle \\ &= \sum_{j=0}^{N-1} u_j |v_j\rangle . \end{aligned}$$

Since these two maps are equal on their whole domain, they are the same, i.e.

$$A = \sum_{j=0}^{N-1} |v_j\rangle \langle j| .$$

Further if the codomain is also finite dimensional then we can repeat this process. To show this, observe that the outer product notation $|v\rangle \langle u|$ is linear on $|v\rangle$, regardless of which vector $|u'\rangle$ it is applied to, by

$$(a|v_1\rangle + b|v_2\rangle) \langle u|u'\rangle = a|v_1\rangle \langle u|u'\rangle + b|v_2\rangle \langle u|u'\rangle .$$

Now suppose the image vectors $|v_j\rangle$ are in $\mathbb{C}^M$ and have coordinates themselves, $|v_j\rangle = \sum_i^{M-1} a_{ij} |i\rangle$, then

$$A = \sum_{i,j} a_{ij} |i\rangle \langle j| ,$$

that is, all linear operators are a linear combination of outer products between basis vectors, making these outer products a basis for the (Hilbert) space of linear operators between finite dimensional Hilbert spaces. We have derived a way of representing linear operators $\mathbb{C}^N \to \mathbb{C}^M$ as an array of $N \times M$ scalar coordinates, which is the exact structure we want from a matrix representation.

For example any linear operator $A : \mathbb{C}^2 \to \mathbb{C}^2$ can be written as follows:

$$\begin{aligned} A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} &= a \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + b \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} + c \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} + d \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \\ &= a |0\rangle \langle 0| + b |0\rangle \langle 1| + c |1\rangle \langle 0| + d |1\rangle \langle 1| \end{aligned}$$

Further, by composing two linear maps that are in this previous outer product form, we get

$$\left(\sum_{i,j} a_{ij} |i\rangle \langle j| \right) \left(\sum_{j',k} b_{j'k} |j'\rangle \langle k| \right) = \sum_{i,k} \left(\sum_j a_{ij} b_{jk} \right) |i\rangle \langle k| ,$$

the well known formula for matrix multiplication.

When defining a linear operator $L : \mathbb{C}^N \to \mathbb{C}^N$ we can define it as the *linear extension* of a more succinct map $\phi : \{|i\rangle \mid 0 \leq i < N\} \to \mathbb{C}^N$, by first defining $|v_j\rangle = \phi(|j\rangle)$ and then setting $L = \sum_j |v_j\rangle \langle j|$. This is simply a matrix whose columns are the vectors $|v_j\rangle$.

1.1.4 The Hermitian Adjoint and Diagonalisable Matrices

An aspect of linear algebra essential to quantum computation is diagonalisation. We shall describe a number of properties that a square matrix can have, in terms of both its Hermitian Adjoint, and its eigenvalues, but first we define these concepts.

The *Hermitian Adjoint of a linear operator* A from a Hilbert space U to a Hilbert space[1] V is the unique linear operator $A^\dagger$ with the property that $|v\rangle$

[1] or merely inner product spaces

and $A|u\rangle$ have the same inner product as $A^\dagger|v\rangle$ and $|u\rangle$, regardless of $|u\rangle$ and $|v\rangle$. It turns out that the matrix representation of $A^\dagger$ is exactly the complex conjugate of the transpose of A, so we define the *Hermitian Adjoint of a matrix* A to be this matrix $A^\dagger$ formed by the complex conjugate of the transpose of A. As an example observe the following matrix A and its Hermitian adjoint,

$$A = \begin{bmatrix} 1 & 1+i \\ 0 & 1 \end{bmatrix}, \qquad A^\dagger = \begin{bmatrix} 1 & 0 \\ 1-i & 1 \end{bmatrix}.$$

Note now that in the vector space $\mathbb{C}^2$ the complex conjugate of the transpose of a vector has the same behaviour as a co-vector,

$$\begin{aligned} |u\rangle^\dagger |v\rangle &= \begin{bmatrix} a \\ b \end{bmatrix}^\dagger \begin{bmatrix} c \\ d \end{bmatrix} \\ &= \begin{bmatrix} a^* & b^* \end{bmatrix} \begin{bmatrix} c \\ d \end{bmatrix} \\ &= a^*c + b^*d \\ &= \langle u|v\rangle . \end{aligned}$$

This generalizes to any vector in any Hilbert space. We will not dwell on what it means to interpret vectors and co-vectors as adjoint linear operators, but will nonetheless talk about this operation as the adjoint and use the notation $|v\rangle^\dagger$ when needed.

Associativity of the matrix product also lets us represent the outer product $|u\rangle\langle v|$ using the Hermitian adjoint, as the matrix product $|u\rangle|v\rangle^\dagger$. This can be used to prove that $(|u\rangle\langle v|)^\dagger = |v\rangle\langle u|$

Now we move on to the eigenvector problem, which is the problem of finding a scalar λ and non-zero vector $|v\rangle$ so that $A|v\rangle = \lambda|v\rangle$. Such a $|v\rangle$ is called an *eigenvector* of A, and such a λ is called the *corresponding eigenvalue* of A. For example if A has a non-trivial null-space, then any non-zero vector in the null-space of A will be an eigenvector of A, and $\lambda = 0$ will be a corresponding eigenvalue of A, i.e. $A|v\rangle = 0|v\rangle = 0$.

If $|v\rangle$ is a normalized eigenvector of A, with eigenvalue λ, then the matrix $B = \lambda|v\rangle\langle v|$ will also satisfy the eigenvalue problem $B|v\rangle = \lambda|v\rangle$. Further, any vector orthogonal to $|v\rangle$ will be in the null-space of B. This means that if all of the eigenvectors of A are orthogonal to eachother, then we can write A as a sum of such B operators/matrices. This structure $A = \sum_i \lambda_i |v_i\rangle\langle v_i|$ tells us that A acts like a diagonal matrix on its eigenvalues, which we call the *diagonal representation* of A, and for this reason call any such A *diagonalisable*.

As an example, the following matrix Z is already diagonal and so has a straight-forward diagonal representation:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = 1\,|0\rangle\langle 0| - 1\,|1\rangle\langle 1|$$

The property of being diagonalisable can be characterised very powerfully using the Hermitian adjoint, using a major result of linear algebra called the spectral theorem of 'normal' matrices. A matrix is *normal* if it satisfies $A^\dagger A = AA^\dagger$, and the spectral theorem of normal matrices says that a matrix is diagonalisable if and only if it is normal. When a matrix is normal, and hence diagonalisable, additional properties become easier to analyse as well. What follows is a list of different classes of normal matrix, each defined by a property of the matrix, and by an equivalent property of all of its eigenvalues:

Hermitian matrix:	$A^\dagger = A$	$\iff \lambda \in \mathbb{R}$
Unitary matrix:	$A^\dagger = A^{-1}$	$\iff \lvert\lambda\rvert = 1$
Positive normal:	$\langle v\rvert A\lvert v\rangle \in \mathbb{R}, \geq 0$	$\iff \lambda \in \mathbb{R}, \geq 0$
Positive definite normal:	$\langle v\rvert A\lvert v\rangle \in \mathbb{R}, > 0$	$\iff \lambda \in \mathbb{R}, > 0$
Projection matrix:	$A^2 = A$	$\iff \lambda \in \{0, 1\}$

The most relevant of these is the unitary matrix, whose defining property can also be written $AA^\dagger = I$, which is simply the matrix equation corresponding to the fact that the columns of A form an orthonormal basis. In addition to the above, another characterisation of unitary matrices is that they preserve inner products, which follows naturally from the matrix interpretation of inner products, by

$$(A\,|u\rangle\,, A\,|v\rangle) = (A\,|u\rangle)^\dagger A\,|v\rangle = \langle u|\,AA^\dagger\,|v\rangle = \langle u|v\rangle\,.$$

This means in particular that a unitary matrix will preserve ℓ^2 norms, and will therefore induce an invertible operation on any (complex, centre at origin) N-sphere $\{|v\rangle \;|\; \langle v|v\rangle = r^2\}$.

Finally, if P is a polynomial then we can evaluate it on a square matrix A as well, and it is easy to show that this polynomial 'applies itself' to the eigenvalues of A, in the sense that if $A\,|v\rangle = \lambda\,|v\rangle$ then $P(A)\,|v\rangle = P(\lambda)\,|v\rangle$.

Now any analytic function can be written as the limit of some polynomials; most notable for us is the exponential

$$e^x = \sum_n^\infty \frac{x^n}{n!}.$$

By applying each of these polynomials to a square matrix we can derive an analytic definition of the exponential of a matrix, or any other analytic function which in the case of diagonalisable matrices looks like

$$\begin{aligned} f\left(\sum_i \lambda_i |i\rangle\langle i|\right) &= \lim_{L\to\infty} \sum_{n=0}^{L} P_i \left(\sum_i \lambda_i |i\rangle\langle i|\right) \\ &= \lim_{L\to\infty} \sum_{n=0}^{L} \sum_i P_i(\lambda_i) |i\rangle\langle i| \\ &= \sum_i \lim_{L\to\infty} \left(\sum_{n=0}^{L} P_i(\lambda_i)\right) |i\rangle\langle i| \\ &= \sum_i f(\lambda_i) |i\rangle\langle i| . \end{aligned}$$

This can be very useful for solving certain matrix equations such as $A^2 = B$ having the solution $A = \sqrt{B}$. It is also useful for mapping between Hermitian and unitary matrices with the map $U = e^{iH}$; since each eigenvector $H|v\rangle = \lambda|v\rangle$ will have a real eigenvalue λ; we have by definition $U|v\rangle = e^{i\lambda}|v\rangle$, meaning all of the eigenvalues of U will have unit complex modulus, making it unitary.

1.1.5 Kronecker Products

We have made explicit the way in which vectors, co-vectors, and linear operators are represented as matrices, that is as arrays of complex numbers. An operation that is useful for all of these objects is the tensor product, and while the tensor product can be described in the abstract as an operation between Hilbert spaces, the only cases of interest here are complex-valued matrices, whose tensor products are described by a much more concrete operation called the Kronecker product.

If A is an $M_1 \times N_1$ matrix with elements a_{ij}, and B is an $M_2 \times N_2$ matrix with elements b_{ij} then the *Kronecker product* $A \otimes B$ will be an $M_1 M_2 \times N_1 N_2$ matrix which in block matrix form will look like

$$A \otimes B = \begin{bmatrix} a_{0,0}B & a_{0,1}B & \dots & a_{0,N_1-1}B \\ a_{1,0}B & a_{1,1}B & \dots & a_{1,N_1-1}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{M_1-1,0}B & a_{M_1-1,1}B & \dots & a_{M_1-1,N_1-1}B \end{bmatrix}.$$

Compare this to the matrix representation of A itself,

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,N_1-1} \\ a_{1,0} & a_{1,1} & \dots & a_{1,N_1-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M_1-1,0} & a_{M_1-1,1} & \dots & a_{M_1-1,N_1-1} \end{bmatrix}.$$

Written more compactly, if $A \otimes B = C$ has elements $c_{M_2 i_1 + i_2, N_2 j_1 + j_2}$, where i_1 and j_1 are the indices of each block, and i_2 and j_2 are the indices of each entry within each block, then these elements of C are exactly the products of elements of A and B, by the formula

$$c_{M_2 i_1 + i_2, N_2 j_1 + j_2} = a_{i_1 j_1} b_{i_2 j_2}.$$

In the special case where A is a vector $|u\rangle$ and B is a co-vector $\langle v|$, their Kronecker product will be exactly the matrix product AB, which is exactly the outer product $|u\rangle \langle v|$:

$$A \otimes B = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{M-1} \end{bmatrix} \otimes \begin{bmatrix} b_0 & b_1 & \dots & b_{N-1} \end{bmatrix} = \begin{bmatrix} a_0 b_0 & a_0 b_1 & \dots & a_0 b_{N-1} \\ a_1 b_0 & a_1 b_1 & \dots & a_1 b_{N-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M-1} b_0 & a_{M-1} b_1 & \dots & a_{M-1} b_{N-1} \end{bmatrix}$$

On the other hand if A and B are both vectors, in $\mathbb{C}^{M_1}$ and $\mathbb{C}^{M_2}$ respectively, then their Kronecker product will be a vector in the larger space $\mathbb{C}^{M_1 M_2}$, and in particular if A and B are canonical basis vectors $|j_1\rangle$ and $|j_2\rangle$ then their Kronecker product will be another canonical basis vector, $|M_2 j_1 + j_2\rangle$. This allows the whole space $\mathbb{C}^{M_1 M_2}$ to be spanned by Kronecker products of $\mathbb{C}^{M_1}$ and $\mathbb{C}^{M_2}$.

The Kronecker product satisfies the following algebraic properties, all of which follow directly from their relevant definitions:

- $(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$,
- in particular $(A \otimes B)(|u\rangle \otimes |v\rangle) = (A|u\rangle) \otimes (B|v\rangle)$,
- and $|u'\rangle |v'\rangle \langle u| \langle v| = (|u'\rangle \langle u|) \otimes (|v'\rangle \langle v|)$,
- $I_{d_1} \otimes I_{d_2} = I_{d_1 \times d_2}$,
- $(A \otimes B)^\dagger = A^\dagger \otimes B^\dagger$,

- in particular $(|u\rangle \otimes |v\rangle)^\dagger = \langle u| \otimes \langle v|$,
- if λ is a scalar, then $\lambda(A \otimes B) = (\lambda A) \otimes B = A \otimes (\lambda B)$,
- in particular $\lambda(|u\rangle \otimes |v\rangle) = (\lambda |u\rangle) \otimes |v\rangle = |u\rangle \otimes (\lambda |v\rangle)$.

With these properties we can show that the Kronecker product of two unitary matrices will be unitary:

$$\begin{aligned}(A \otimes B)(A \otimes B)^\dagger &= (A \otimes B)(A^\dagger \otimes B^\dagger) \\ &= AA^\dagger \otimes BB^\dagger \\ &= I \otimes I \\ &= I.\end{aligned}$$

We can also use the property that $|u'\rangle |v'\rangle \langle u| \langle v| = (|u'\rangle \langle u|) \otimes (|v'\rangle \langle v|)$ to see that the outer products $|i\rangle \langle j|$ which act as a canonical basis for the space of linear operators, can be combined with the Kronecker product to create a basis for linear operators on larger vector spaces.

Considering the degrees of freedom involved, we expect that in general there are many $M_1M_2 \times N_1N_2$ matrices that are not the Kronecker product of an $M_1 \times N_1$ matrix with an $M_2 \times N_2$ matrix. Written in block matrix form it is easy to tell whether or not a matrix C is a Kronecker product $A \otimes B$, by checking that every block B_{ij} is a scalar multiple of any non-zero block B. The entries of A are simply these scalar multiples, solving $B_{ij} = a_{ij}B$, and hence $C = A \otimes B$. (and if there are no non-zero blocks then $C = 0 = 0 \otimes 0$)

1.1.6 The Unitary Group

So far we have discussed a number of useful algebraic properties exhibited by unitary matrices, but the most basic properties of unitary matrices are of course their forming a group. Let A and B be unitary, and observe:

- Product of unitaries is unitary: $(AB)^\dagger = B^\dagger A^\dagger = B^{-1}A^{-1} = (AB)^{-1}$,
- Matrix products are always associative, (since composition of any operator will be associative)
- The identity matrix is unitary: $I^\dagger = I = I^{-1}$
- $A^{-1} = A^\dagger$ always exists and is unitary.

Quantum computation deals extensively with unitary operators, and so subgroups formed by particular sets of unitary operators will be a frequent point of discussion. As described in Section A.4, a subgroup of a group is simply a subset that is still a group when equipped with the same binary operation as the original. Our binary operation is matrix multiplication, and so the group of $N \times N$ unitaries written $U(N)$ is actually a subgroup of the much larger general linear group $GL(N, \mathbb{C})$, of $N \times N$ matrices with non-zero determinant.

A further subgroup of both of these is the symmetric group $\mathcal{S}_N$, which we represent as the set of $N \times N$ 'permutation' matrices. A *permutation* is an invertible function mapping a set to itself, and given a permutation σ on the canonical basis $\{|i\rangle \mid 0 \leq i < N\} \to \{|i\rangle \mid 0 \leq i < N\}$, we can extend this linearly to an invertible matrix $\mathbb{C}^N \to \mathbb{C}^N$ whose columns are all canonical basis vectors, being exactly the images of σ. We call such a matrix a *permutation matrix*, and note that the adjoint of its outer product form $P = \sum_i |\sigma(i)\rangle \langle i|$ will be $P^\dagger = \sum_i |i\rangle \langle \sigma(i)|$, giving

$$PP^\dagger = \sum_{i,j} |\sigma(i)\rangle \langle i|j\rangle \langle \sigma(j)| = \sum_i |\sigma(i)\rangle \langle \sigma(i)| = I.$$

The symmetric group of any finite set is always finite, and is typically the first group considered when exploring finite subgroups of $U(N)$.

1.2 Quantum Mechanics

Quantum mechanics is a general theory and mathematical framework for describing physical matter, and has become an essential component of some of the most empirically precise physical models ever described, such as quantum electrodynamics. Quantum mechanics is most famous for its proposal of physical phenomena radically different to that of the classical world, so long as the objects under consideration are able to achieve 'coherence', e.g. when sufficiently small or low in temperature. Quantum mechanics by itself is too general to make statements about how the world behaves, but is a consistent framework from which models such as the Schrödinger equation of the hydrogen atom or the Pauli equation of the electron can be derived.

Quantum computation is the study of how quantum phenomena can be used to effect computation. When designing an individual experiment or programmable quantum computer it is necessary to use a concrete, empirically verified model of physical phenomena, but the specific algorithms that can be run on a quantum computer turn out to be described sufficiently by the general

framework of quantum mechanics without much elaboration. Since this thesis is concerned with algorithms, we shall summarize the description of quantum mechanics given in the ubiquitous textbook "Quantum Computation and Quantum Information" by Nielsen and Chuang [11]. This will provide a foundation for explaining certain techniques and algorithms prominent in quantum computation, and their relationship to the questions we aim to explore in this thesis.

1.2.1 Postulate One: State Space

The first postulate of quantum mechanics is that any closed quantum system can be modeled by some complex Hilbert space. Specifically in the Schrödinger theory of quantum mechanics this will be a space of functions satisfying the linear differential equation called the Schrödinger equation, which will become relevant in Section 1.2.2. While many of these Hilbert spaces such as the free electron are of infinite dimension, the form of quantum computation most commonly understood, and most relevant to this thesis, is computation in finite dimensional Hilbert spaces, which occur naturally in many forms especially with particles trapped in one place, such as an electron bound to an atom. Once we have a finite dimensional Hilbert space of dimension N, we can apply the isomorphism outlined in Theorem A.16, modeling quantum systems directly as the space of column vectors $\mathbb{C}^N$.

We call the specific elements of $\mathbb{C}^N$ *state vectors* whenever they represent a state that a quantum computer can take, and we shall see soon that the state vectors are exactly the vectors of unit length in $\mathbb{C}^N$. This allows us to model quantum computation itself as an operation on these unit length state vectors, and to describe the initial, final, and intermediate states of any given computation as an individual state vector.

The simplest possible quantum system is one which can be modelled with the two-dimensional Hilbert space

$$\mathbb{C}^2 = \left\{ \begin{bmatrix} a \\ b \end{bmatrix} \;\middle|\; a, b \in \mathbb{C} \right\}.$$

Such a system is called a *quantum bit*, or a *qubit*.

We will see in Section 1.2.3 that we don't really have empirical access to the coordinates of the state vector of any given quantum system; they always appear to have an individual basis vector as their state, so from this perspective state vectors that aren't canonical basis vectors are said to be in a *superposition* of two or more canonical basis vectors, in the sense that they

hold multiple visible/measurable states at the same time, like two images or readings superposed over one another.

1.2.2 Postulate Two: Evolution Over Time

The second postulate of quantum mechanics is that the evolution of a closed quantum system after a given amount of time always corresponds to some unitary operation acting on the state vector of that system. The more general statement of this postulate is that the state vector $|\phi\rangle$ will satisfy the Schrödinger equation,

$$-\frac{ih}{2\pi}\frac{d}{dt}|\phi(t)\rangle = H|\phi(t)\rangle,$$

where h is the Planck constant, a known scalar constant, and H is the Hamiltonian operator, a Hermitian operator associated with the energy present in the system. By using the matrix exponential we can solve this equation, giving

$$|\phi(t)\rangle = \exp\left(\frac{i2\pi tH}{h}\right)|\phi(0)\rangle,$$

which, by fixing t, and combining the variables on the right hand side into a single matrix, can be written as

$$|\phi(t)\rangle = U|\phi(0)\rangle.$$

U will have eigenvalues of the form $\exp(i2\pi t\lambda/h)$, which are always unit modulus, meaning U is unitary, and thus after any fixed amount of time the evolution of a quantum system will correspond to some unitary matrix U.

A quantum computer, then, is simply a device that is able to manipulate the Hamiltonian H of a quantum system, for specific durations of time t, in order to effect a sequence of unitary operations. For example, suppose there is a qubit system such as an electron that can be in one of two spin states, or an ion that can be in ground or excited states, and there is an external electromagnetic field that can be activated on command which affects one of these qubit states without affecting the other, giving a Hamiltonian like

$$H = \begin{bmatrix} 0 & 0 \\ 0 & c \end{bmatrix},$$

then by setting $t = h/2c$ we get

$$\begin{aligned} U &= \exp\left(\frac{i2\pi tH}{h}\right) \\ &= \exp\left(\begin{bmatrix} 0 & 0 \\ 0 & \frac{i2\pi tc}{h} \end{bmatrix}\right) \\ &= \exp\left(\begin{bmatrix} 0 & 0 \\ 0 & i\pi \end{bmatrix}\right) \\ &= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}. \end{aligned}$$

In this thesis we will call a unitary matrix U an *elementary evolution* of a given quantum computer whenever we suppose or know that this quantum computer can implement some H, t pair that would effect this matrix U. Composing these elementary evolutions will always give some overall unitary matrix $U = U_1 U_2 \dots U_k$, so the core of quantum computation then, is to understand how computationally interesting unitary matrices can be implemented using elementary evolutions that could be realistically implemented by a quantum computer.

1.2.3 Postulate Three: Measurement

Although the quantum systems we are discussing are represented as having a continuous state space of possible state vectors, a fundamental (and titular!) peculiarity of quantum mechanics is that when observed, a quantum object will always appear to be in states that are elements of some orthonormal basis of the corresponding Hilbert space. What is more peculiar, and more well known of quantum mechanics, is that the state that is measured becomes the new state of the system.

Suppose that we have a quantum system with state vector

$$|\phi\rangle = \sum_{i=0}^{N-1} a_i |i\rangle ,$$

in some experiment where a measurement has N possible outcomes, corresponding to the N canonical basis vectors $|i\rangle$; mathematically, the third postulate states that, when measured, the measurement corresponding to the state $|i\rangle$ will occur with probability $|a_i|^2$, and after this point will actually be in this state, despite potentially being in a superposition before this point in

time. Given that the coordinates of a state vector are now interpreted as being related to probabilities, it is important that state vectors have unit length, i.e. that

$$\| |\phi\rangle \| = \sum_{i=0}^{N-1} |a_i|^2 = 1.$$

A defining feature of unitary matrices was that they preserve vector magnitudes, meaning this assumption will be preserved at all points in time in a quantum algorithm.

As an example consider the $|-\rangle$ state,

$$|-\rangle = \frac{1}{\sqrt{2}} |0\rangle - \frac{1}{\sqrt{2}} |1\rangle ,$$

which, when measured, will have probability $1/2$ of being in the $|0\rangle$ state, and probability $1/2$ of being in the $-|1\rangle$ state, although as far as the measurement is concerned we only see that it is some multiple of the $|1\rangle$ state.

Measurement does not have to be in the canonical basis of the coordinate system, and sometimes the result of a measurement will correspond to multiple basis vectors at once, leaving the state in a new superposition, so the most general form of this postulate is that when measured, as many as N distinct outcomes will be possible, and each of these outcomes will correspond to some non-zero projection operator P_i, for example the projections onto the canonical basis $P_i = |i\rangle \langle i|$. These projection operators must be mutually orthogonal, so that $P_i P_j = 0$ whenever $i \neq j$, and must cover the whole Hilbert space, in the sense that $\sum_i P_i |\phi\rangle = |\phi\rangle$, but are otherwise unconstrained by this general postulate of quantum mechanics. The probability of outcome i then, is not $|a_i|^2$, but $|P_i |\phi\rangle|^2$, and the state afterwards is the result of normalising $P_i |\phi\rangle$ to unit length,

$$|\phi'\rangle = \frac{1}{\|P_i |\phi\rangle\|} P_i |\phi\rangle .$$

In the case of a canonical basis measurement with $P_i = |i\rangle \langle i|$, the normalised state after measurement would not be $|i\rangle$, but $a_i' |i\rangle$ instead, where a_i' is the normalized complex number $a_i/|a_i|$, but we cannot infer the factor a_i' from measurement, so this does not make a practical difference when compared to our earlier description of measurement in the canonical basis.

This general form will come in useful in Section 1.2.6 when describing measurement of one object or particle at a time, in a composite system with multiple objects in it, but while developing the foundations of computation it is important for our assumptions about measurement to otherwise be as minimal as

possible. We require at least one form of measurement to be possible, or else we can never tell what the result of our algorithms were, so when discussing an N dimensional quantum system consisting of a single object or particle, we will assume that some measurement is available with N distinct outcomes, such as the $|i\rangle\langle i|$ measurement outcomes of our original description of measurement in the canonical basis. Since we are imagining these systems as consisting of a single object, whenever we introduce a system that is assumed to have some measurement available in this form, we call the system a *quantum object* or a *non-composite system.*

It turns out that this weak measurement assumption can be used to implement exactly our original measurement in the canonical basis; the N outcomes will be associated with some N corresponding projection operators P_i, but since their ranks must add up to N, they will all be rank 1, giving them an outer product form $|v_i\rangle\langle v_i|$, where $|v_i\rangle$ is some unit vector. Each of the $|v_i\rangle$ will be orthogonal, meaning we are really measuring in some orthonormal basis of the Hilbert space, which we will call the *computational basis* of the quantum system. Now these $|v_i\rangle$ in $\mathbb{H}$ are generally in a different space to the $|i\rangle$ of our original description of measurement, but by taking as a convention that the canonical basis vectors $|i\rangle$ correspond exactly to the computational basis vectors $|v_i\rangle$, we find that this description of measurement was sufficient all along. The correspondence between a general N-dimenional Hilbert space $\mathbb{H}$ and the more concrete space $\mathbb{C}^N$ was laid out in Theorem A.16, where some orthonormal basis of $\mathbb{H}$ was acquired, and used to define an invertible linear map to $\mathbb{C}^N$. Since this map happens to map the given basis to the canonical basis of $\mathbb{C}^N$, all we need to do to achieve this convention is to use the computational basis in this process.

1.2.4 Postulate Four: Composite Systems

Suppose that we have two quantum systems, which when closed and isolated from each other would be modeled as $\mathbb{C}^M$ and $\mathbb{C}^N$ respectively. When we allow these to interact with each-other, and form a larger closed composite system, we will need some distinct Hilbert space with which to model them. By the measurement postulate when we measure the first system we will observe one of as many as M distinct outcomes, and as many as N distinct outcomes in the second system, giving as many as $M \times N$ distinct outcomes overall. Intuitively the fourth postulate is that all $M \times N$ of these outcomes are available as orthogonal state vectors in the composite system, making the composite space the Hilbert space $\mathbb{C}^{MN}$.

Formally, the fourth postulate is that when combining two quantum systems,

the composite system is modelled not by pairs of state vectors $(|\phi\rangle, |\psi\rangle)$, but by linear combinations of such pairs. In general this is modelled using the tensor product, but since we are only concerned with finite dimensional spaces equipped with a particular computational basis, we model the composite space with linear combinations of Kronecker products, meaning our new Hilbert space is exactly $\mathbb{C}^{MN}$. We further require that state vectors have unit length in this composite space, so a general state vector in this composite system would be of the form

$$|\phi\rangle = \sum_{i,j} a_{i,j} |i\rangle |j\rangle ,$$

with $a_{i,j}$ all in $\mathbb{C}$, and $\sum_{i,j} |a_{i,j}|^2 = 1$.

As an example of a state vector in a composite system, consider the following pair of qubit states,

$$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \qquad |+\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}.$$

If we combine two qubit objects into a composite system, and these are their state vectors, the corresponding Kronecker state of the composite system would be

$$|1\rangle \otimes |+\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}.$$

Note that when taking the Kronecker product of multiple vectors such as $|v_1\rangle \otimes |v_2\rangle$, or multiple co-vectors for that matter, we can suppress the $\otimes$ operator, simply writing it as a concatenated sequence of kets like $|v_1\rangle |v_2\rangle$. This does not introduce any ambiguity since the matrix product of two vectors is not defined.

In classical computation we represent integers and other data as a sequence of small (typically binary) digits and in the same way we can represent integers as computational basis vectors $|i_1\rangle |i_2\rangle \ldots |i_n\rangle$ in some large system made from composing individual 'quantum digits', binary or otherwise. In other contexts it is common to take the corresponding sequence of digits as the identifier for a single ket $|i_1 i_2 \ldots i_n\rangle$ but we shall refrain from doing this here, since it poses no significant advantage when dealing with the most foundational aspects of quantum computation.

As was described in our original discussion of Kronecker products, not all quantum states in a composite system $\mathbb{C}^{MN}$ are of the form $|u\rangle \otimes |v\rangle$. States

that can't be represented as a single Kronecker product $|u\rangle \otimes |v\rangle$ are called *entangled states*. As an example observe the 'Bell' state

$$\frac{1}{\sqrt{2}}|0\rangle \otimes |0\rangle + \frac{1}{\sqrt{2}}|1\rangle \otimes |1\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix}.$$

As a block matrix this would be written

$$\frac{1}{\sqrt{2}} \begin{bmatrix} |0\rangle \\ |1\rangle \end{bmatrix},$$

and since $1/\sqrt{2}\,|0\rangle$ and $1/\sqrt{2}\,|1\rangle$ are not multiples of each other, we know that this Bell state is not a Kronecker product, making it an entangled state.

Seeing as classical systems aren't treated as having any form of superposition, there is no classical analogy for entangled states. It is worth noting that in classical systems a composite system would be understood to have a state space that is the Cartesian product of two individual state spaces, and that the Cartesian product of an M-dimensional vector space with an N-dimensional vector space would have $M + N$ degrees of freedom rather than the $M \times N$ degrees of freedom given by the vector spaces associated with composite quantum systems. Entanglement is a very important feature of quantum mechanics and quantum computation, and in fact all of the novel behaviour and power of quantum computation ultimately depends on entanglement, but we will not spell this out any further in this thesis, simply taking as a matter of course that unitary matrices acting on $\mathbb{C}^{MN}$ are going to be of a very kind to those acting on $\mathbb{C}^M \times \mathbb{C}^N$.

1.2.5 Evolution in Composite Systems

Just as with individual systems, a closed composite system evolves according to some unitary operator after any discrete time step. In the composite system $\mathbb{C}^{MN}$ this could be any $MN \times MN$ unitary matrix, depending on the elementary evolutions available to this composite system. If we have U_1 and U_2 available as elementary evolutions in each of the individual systems, then the Kronecker product $U_1 \otimes U_2$ would be available as an elementary evolution simply by allowing each of U_1 and U_2 to play out in parallel. For reference this would apply to any linear combination of Kronecker products according to the straight forward rule

$$\sum_i |u_i\rangle \otimes |v_i\rangle \mapsto \sum_i (U_1\,|u_i\rangle) \otimes (U_2\,|v_i\rangle).$$

As a more concrete example, consider a two-qubit system $\mathbb{C}^{2\times 2}$ where we want to apply H to the second qubit; this would be represented with the Kronecker product $I \otimes H$, which has the matrix representation

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}.$$

If all of the elementary gates available to a quantum computer can be represented as the Kronecker product of individual gates, and the initial state of the computer can be represented as the Kronecker product of individual states, then the computer will never be able to create entangled states, and will essentially be two distinct quantum computers running in parallel. This means that it is crucial for a composite quantum computer to have at least one operation that is not the Kronecker product of individual operations, and the most common of these elementary gates is the controlled increment gate, which acts as a permutation on the computational basis $|i\rangle |j\rangle \mapsto |i\rangle |i + j \mod N\rangle$. We call any operation that cannot be factored as a Kronecker product acting on each individual object in the quantum system *dependent*, due to the fact that the effect on one object is dependent on the state of other objects; correspondingly, we call operations *independent* if the cannot be factored in this way.

The controlled increment operation in binary systems is also called the controlled-not operation, which will act mod 2, and therefore simply swap the $|1\rangle |0\rangle$ state with the $|1\rangle |1\rangle$ state, while leaving $|0\rangle |0\rangle$ and $|1\rangle |1\rangle$ unchanged. This can not be represented as a Kronecker product of two individual operations, which can be seen in the matrix representation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

or in block form,

$$\begin{bmatrix} I & 0 \\ 0 & X \end{bmatrix}.$$

In a similar way no other controlled increment operation will be representable as a Kronecker product. In this example we see that the control operation can change the state of the second qubit, depending on the state of the first qubit.

This notion of controlled operations is useful to generalize to any unitary operation acting on a smaller quantum system, which will be discussed in more detail in the context of both binary and ternary quantum systems in Section 1.4.3.

1.2.6 Measurement in Composite Systems

Given a composite system formed from an M dimensional system and an N dimensional system, any measurements that are possible in the individual systems will be theoretically possible in the composite system as well. Given projection operators P_i that represent a possible measurement of the $\mathbb{C}^M$ system, the projection operators $P_i' = P_i \otimes I$ will together represent this same measurement within the composite system. Similarly if some projection operators Q_j represent a measurement of the $\mathbb{C}^N$ system then $Q_j' = I \otimes Q_j$ will represent this measurement within the composite system.

If the $\mathbb{C}^M$ system and the $\mathbb{C}^N$ system are both what we have been calling quantum objects, or non-composite systems, then we will have two measurements available, one in the computational basis of $\mathbb{C}^M$ and the other in the computational basis of $\mathbb{C}^N$. For example observe the following two-qubit state vector $|+\rangle|+\rangle$:

$$|+\rangle|+\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) = \frac{1}{2}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}$$

If we measured the first qubit then we would observe $|0\rangle$ with probability $1/4 + 1/4 = 1/2$, and similarly $|1\rangle$ with probability $1/2$. Upon measuring $|0\rangle$ the overall state would be $|0\rangle|+\rangle$, whereas upon measuring $|1\rangle$ the overall state would be $|1\rangle|+\rangle$.

While this is straight-forward for states that are simply the Kronecker product of two states, measuring entangled states can behave differently, and is essential in certain quantum algorithms. For example suppose we measured the first component of the Bell state $1/\sqrt{2}(|00\rangle + |11\rangle)$. As before we would find the first object in state $|0\rangle$ with probability $1/2$, and $|1\rangle$ with probability $1/2$, but after measurement the overall system would be in states $|00\rangle$ and $|11\rangle$ respectively. Measuring the first component of the system changed the second one! This is also the basis of a number of novel techniques in the related fields of quantum information theory and quantum cryptography.

Extending to systems with more than two objects, note inductively that if the $\mathbb{C}^N$ system is a composite system of some n quantum objects, with n measurements available, and the $\mathbb{C}^M$ system is a composite of m quantum objects, with m measurements available, then the $\mathbb{C}^{MN}$ system as a whole will have $n+m$ measurements available, i.e. one for each quantum object in the system. All of the algorithms we will discuss in this thesis will assume some composite system $\mathbb{C}^N$ formed by combining quantum objects $\mathbb{C}^{d_1}, \mathbb{C}^{d_2}, \ldots \mathbb{C}^{d_n}$ together into one system, giving n different measurements, with each measurement occurring in the computational/canonical basis of one of the objects. One might also describe such a system as a whole as having a computational basis consisting of its canonical basis vectors $|i_1\rangle |i_2\rangle \ldots |i_n\rangle$, being the basis corresponding to measuring all n objects simultaneously.

1.3 Quantum Computation

With quantum mechanics and the associated concept of a quantum algorithm in place, we can begin to reason about the techniques of quantum computation. To this end we shall briefly discuss the computer science of probabilistic vs deterministic algorithms, as well as the group theory of different sets of unitary matrices, important concepts that lay the foundation of technical discussions in the field of quantum computation. With this in place we shall be prepared to discuss the state of the field when it comes to quantum computation on objects with more than two computational basis states, and our generalisations of these findings to quantum computers that mix objects with two and more than two basis states.

1.3.1 Classical Computation in Quantum Algorithms

In classical computation we can imagine an algorithm or circuit mapping some M discrete states to some N discrete states, according to some function $f : \{0 \ldots M-1\} \to \{0 \ldots N-1\}$. (say $M = 2^m$, $N = 2^n$ where m and n are the number of physical wires leading in and out of the circuit) For example we could define the logical conjunction or AND map which maps pairs of bits $\{00, 01, 10, 11\}$ to single bits $\{0, 1\}$, under the rule

$$\mathrm{AND}(x) = \begin{cases} 1 & \text{if } x = 11 \\ 0 & \text{otherwise.} \end{cases}$$

Given such a map f we can define a linear operator by extending linearly on

the computational basis, defining $A_f |i\rangle = |f(i)\rangle$. Note that this gives a matrix whose columns are all computational basis vectors. For example our logical conjunction becomes the matrix

$$A_{\text{AND}} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

In quantum computation we require that all operations be reversible, unitary operations, which is clearly not the case for the AND operation. It is straight forward to show that A_f will be a unitary operator if and only if $M = N$, and f is invertible, i.e. if f is a permutation. When $M = N = 2$ we only have two such permutation matrices,

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad \text{and } X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

Maps that don't satisfy $M = N$ or f invertible can still be represented as a permutation matrix, by performing a reversible operation on some additional quantum registers, giving an operation in the larger space $\mathbb{C}^{M\times N}$ with the definition

$$B_f(|i\rangle \otimes |j\rangle) = |i\rangle \otimes |j + f(i) \mod N\rangle.$$

The inverse of this matrix is simply $|i\rangle |j\rangle \mapsto |i\rangle |j - f(i) \mod N\rangle$.

In the case of the binary AND map, B_{AND} will be a 8×8 permutation matrix known as the Toffoli gate. It can be shown that compositions of Toffoli gates acting on some number of bits can be used to implement any n-qubit permutation matrix as an algorithm on some larger number of qubits $n+m$, so long as the extra m qubits are initialized to known values.

Permutation matrices by themselves might not seem interesting, since they exclusively represent calculations that can be done in classical contexts, but in fact are crucial for many quantum algorithms, since they will act linearly on superposition states. For example if we consider the map $f(x) = x^2 \mod 16$, then the constructed matrix B_f acting on eight qubits will of course distribute linearly over any linear combination of basis states including the following:

$$\begin{aligned} B_f(|2\rangle |0\rangle + |5\rangle |0\rangle) &= B_f |2\rangle |0\rangle + B_f |5\rangle |0\rangle \\ &= |2\rangle |4\rangle + |5\rangle |9\rangle \end{aligned}$$

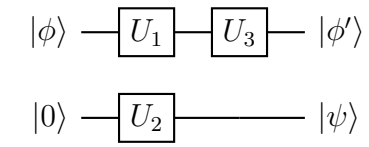
Further manipulations or measurement of the result of such a transformation can enable many powerful quantum algorithms including Shor's period finding algorithm. This means that permutation matrices are an important topic in

quantum computation, and a good deal of research has been and continues to be done to better understand how permutation matrices can be decomposed into efficient quantum algorithms.

1.3.2 Quantum Circuits

We have described quantum algorithms as a sequence of unitary transformations and measurements on quantum states represented using Kronecker products, but as the number of quantum objects gets large the relevant vectors and matrices grow exponentially, which is a problem because these exponentially large operations are precisely the ones that we can't calculate on a classical computer, so they are of the most interest. In order to notate these operations which could become quite large, we introduce the quantum circuit diagram.

The diagram of a quantum circuit features multiple horizontal lines representing distinct quantum objects, and with squares on these lines representing operations that should be performed on the corresponding objects over time:



This would represent two quantum objects, initially in states $|\phi\rangle$ and $|0\rangle$, i.e. the whole system was in state $|\phi\rangle \otimes |0\rangle$. Then $U_1 \otimes U_2$ is applied to this state, giving

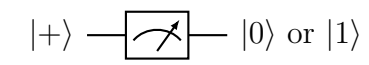
$$(U_1 \otimes U_2)(|\phi\rangle \otimes |0\rangle) = (U_1 |\phi\rangle) \otimes (U_2 |\phi\rangle).$$

After this $U_3 \otimes I$ is applied, giving the overall result

$$(U_3 \otimes I)((U_1 |\phi\rangle) \otimes (U_2 |\phi\rangle)) = (U_3 U_1 |\phi\rangle) \otimes (U_2 |0\rangle).$$

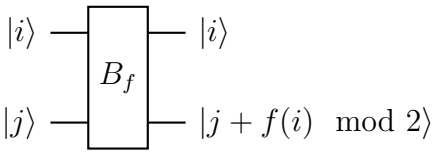
We also label these final states $|\phi'\rangle = U_3 U_1 |\phi\rangle$, and $|\psi\rangle = U_2 |0\rangle$.

These diagrams are inspired by classical circuit diagrams, and as such the horizontal lines are typically referred to as 'wires', but unlike classical circuits there is no such wire in practice. The horizontal axis of a quantum circuit is strictly chronological, as a single object evolves over time, usually while sitting still in space. Additionally wires can't split or merge, reflecting the fact that all quantum circuits are reversible between measurements. Measurements can still be included in a quantum circuit, however. For example measuring the $|+\rangle$ state can be represented with this circuit:



In general measurement can be quite flexible, with different bases of measurement, and with conditional execution of gates depending on the result of measurement, but we only need this simplest case of measurement in the computational basis.

Finally when something other than a Kronecker product of single-object matrices is applied, we can have boxes spanning multiple wires:



1.3.3 Probabilistic Algorithms

In computer science there is a distinction between deterministic algorithms, and randomized/probabilistic/non-deterministic algorithms. In summary a *deterministic algorithm* is a sequence of exact steps that can be executed in order to compute a result, whereas a *probabilistic algorithm* is permitted to rely on some source of random information to determine its control flow, meaning that the same input could result in many different outputs.

The advantage of probabilistic algorithms is that they can often avoid the worst-case performance associated with certain input states in deterministic algorithms; for example, many implementations of sorting algorithms will take much longer than usual to sort a list in ascending order if it is initially in descending order. At an intuitive level such worst-case input states tend to exploit the specific order in which an algorithm explores its possible solutions, and so since a randomized algorithm has no single order in which it might explore solutions, such worst-case inputs do not exist.

On classical computers deterministic algorithms are often more natural, and even when a randomized algorithm is used it will likely use a pseudo-random number generator in place of a true random source of information. This is heavily contrasted with quantum computation and quantum physics more generally, where measuring the state of a quantum object is inherently probabilistic, and physical implementations of quantum algorithms often introduce physical sources of error as well, as a result the probabilistic quantum algorithm is taken as the norm, with the exception of algorithms that only performs measurement on states that are already computational basis states, which when simulated theoretically will act deterministically. In physical quantum computers to date noise has always been present, and so even these algorithms that are deterministic when simulated end up being non-deterministic

in practice. This distinction is important to keep in mind, since it means that quantum algorithms that appear to perform well might be classical probabilistic algorithms in disguise, meaning that the quantum system did little more than provide a source of randomness. All of the famous algorithms based on the techniques of quantum Fourier transform, quantum search, and quantum simulation do not fall into this category, and are in fact faster at solving particularly large problems on a quantum computer than even a probabilistic classical computer.

A further distinction in computer science is made between *Las Vegas* algorithms that always yield a result, but have random run-time, and *Monte Carlo* algorithms, that yield some result in fixed run-time, but have a random chance of failing or producing a result that is incorrect. Often an algorithm in one of these classes can be converted into the other, Las Vegas algorithms that repeatedly search for a solution can be modified to yield a false negative after a fixed number of attempts, and Monte Carlo algorithms that may produce a false negative can check their solution using a deterministic algorithm, and retry until a valid solution is found. This means that in classical contexts this distinction is less important than that of deterministic vs probabilistic algorithms, since both classes can achieve similar things with the same resources. In quantum contexts however, algorithms are generally assumed to be Monte Carlo algorithms, since the sources of error discussed are sources of incorrect output, rather than sources of increased run-time. Further when a physical quantum computer is run, it is run repeatedly, often thousands of times, in order to determine the proportion of each output, so that one can tell which outputs occurred frequently enough to be the 'correct' result of the computation.

When constructing unitary matrices out of some elementary gate set, it becomes possible to use approximate constructions, since all quantum algorithms are Monte Carlo by default, and small changes in a state's coordinates tend not to significantly affect the overall probability distribution of the algorithm's outputs. Often quantum algorithms and their associated unitary matrices are implemented asymptotically, where the desired accuracy of the algorithm is taken as a parameter, and as this parameter gets smaller a longer sequence of elementary gates must be constructed to achieve this level of accuracy.

1.3.4 Error Correction

We have described how by controlling the external forces acting on a quantum system, we can implement some set of elementary evolutions on that system. In practice however, no matter how much engineering work we do to control an

environment, the forces we deliberately impose are not the only forces present. This means there is always some (hopefully small) quantity of noise affecting the system, and so the longer that a quantum computer is left running, the more deviation there will be between what state the quantum computer is in, and what theoretical state our model would predict it to be in. This is a major source of error in quantum computation, and since these errors persist across time, and even propagate between different quantum objects as those objects interact with each other, this imposes strict limits on how long a naive algorithm can run before the whole process is drowned in error.

The paper [1] describes many algorithms and demonstrates their implementation on the five-qubit `ibmqx4/ibmqx5` quantum computers, all of which were implemented at scales that would be trivial to implement classically instead, such as finding the smallest p so that $15^p - 1$ is divisible by 11. These implementations did not use any error correction, and so results that should have had a 0.5 chance of being measured in simulation, (among eight possible outcomes) were measured with a frequency of 0.25 in practice. This goes to show that if the algorithms had to compute on larger numbers, and thus got any longer than the ten to thirty operations demonstrated in [1], then measurement at the end would have started to become entirely random, making computation impossible.

What makes quantum computation scalable beyond this limit is error correction, where redundant qubits are added, so that if the state of only one qubit changes then it can be reset to the state of the others. The theory of error correction is very sophisticated but it is enough for now to know only that it exists, taking as the only example a partial version of Shor's coding scheme[2] [12], if we represent the logical state $|0\rangle$ with a physical state $|0\rangle\,|0\rangle\,|0\rangle$ in three qubits, and a 'bit flip' error occurs, turning this into $|0\rangle\,|1\rangle\,|0\rangle$, then we can write a quantum algorithm that detects and corrects this error. Algorithms capable of detecting and correcting single errors are of a smaller scale to the algorithms demonstrated in [1], so by alternating between calculation and error detection one can reasonably expect to perform larger calculations without any single bit-flip errors propagating through the calculation. This allows computation to be performed at a greater scale than naive implementation of algorithms would allow.

When using these redundant coding schemes, a distinction is made between operations that are called fault tolerant, versus those that are not. Taking the example of a three qubit coding scheme capable of detecting bit flip errors, whose physical state is in $|0\rangle\,|0\rangle\,|0\rangle$, before undergoing an error that trans-

[2]Shor's full scheme nests this process inside another process for detecting 'phase flip' errors as well.

forms the physical state into the invalid state $|0\rangle|1\rangle|0\rangle$. Our intention would normally be to detect and correct this error, but what if this error occurs right before a computation is performed? If we want to map the logical state $|0\rangle$ to $|1\rangle$, then we could imagine this erroneous physical state mapping to $|1\rangle|0\rangle|1\rangle$, which after error correction will in fact represent $|1\rangle$. If we had gotten $|1\rangle|0\rangle|0\rangle$ instead, i.e. if the error had propagated between physical qubits, then after error correction we would have mapped $|0\rangle$ to $|0\rangle$. An algorithm that always exhibits the former behaviour is called *fault tolerant*. In analogue to the elementary evolutions of individual quantum objects, we define an *elementary gate* to be a unitary transformation that can be implemented fault tolerantly on some logical states, as a sequence of elementary evolutions acting on the corresponding physical states.

Once the technology of fault tolerant computation is set up, we can effectively pretend that it doesn't exist, and draw quantum circuits that act on logical bits directly. These logical circuits can then be fleshed out automatically into a physical circuit acting on perhaps seven times as many physical objects, with error correction occupying much of what may have once been a simple algorithm. Typically algorithms are first described in a theoretical, ideal context, with no errors, possible only in classical simulations of these algorithms, and are later run as a naive physical implementation, as in [1], and then finally as a scalable, fault tolerant algorithm, supposing that there is a quantum computer with enough qubits to run it. In this way all of the algorithms and techniques described in this thesis are intuitively understood to run at the physical level, in terms of elementary evolutions, but we will still pay deliberate mind to the eventual need for fault tolerant implementation, which will force our algorithms to be implemented using a much more restricted set of elementary gates acting on logical quantum objects.

1.3.5 Phase, Bloch Sphere

Since we assume a very specific space $\mathbb{C}^N$ with a canonical/computational basis available, we can and often do talk directly about the coordinates of vectors in this space. In a composite space we can use the coordinates in the Kronecker basis, so for example the Kronecker product $|+\rangle \otimes |+\rangle$ and $|+\rangle \otimes |-\rangle$ have coordinates

$$|+\rangle \otimes |+\rangle = \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \end{bmatrix}, \qquad |+\rangle \otimes |-\rangle = \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \end{bmatrix}.$$

If we have two states $|\psi\rangle$ and $|\phi\rangle$ such as the above two states, whose pth coordinate have coordinates that have the same complex modulus, and hence are unit complex multiples of each other, then we say they differ by a *relative phase factor* $e^{i\theta}$ in this coordinate, such as in the above example of $|+\rangle \otimes |+\rangle$ and $|+\rangle \otimes |-\rangle$, where their $|0\rangle|1\rangle$ and $|1\rangle|1\rangle$ coordinates differ by a relative phase factor of -1. A more general example in a single qubit is $|\psi\rangle = |0\rangle/\sqrt{2} + e^{i\theta}|1\rangle/\sqrt{2}$ vs. $|\phi\rangle = |0\rangle/\sqrt{2} + |1\rangle/\sqrt{2}$, whose $|1\rangle$ amplitude differs by a relative phase factor $e^{i\theta}$.

On the other hand, if two states are unit complex multiples of each other overall, i.e. $|\psi\rangle = e^{i\theta}|\phi\rangle$, then we say that they differ by a *global phase factor* $e^{i\theta}$. Interestingly, global phase differences are a peculiarity of the model we are using, and are not understood to have any effect on measurement, since global phase differences cannot affect the size of any coordinate in a given quantum state, nor can it affect any resultant states after performing unitary operations. For example if we have two states $|+\rangle$ and $-|+\rangle$ and apply the Hadamard matrix H to each we would get $|0\rangle$ and $-|0\rangle$ respectively, both before and after they have identical behaviour when measured.

For comparison, relative phase differences do not affect the probability of getting a certain measurement when measured in the computational basis, but can affect the course of calculation, producing states that are very different when measured themselves. For example $|+\rangle$ and $|-\rangle$ differ only by relative phase -1 in the $|1\rangle$ amplitude, but after applying H we get $|0\rangle$ and $|1\rangle$ respectively, different computational basis vectors, which are distinct when measured. Additionally, while global phase differences are a basis independent property of two quantum states, different bases may have different relative phase factors, or simply different coordinates altogether, such as in this $|+\rangle$ and $|-\rangle$ example where if measured in the $|+\rangle/|-\rangle$ basis the two states would give distinct measurements from each other every time.

By ignoring global phase differences, we can remove a degree of freedom from the state space being discussed. In a single qubit we have the state space

$$\left\{ \begin{bmatrix} a+ib \\ c+id \end{bmatrix} \;\middle|\; a,b,c,d \in \mathbb{C}, a^2+b^2+c^2+d^2 = 1 \right\}.$$

This is essentially the unit sphere in $\mathbb{R}^4$, but by presenting these coordinates in polar form, and making global phase explicit we can present the same set as

$$\left\{ e^{i\gamma} \begin{bmatrix} \cos(\theta) \\ e^{i\phi}\sin(\theta) \end{bmatrix} \;\middle|\; \gamma,\theta,\phi \in \mathbb{R} \right\}.$$

This presentation motivates what is called the 'Bloch sphere', which ignores global phase factors in order to represent a single qubit as a sphere in $\mathbb{R}^3$, a

space much more amenable to human intuition. The coordinates are chosen by the map

$$e^{i\gamma}\begin{bmatrix}\cos(\theta)\\ e^{i\phi}\sin(\theta)\end{bmatrix}\mapsto\begin{bmatrix}\cos(\phi)\sin(2\theta)\\ \sin(\phi)\sin(2\theta)\\ \cos(2\theta)\end{bmatrix}.$$

We directly remove the global phase factor $e^{i\gamma}$, and we also double θ which removes a global phase factor of -1 associated with $\pi\leq\theta<2\pi$. This means that while in the basic $\mathbb{C}^2$ picture of a qubit, $|0\rangle$ and $|1\rangle$ are orthogonal, in the Bloch sphere they are in fact collinear;

$$\begin{bmatrix}1\\0\end{bmatrix}\mapsto\begin{bmatrix}0\\0\\1\end{bmatrix},\qquad\qquad\begin{bmatrix}0\\1\end{bmatrix}\mapsto\begin{bmatrix}0\\0\\-1\end{bmatrix}.$$

The same applies to the orthogonal states $|+\rangle$ and $|-\rangle$;

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix}\mapsto\begin{bmatrix}1\\0\\0\end{bmatrix},\qquad\qquad\frac{1}{\sqrt{2}}\begin{bmatrix}1\\-1\end{bmatrix}\mapsto\begin{bmatrix}-1\\0\\0\end{bmatrix}.$$

By this reasoning, any scalar multiple of the identity matrix will actually have no effect on the results of quantum computation, and further any scalar multiple of a matrix A will be equivalent to A itself. For example the matrix

$$-X=-\begin{bmatrix}0&1\\1&0\end{bmatrix}=\begin{bmatrix}0&-1\\-1&0\end{bmatrix}$$

would be equivalent to X itself.

When considering sets of matrices, this equivalence can significantly simplify what needs to be considered, and further in a group of unitary matrices, i.e. a subgroup $G\leq U(N)$, we can make this equivalence explicit, since the scalar multiples of the identity in G, written $G\cap\{e^{i\gamma}I\}$, will form a subgroup of G, and since scalars commute this will be normal in G as well. This lets us take the group quotient $G/(G\cap\{e^{i\gamma}I\})$. If G contained both X and $-X$ for example, then they would be recognized as equivalent by this quotient, in the formal sense that they belong to the same coset of $G\cap\{e^{i\gamma}I\}$. In order to be succinct we will simply write $G/U(1)$, where $U(1)$ is the group of '1×1' unitary matrices, i.e. the set of complex numbers of unit modulus, even though these scalars are technically different to the scalar multiples of identity in $\{e^{i\gamma}I\}\subseteq U(N)$, and not all such scalar multiples will necessarily be in G anyway.

In the case of the qubit we can go further and interpret cosets in $U(2)/U(1)$ as rotations of the three dimensional Bloch sphere, i.e. rotations in $SO(3)$. For example X corresponds to the half-turn in the y-z plane

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}.$$

This example demonstrates that by focussing on real 2×2 matrices until now, we have been hiding a whole degree of freedom in the qubit. As an example of a complex unitary matrix, consider

$$D_2 = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}.$$

Recall that the Bloch sphere was defined in terms of the two angle parameters of a qubit, the angle θ corresponding to the complex modulus of the $|0\rangle$ and $|1\rangle$ coordinates, and the angle ϕ corresponding to the relative phase on the $|1\rangle$ coordinate. D_2 increases this ϕ by $\pi/2$, meaning on the Bloch sphere it is a quarter turn in the x-y plane, giving the matrix

$$\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

This interpretation is very powerful, and makes quantum computation on single qubits much more intuitive, however, as soon as one moves to having multiple qubits, or even a single qutrit, we lose the capacity to meaningfully visualise the full state space of the system in three dimensional space.

1.4 Non-Binary Logic

In classical computation data is represented in the voltage levels of conductive materials, and logic is implemented using transistors as digital switches, whose behaviour is more responsive and more consistent to the difference between zero voltage and high voltage, than to the difference between different levels of high voltage. This means that hardware for classical computation deals almost exclusively with Boolean algebra and binary arithmetic.

In contrast to this quantum computation inherits its number system directly from the dimension of the Hilbert space of the particles being used for computation. For example the spin of an electron forms a two-dimensional space,

the net spin of a Nitrogen-14 atom forms a three-dimensional space, and the energy level of a trapped ion can form a multi-dimensional space depending on the number of energy levels allowed. This means that quantum computation has a much easier time implementing logical and numerical systems other than the binary system used in classical contexts, but despite this most of the quantum computation in theoretical and practical contexts is based on binary logic and arithmetic, since this is the simplest, and inherits a lot of theoretical results directly from classical computation.

In practice the problem of implementing higher logic systems than binary is much more tractable in the quantum case, whereas the problem of creating and maintaining a large number of entangled quantum objects over time, a problem that doesn't exist in the classical case, becomes the primary constraint limiting practical quantum computation. By using logic systems that store more information in a smaller number of particles, we might actually have an easier time developing a quantum computer capable of performing any given scale of computation. This means that the step from binary to higher logic systems has a lot of motivation in the quantum case.

While there is a lot of potential in the topic of non-binary quantum computation, and a lot of novelty already described, even this has the implicit assumption that all quantum objects need to have the same dimension. In theory this isn't necessary either, and it might be possible to have a quantum computer with a mixture of objects of different dimension. Writing algorithms for such a device could allow one to economise on the strengths and weaknesses of each individual number system, potentially requiring even less physical complexity for the same level of computational power.

Further, it is worth recalling that quantum algorithms are often written for logical quantum objects that are actually encoded in larger collections of physical objects, and that these objects do not need to have the same dimension; it may turn out that fault tolerant implementations of non-binary operations can be implemented directly using qubit evolutions, or vice versa, providing significant motivation for understanding the general space of non-binary and mixed quantum computation at both a physical and logical level, since either could interface directly with the existing body of qubit theory and infrastructure.

1.4.1 Pauli and Clifford Matrices

A useful family of matrices in non-binary computation are the Pauli matrices, which generalize two familiar matrices acting on a qubit,

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \qquad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

These are generalized to $d \times d$ matrices by mapping the computational basis according to

$$X_d \left|j\right\rangle = \left|j+1 \mod d\right\rangle, \qquad Z_d \left|j\right\rangle = \exp\left(\frac{2\pi i j}{d}\right)\left|j\right\rangle,$$

where i is the complex square root of -1, and j is an index of basis vectors. For example when $d = 3$ we get

$$X_3 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \qquad Z_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -\frac{1}{2} + \frac{\sqrt{3}}{2}i & 0 \\ 0 & 0 & -\frac{1}{2} - \frac{\sqrt{3}}{2}i \end{bmatrix}.$$

The algebraic behaviour of these matrices becomes much simpler to see by writing $\omega_d = \exp(2\pi i/d)$, so that Z_d maps $\left|j\right\rangle$ to $\omega_d^j \left|j\right\rangle$. Since $\omega_d^d = 1$, we can see easily that $Z_d^d = I$. Clearly $X_d^d = I$ as well, forming a simple cyclic group of permutation matrices, which in this sense allows us to visualize the computational basis vectors arranged in a circle, like numbers on a clock, with X_d rotating this circle. Z_d will not rotate or move this circle at all, but it will map the complex harmonics of this circle to each other cyclically, which may be a helpful image for understanding how these operations relate to each other.

Observe that these matrices do not commute, but rather that

$$\begin{aligned} Z_d X_d \left|i\right\rangle =& Z_d \left|i+1 \mod d\right\rangle \\ =& \omega_d^{i+1} \left|i+1 \mod d\right\rangle \\ =& \omega_d^{i+1} X_d \left|i\right\rangle \\ =& \omega_d X_d Z_d \left|i\right\rangle. \end{aligned}$$

From this it can be seen that the set of matrices generated by X_d and Z_d, will be a set of order d^3, with elements of the form $\omega_d^i X_d^j Z_d^k$. We call this set the *Pauli group*.

This group is an interesting finite group in quantum computation, first brought to attention due to the fact X_2 and Z_2 along with $Y = iX_2Z_2$ have physical meaning in quantum mechanical description of electron spin. These three matrices each transform the Bloch sphere by a half turn in a different pair of axes, making them a powerful starting point for reasoning about $SO(3)$ as well. Similarly, the Pauli groups of non-binary systems provide a lot of algebraic power for understanding the geometry of their corresponding state spaces, being generated by two gates that are simple to understand and to implement fault tolerantly, but with different combinations of X_d and Z_d providing complete generality in the combinations of axes one rotates about.

In order to capture the Y_2 matrix as well we need to introduce additional scale factors ω_{2d}, which in the $d = 2$ case gives $\omega_4 = i$, and $Y_2 = \omega_4 X_2 Z_2$. This gives a group of twice the size called the *Weyl-Heisenberg group*, denoted

$$H(d) = \{\omega_{2d}^i X^j Z^k\}.$$

In a quantum system with one object of dimension d, we call the normaliser of the Weyl-Heisenberg group the *Clifford group*, written $\mathcal{C}$. Since $H(d)$ is generated by X, Z, and ω_{2d}, we can write

$$\mathcal{C} = N(H(d)) = \{A \mid A \in U(d), AXA^{-1}, AZA^{-1} \in H(d)\}.$$

The Clifford group contains arbitrary scale factors, making it uncountably infinite, but when removing these scale factors it turns out to be another powerful finite group, making its generator a very useful place to start when considering possible elementary gate sets for a quantum computer. In a single-qubit system the Clifford group (with scale factors removed) is commonly known to be generated by

$$D_2 = \sqrt{Z_2} = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}, \qquad H_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

and in a single-qutrit system it is generated by

$$S_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \omega_3 \end{bmatrix}, \qquad H_3 = \frac{1}{\sqrt{3}} \begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega_3 & \omega_3^2 \\ 1 & \omega_3^2 & \omega_3 \end{bmatrix}.$$

The matrices H_2 and H_3 are also called the *discrete Fourier transform* in two and three dimensions respectively. They can be generalized to arbitrary dimension by defining

$$H_d = \frac{1}{\sqrt{d}} \sum_{i,j} \omega_d^{ij} |i\rangle \langle j| .$$

When we visualise the Pauli matrices as cyclic operations on the displacements and harmonics of a circle, the discrete Fourier transform is precisely the map from displacements *to* harmonics, simultaneously making the simple geometry of X_d available to Z_d, and the simple action of Z_d available to X_d. Algebraically this can be written

$$H_d^{-1} Z_d H_d = X_d, \qquad H_d^{-1} X_d H_d = Z_d^{-1},$$

which tells us that in general $H_d \in N(H(d))$.

The Clifford group is often easy to implement fault tolerantly, and for example in the Steane code [13] both H_2 and D_2 can be implemented on a logical qubit by simply applying it to each of the seven physical qubits in the code. Being a group that is easy to implement, easy to algebraically manipulate, and quite wide reaching in its coverage of $U(d)$, the Clifford group comes up very frequently in discussions of quantum computation.

Pauli, Weyl-Heisenberg, and Clifford groups also exist in composite systems. In the Pauli case this group contains arbitrary combinations of X_{d_i} and Z_{d_i} on each object i of the system, so in a system with two objects of dimension d_1 and d_2 the Pauli group would be

$$\{\omega_{d_1}^a \omega_{d_2}^b (X_{d_1}^c Z_{d_1}^d \otimes X_{d_2}^e Z_{d_2}^f) \mid a, b, c, d, e, f \in \mathbb{Z}\}.$$

The scale factors ω_{d_i} are shared, and can be combined into a single term $\omega_{d'}$ where d' is the greatest common divisor of d_1 and d_2, but the operations $X_{d_1} Z_{d_1}$ and $X_{d_2} Z_{d_2}$ will not combine in any analogous way. Adding the scale factors ω_{2d_i} corresponding to the even-dimension objects in the system allows us to define the Weyl-Heisenberg group of the composite system, written $H(d_1, d_2, \ldots, d_n)$. The Clifford group of a composite system, again written $\mathcal{C}$, is simply the normaliser of the corresponding composite Weyl-Heisenberg group. It is interesting and powerful to note that while the Weyl-Heisenberg group only contains Kronecker products, the Clifford group can contain operations dependently on multiple objects. In a composite system with two d level objects the operations SUM and SWAP will always be Clifford. These are defined as the linear extensions of

$$SUM(|i\rangle |j\rangle) = |i\rangle |i + j \mod d\rangle, \qquad SWAP(|i\rangle |j\rangle) = |j\rangle |i\rangle.$$

Interestingly, when $d = 2$, the SUM operation is exactly the controlled increment or CNOT operation. This does not occur in higher systems, where controlled increment and SUM are two distinct operations with very different properties.

To prove that SWAP is Clifford, observe $SWAP^{-1} = SWAP$, and let $A, B \in H_d$, then

$$\begin{aligned} SWAP(A \otimes B)SWAP(|i\rangle |j\rangle) &= SWAP(A \otimes B)(|j\rangle |i\rangle) \\ &= SWAP(A |j\rangle \otimes B |i\rangle) \\ &= B |i\rangle \otimes A |j\rangle \end{aligned}$$

Extended linearly this means $SWAP(A \otimes B)SWAP$ is the same linear operation as $B \otimes A$, which is in the Weyl-Heisenberg group. For SUM one must manipulate X and Z separately, with many sums which we will implicitly take to be mod d.

$$\begin{aligned} &SUM^{-1}(X^a Z^b \otimes X^e Z^f)SUM(|i\rangle |j\rangle) \\ &= SUM^{-1}(X^a Z^b \otimes X^e Z^f)(|i\rangle |i+j\rangle) \\ &= \omega_d^{bi+fi+fj} SUM^{-1}(X^a \otimes X^e)(|i\rangle |i+j\rangle) \\ &= \omega_d^{bi+fi+fj} SUM^{-1}(|i+a\rangle |i+j+e\rangle) \\ &= \omega_d^{bi+fi+fj} |i+a\rangle |j+e-a\rangle \\ &= \omega_d^{bi+fi+fj} (X^a \otimes X^{e-a}) |i\rangle |j\rangle \\ &= (X^a Z^{b+f} \otimes X^{e-a} Z^f) |i\rangle |j\rangle . \end{aligned}$$

Extended linearly this is once again a Weyl-Heisenberg matrix, $X^a Z^{b+f} \otimes X^{e-a} Z^f$.

1.4.2 Notation for Mixed Systems

As we move from non-mixed systems to more arbitrary mixed systems, we must take care with the notation that we use. Not only must we distinguish between X_{d_1} and X_{d_2} which are defined to act on different kinds of quantum object, we must also distinguish between $I \otimes X_d$ and $X_d \otimes I$, which apply X_d to different objects with the same dimension. In order to denote operations succinctly but unambiguously, we aim to use most of our pronumerals in a consistent and predictable manner, so that subscripting an operation with a different letter can unambiguously specify a different piece of information about that operation. For example X_{d_i} is the Pauli X operation in $U(d_i)$, whereas X_i is the same Pauli operation but acting on the ith object in a composite system $U(N) = U(d_1 d_2 \dots d_n)$.

First of all, when talking about a quantum system, we will use N or M to denote the number of computational basis states in that system, and either

n or $m+n$ to be the number of individual objects in the composite system, where the first m objects will be somehow distinct from the last n. This means the most general operation we could describe in such a system would be an operation in $U(N)$, and such an operation would be denoted U_N to indicate its dimension.

When we want to talk about an individual object within a composite system, we index each object 1 through to n, (or 1 through to $n+m$) and use letters such as i, j, and k for variables that represent such an index. We then define d_i to be the dimension of the object indexed by i. When the system is a single object we will have $n = 1$, and $N = d_1$, which means in such a system we are simultaneously using the N notation of [14] and the d notation of [10].

Now, when indexing computational basis states in a quantum system, our notation will vary depending on how specific we need to be. Up until now we have only indexed the computational basis states in single-object systems, and have used i and j to index these states, and in Section 2.4 we will do the same, but in the rest of Chapter 2 we follow [3] and use i to index the trits in the ternary expansion of an integer, so that a_i and b_i are each the ith trit of two different ternary numbers a and b. When we reason about arbitrary unitary operations in Chapter 3 we will instead use variables such as p, q, and r to index the computational basis states of an arbitrary composite system $\mathbb{C}^N$, meaning they will be integers in the range 0 through to $N-1$. Usually we will introduce p indirectly, based on the ternary expansion notation used in [3], by naming a computational basis state $|p_n\rangle \dots |p_1\rangle$, which we take to implicitly define p to be the integer corresponding to this computational basis state, where of course $|p_i\rangle$ will be a computational basis state in the individual system $\mathbb{C}^{d_i}$. Note that we have reversed the order of $p_1 \dots p_n$, so that p_n is the most significant digit, and p_1 is the least significant digit.

When using this p and p_i convention, we will use a notation similar to that of [3] to denote the special class of permutation matrices that represent transpositions; we will write $S_{p,q}$ to denote the matrix that exchanges the computational basis states $|p\rangle$ and $|q\rangle$, while leaving all others unchanged;

$$S_{p,q}|r\rangle = \begin{cases} |q\rangle, & \text{if } r = p, \\ |p\rangle, & \text{if } r = q, \\ |r\rangle, & \text{otherwise.} \end{cases}$$

Additionally, we can apply transpositions to a single quantum object, which will be written S_{p_i,q_i} instead; this gives an operation acting on the much smaller space $\mathbb{C}^{d_i}$. For example in the smallest possible mixed quantum system, with one qubit and one qutrit, $n = 2$, $d_2 = 2$, $d_1 = 3$, $N = d_1 d_2 = 6$, the $S_{p,q}$

notation would describe matrices such as

$$S_{00,01} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

whereas the S_{p_i,q_i} notation would describe matrices such as

$$S_{0,1} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

We can use the Kronecker product to map this matrix into one acting on the composite system, giving

$$I \otimes S_{0,1} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

which we can see swaps two pairs of states instead of just one. In this system $S_{0,1} = S_{00,01}S_{10,11}$.

Next, if we have a unitary matrix $G_d \in U(d)$, and some object i with dimension $d_i = d$, then we can define G_i to be this operation G_d applied to the ith object, i.e.

$$G_i = I_{d_n} \otimes \cdots \otimes I_{d_{i-1}} \otimes G_d \otimes I_{d_{i+1}} \otimes \cdots \otimes I_{d_n}$$

When indexing an operation other than $S_{p,q}$ with a bare numeral, such as X_2 or Z_3, we exclusively interpret this as X_d with $d = 2$, or Z_d with $d = 3$ respectively. We will not use X_2 to mean $X_{d_2} \otimes I$, for example. If we talk about an operation such as $X_2 \in U(N)$ or $X_d \in U(N)$, in a composite system, we mean X_i, where i is an *arbitrary* index satisfying $d_i = d$. For example we could suppose that a quantum computer has X_2 available as an elementary gate, and what we would mean is that the system has X_i available as an elementary gate for every i satisfying $d_i = 2$.

Take as an example the mixed quantum system from before, setting $i = 2$, we would have

$$X_i = X_2 \otimes I_3 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

On the other hand if $i = 3$ then

$$X_i = I_2 \otimes X_3 = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

Here $d_i = 2$ and $d_i = 3$ are unique, so in fact the notation we have described already identifies X_2 with $X_2 \otimes I_3$, and X_3 with $I_2 \otimes X_3$. We have six computational basis states:

$$\begin{aligned} p = 0 &\implies |p_2\rangle |p_1\rangle = |0\rangle |0\rangle, \\ p = 1 &\implies |p_2\rangle |p_1\rangle = |0\rangle |1\rangle, \\ p = 2 &\implies |p_2\rangle |p_1\rangle = |0\rangle |2\rangle, \\ p = 3 &\implies |p_2\rangle |p_1\rangle = |1\rangle |0\rangle, \\ p = 4 &\implies |p_2\rangle |p_1\rangle = |1\rangle |1\rangle, \\ p = 5 &\implies |p_2\rangle |p_1\rangle = |1\rangle |2\rangle. \end{aligned}$$

1.4.3 Controlled Operations

In Section 1.2.5 we saw the controlled increment operation, which we shall now generalize to controlled versions of arbitrary operations in arbitrary mixed systems. If $U_j \in U(d_j)$ is an operation acting on object j of a composite system, then for each $i \neq j$ we would like to define $C_{r_i=q_i}(U_j)$ to be the linear extension of

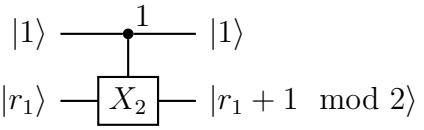
$$|r_n\rangle \ldots |r_1\rangle \mapsto \begin{cases} |r_n\rangle \ldots |r_{j+1}\rangle \left(U_j |r_j\rangle\right) |r_{j-1}\rangle \ldots |r_1\rangle, & \text{if } r_i = q_i, \\ |r_n\rangle \ldots |r_1\rangle, & \text{otherwise.} \end{cases}$$

We have already seen the CNOT operation, which can now be written

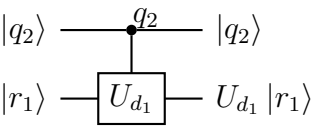
$$C_{r_2=1}(X_2) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

In an operation $C_{r_i=q_i}(U_j)$ we call object i the *control object*, or the *control digit*, and object j the *target object*, or the *target digit*. In systems where the base is known these can also be called the control qubit and target qubit respectively, or control qutrit and target qutrit. We also call the value q_i the *control value*. Just as we can say X_2 to mean X_i for an arbitrary qubit i, we can say $C_d(U_j)$ to represent $C_{r_i=d_i-1}(U_j)$ an arbitrary control object i satisfying $d = d_i$, with $q_i = d_i - 1$ chosen as a conventional default control value.

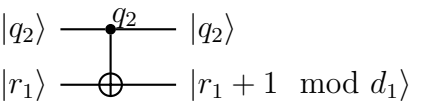
Control operations can be quite complicated to write in this form, but become much more simple in circuit form.



This example naturally generalizes to an arbitrary $C_{r_2=q_2}(U_{d_1})$ operation, i.e. an arbitrary controlled U_{d_1} operation with control object $i = 2$ and target object $j = 1$.

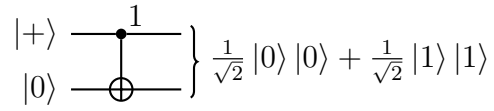


The controlled increment or controlled X operation is so common that we give it a special notation; rather than write the gate X_{d_1} in a box we can simply write



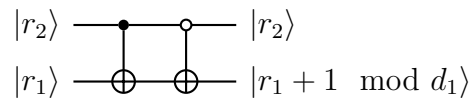
In all of the above examples if $|r_2\rangle$ were not $|q_2\rangle$ then we would have simply written $|r_1\rangle$ and $|r_2\rangle$ unchanged as the outputs on the right hand side of the circuit. One should remember that although we have been demonstrating these controlled operations on computational basis states, like any other unitary operation they will distribute linearly on computational basis states, for example consider the CNOT gate acting on $|+\rangle|0\rangle$ to create an entangled

state:

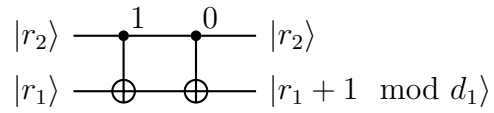


This works by expanding the Kronecker $|+\rangle|0\rangle$ to $\frac{1}{\sqrt{2}}|0\rangle|0\rangle + \frac{1}{\sqrt{2}}|1\rangle|0\rangle$, and then applying $C(X)$ to each term. It cannot be overstated how important this linear distribution is for quantum computation to be useful, especially when dealing with simple permutation matrices such as $C(X)$.

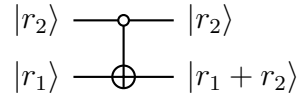
In the broader literature, when dealing with qubit computers, it has been conventional to omit the control value from circuit diagrams, and indicate the control value with either a filled in or hollow dot, for control values 1 and 0 respectively.



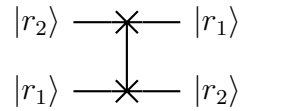
We cannot use this convention since we have more than two computational basis states to represent, so even when dealing with qubits we notate control values explicitly.



This makes the open dot notation available for reuse, so we follow [3] in using this to represent the Clifford SUM operation:

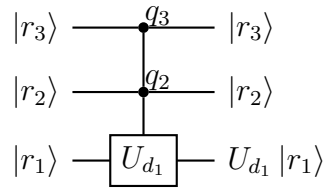


The Clifford SWAP operation that we have also discussed has its own representation in qubit contexts, as a pair of crosses linked with a vertical line, we can easily adopt this for other systems as well, as long as the two digits are the same size of course.



1.4.4 Multi-control Operations

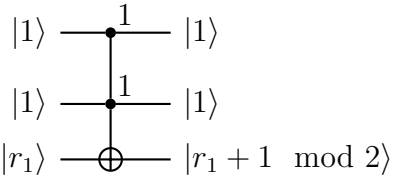
One can generalize control operations to control an operation that acts on multiple target bits, and hence define operations of the form $C_{r_3=q_3}(C_{r_2=q_2}(U_{d_1}))$, which is similar to how this gate is notated in [3]; this notation is especially convenient because of its symmetry with how they notate the gate mapping $|i\rangle\,|j\rangle\,|k\rangle$ to $|i\rangle\,|j\rangle\,|ij+k\rangle$, but since we do not use any gates of this form, we opt instead to define $C_{r_3=q_3,r_2=q_2}(U_{d_1})$ as directly being a gate with multiple control values, applying U_{d_1} when *all* conditions are satisfied, and doing nothing otherwise. The circuit representation of such a gate is, unsurprisingly:



For the sake of formality we will often work with an index set $I \subset \{1 \dots n\}$ containing the indices of all of the control objects, and write the control operation more succinctly as $C_c(U_j)$, where c is the set of computational basis vectors that $C_c(U_j)$ will apply U_j to, i.e. the set

$$\{|r_n\rangle \dots |r_1\rangle \;\mid r_i = q_i \forall i \in I\}$$

An operation that is commonly used in qubit contexts is the Toffoli gate, which is simply a controlled X_2 operation with two control qubits, $C_{r_{i_1}=1,r_{i_2}=1}(X_2)$.



In the above examples the target object has always been $i = 1$. This is not necessary, but when it is possible it makes for a much simpler block representation of the resulting $C_c(U_{d_1}) \in U(N)$. If *every* object except for $i = 1$ is a

control object, then this will have block matrix

$$\begin{bmatrix} I & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & I & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & I & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & U_j & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & I & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & I \end{bmatrix},$$

where the point along the diagonal at which U_j appears is determined by interpreting the control values as digits of an integer $0 \leq q_n \ldots q_2 \leq N/d_1$.

Chapter 2

Ternary Arithmetic

The QArC Group at Microsoft have written a number of papers describing logic in ternary quantum computers, including a full description of integer addition and Shor's factoring algorithm in ternary quantum computers [3, 5], paying particular mind to a fault tolerant/elementary gate set called the metaplectic basis, described in [8, 4]. While the algorithms for addition are deliberately designed to be generic to any ternary quantum computer, the specific analysis of elementary gates required is specific to the class of quantum computer being considered, so it is unclear whether these analyses pose any relevance to our discussion of mixed logic. We shall describe the algorithms given for addition, and demonstrate how they can be ported to a mixed qubit-qutrit quantum computer without any modification. This will inform our exploration of theoretical quantum gates that might be powerful to have available in such mixed contexts. We also discuss algebraic techniques used in [3] that fail to generalize to mixed contexts, suggesting some novel complexity present in mixed contexts that does not exist in a purely binary or purely ternary context.

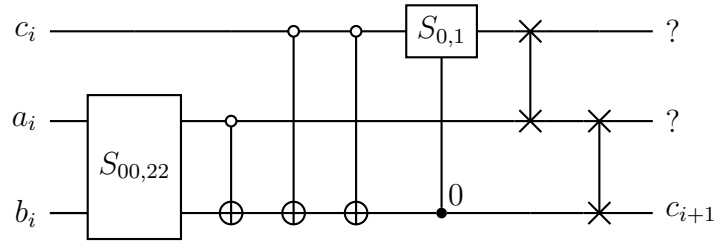
2.1 Ternary Addition With Minimal Width

The first algorithm laid out in [3] was a reversible algorithm for adding the ternary expansion of two integers, a simple generalization of an operation fundamental to computation in integers. In order to implement addition in some base d one must implement the sum of three digits $a_i + b_i + c_i$ to get a number between 0 and $3d - 3$, a number which is itself represented as two

separate digits,

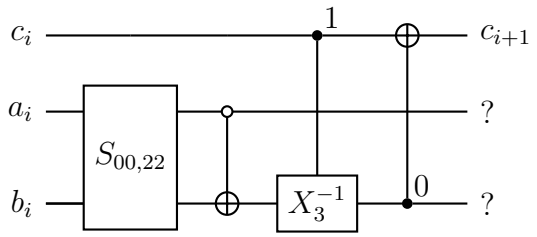
$$c_{i+1} = \left\lfloor \frac{a_i + b_i + c_i}{d} \right\rfloor, \text{ and}$$
$$s_i = a_i + b_i + c_i \mod d.$$

s_i is easily implemented as a pair of SUM operations, which is significantly simpler than c_{i+1} which is an almost arbitrary map $\{0, \ldots, d-1\}^3 \to \{0, \ldots, d-1\}$. A significant simplification made in [3] is to note that when $d = 3$, the only way c_{i+1} can be 2 is if all a_i, b_i, and c_i before it are also 2. Since c_0 is assumed to be 0 this means c_i is always either 0 or 1, meaning we only need to implement a map from $\{0, 1, 2\}^2 \times \{0, 1\} \to \{0, 1\}$. Then with careful analysis of the required map they present a circuit that is sufficient for the computation needed.

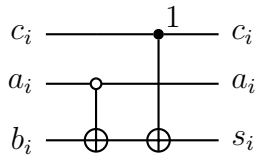


This carry circuit is then used to add two m-qutrit integers by composing it m times, calculating c_m, which is written into a separate output register s_m. Then since these carry operations are reversible, they are reversed one by one, overwriting b_i with each s_i in the process. Note that c_{i+1} is swapped with the b_i register once it is calculated, making presentation cleaner later on, and also simulating the fact that qutrits in a physical quantum computer may not be able to interact with each other arbitrarily, they might only be able to interact with adjacent qutrits in some physical layout. They do all of this while only requiring two SWAP operations, which considered significantly cheaper in fault tolerant contexts since they are Clifford operations, unlike the $S_{00,22}$ or $C(S_{0,1})$ operations.

We note that the core of this improvement was to treat c_i like a qubit! This suggests that the whole algorithm for ternary arithmetic will map very cleanly into a mixed quantum computer. If we modify the carry circuit so that c_i actually is a qubit, we get a similar looking circuit.



The pair of Clifford SUM operations become a single non-Clifford[1] controlled decrement, the inverse of a controlled increment. Further the controlled transposition simply becomes a controlled increment mod 2. Finally we cannot swap c_{i+1} with b_i since one is a qubit and the other is a qutrit. We see very clearly that this algorithm would be very sensitive to connectivity between qubits and qutrits if implemented in a mixed system. We have turned a carry operation with two non-Clifford operations into one with three non-Clifford operations. In addition to this, the calculation $s_i = a_i + b_i + c_i \mod 3$ receives a slight modification.



This means that in total to add two m-qutrit integers we have gone from $2m$ non-Clifford operations to $4m$ non-Clifford operations, which may be a significant loss, but also isn't strictly meaningful at this point in time, since this metric is specific to the class of quantum computer described in [4], whereas we do not have a proposed fault tolerant model of mixed quantum computation with which to judge this outcome. We have used two qubit-qutrit gates, $C_2(X_3)$ and $C_3(X_2)$, gates that we will end up investigating directly in Chapter 4.

2.2 Ternary Addition With Minimal Depth

The second algorithm presented by [3] aims to parallelise the calculation of the carry trits by observing three cases of $a_i + b_i + c_i$:

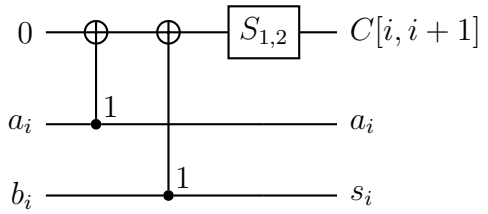
[1] One can compute $C(X)X^jZ^kC(X^{-1})$ as we did in Section 1.4.1, but we will see in Section 4.1 that *no* operation that acts dependently on a qubit-qutrit pair can be Clifford.

- $a_i = b_i = 0 \implies c_{i+1} = 0$, denoted $C[i, i+1] = 0$,
- $a_i + b_i = 1 \implies c_{i+1} = c_i$, denoted $C[i, i+1] = 2$,
- $a_i + b_i \geq 2 \implies c_{i+1} = 1$, denoted $C[i, i+1] = 1$.

These three cases can be calculated up front and stored in trit registers, without knowing any carry digits. This allows the calculations to be done in parallel requiring m Clifford gates, but only taking the amount of time required to compute one or two of these calculations in sequence.

Next a merge algorithm is defined, if $C[i, j] = C[j, k] = 2$ then $c_i = c_j = c_k$, so $C[i, k] = 2$ also. In fact whenever $C[j, k] = 2$ we will have $C[i, k] = C[i, j]$. On the other hand, if $C[j, k] \neq 2$ then c_k will be equal to $C[j, k]$ itself, either 0 or 1, and in either case $C[i, k]$ is simply $C[j, k]$. This calculation of $C[i, k]$ in terms of $C[i, j]$ and $C[j, k]$ is another classical operation, which can be implemented as a permutation. Now different combinations of $C[i, k]$ are merged in $\lfloor \log l \rfloor + \left\lfloor \log \frac{n}{3} \right\rceil + 2$ parallel 'layers', requiring $3l - 2\omega(l) - 2\lfloor \log l \rfloor - 1$ non-Clifford operations across all of these layers, where $\omega(l)$ is the number of 1s in the binary expansion of l. The algorithm far from minimizes width however, taking $l - \omega(l) - \lfloor \log l \rfloor$ additional qutrits on top of the $2l$ needed for the input of the algorithm.

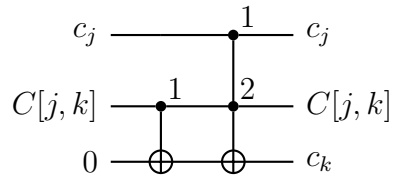
Here a key feature of the algorithm is to observe that $C[i, j]$ is intrinsically ternary, but interestingly the addition of two binary integers would have the same three carry cases, so if we can implement the base case $C[i, i+1]$ as yielding a qutrit based on two qubits, then we would have turned another ternary arithmetic algorithm into a mixed algorithm, but this time using qutrit carry information to add a series of qubits! Let $a_i, b_i \in \{0, 1\}$, then c_{i+1} will be 1 if $a_i = b_i = 1$, 0 if $a_i = b_i = 0$, and c_i if $a_i \neq b_i$. This can be implemented with a simple circuit inspired by the same calculation in the pure qutrit case in [3].



Here $S_{1,2}$ is a potentially cheap Clifford operation that allows us to calculate $C[i, i+1]$ in terms of $a_i + b_i$. This is much like the $S_{0,1}$ used in the original $C[i, i+1]$ circuit in [3]. Much like the SWAP operations in their previous carry gates, Clifford operations are being used liberally to turn quantities that are

easy to calculate into quantities that are more convenient to design algorithms for. The next step of the algorithm is to merge the different qutrit quantities $C[i,k]$, which is unchanged since there are no qubits involved.

Now in order to calculate the resulting qubits we must obviously diverge from [3] once more. A key component of their algorithm was to calculate $C[0,1]$ separately, since $c_0 = 0$ we can replace the $C[0,1] = 2$ case with $C[0,1] = 0$, which is another binary quantity. $C[0,1]$ is 1 if both a_0 and b_0 are 1, which is implemented with a single Toffoli gate $C_{a0=1,b0=1}(X_{C[0,1]})$. Now since we have eliminated the $C[0,1] = 2$ case, what we are really calculating is c_1. Next [3] would have used the merging formula to calculate each c_k implicitly by merging $C[0,j] = c_j$ with $C[j,k]$. Since we want c_k to be a qubit we will replace these specific merges with our own circuit.



Now with all c_k calculated we can of course calculate $s_k = a_i + b_i + c_i \mod 2$ with two SUM operations, i.e. two $C(X_2)$ operations. The circuits for calculating c_k used gates with two control objects, so the performance of this part of the algorithm would depend primarily on how these operations are implemented in the given mixed quantum computer. We have now generalized both algorithms from being purely qutrit based to using a mixture of qubits and qutrits, and in doing so have repeatedly encountered the controlled increment operations $C_2(X_3)$ and $C_3(X_2)$.

2.3 Speculation on Mixed Coding Schemes

At this point we make the small aside that while it makes sense to use physical qubits to encode logical qubits, the primary advantage of doing this is that it allows one to avoid thinking about qutrits altogether. As soon as we start considering a mixture of qubits and qutrits we could develop a coding scheme that encodes these objects into a system that is itself mixed, but we should plausibly be able to encode them into a system that is itself a pure ternary system as well, or even a pure binary system. For example both of the algorithms given in [3] used many binary registers as is, by simply not using the $|2\rangle$ state of the relevant qutrits. Taking the simplified $\{|0\rangle|0\rangle|0\rangle, |1\rangle|1\rangle|1\rangle\}$ example from Section 1.3.4, and generalising it to $\{|0\rangle|0\rangle|0\rangle, |1\rangle|1\rangle|1\rangle, |2\rangle|2\rangle|2\rangle\}$ in

order to implement a logical qutrit, we could then implement a logical qubit on the same computer by simply treating all $|2\rangle$ states in that block as errors that need to be corrected.

In actual fact coding schemes can do quite a bit better than detecting bit flip errors with $3\times$ redundancy, but simultaneously detect bit flip and sign flip errors with $7\times$ redundancy with an encoding scheme called the Steane code [13], which raises the question of what a qubit-in-qutrit Steane code might look like. Further, if a logical qubit and a logical qutrit are both implemented in such a manner on the same ternary system, then another natural question would be what a fault tolerant implementation of $C_2(X_3)$ and $C_3(X_2)$ would look like. All of these arguments apply equally to encoding qutrits in a qubit system, which might be the more compelling option, since the major quantum computers to date have been based on physical qubits.

A standard milestone for any proposed quantum computer is to show that a set of fault tolerant/elementary gates can be used to approximate any unitary operation $U \in U(N)$. In Chapter 3 we will generalize the approach laid out in [6] to show that two different sets of seven gates can be used to make such an approximation. Both of these sets contain $C_3(X_2)$, $C_2(X_3)$, and one of them also contains $C_2(X_3)$, providing further theoretical motivation for fault tolerant implementation of these gates, in addition to their use in out generalisation of the algorithms in [3].

2.4 Supermetaplectic Basis

In the above discussion and related papers the QArC group have paid particular attention to a "Metaplectic" system of computation which is fault tolerant, and implements an elementary gate set called the "metaplectic basis" that can generate all Clifford operations, along with the more expensive implementation of a single non-Clifford operation

$$Y = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}.$$

In [3] an alternative fault tolerant gate set was proposed, using P_9 in place of Y, the simplest gate satisfying $P_9^3 = Z_3$, in matrix form

$$P_9 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \omega_9 & 0 \\ 0 & 0 & \omega_9^2 \end{bmatrix}.$$

Both gate sets can be used to approximate any unitary $U \in U(3^n)$, but [3] showed that this latter gate set can exactly implement any permutation matrix. They refer to this latter gate set as the "supermetaplectic basis", which seems to refer to this increase in power when dealing with permutation matrices.

In order to show that this generator set exactly implements the permutation matrices in $U(3^n)$, i.e. all of $\mathcal{S}_N$, they decompose these matrices in three steps:

1. It seems to have been taken implicitly that with an implementation of $C_{i_1=2}(X)$, other permutations acting on more than two qutrits can be decomposed in a manner similar to the qubit process described previously.

2. Various permutations acting on two or three qutrits that are not in $\mathcal{C}_2$ were shown to be pairwise equivalent, including arbitrary transpositions of pairs of states.

3. Two different diagonal matrices were constructed using P_9, whose Fourier transforms were qutrit permutations including $C_{i_1=2}(X)$ itself, meaning the permutations themselves are in this basis.

Both of these explicit steps were derived through analysis of the permutation and diagonal matrices as polynomial expressions, first interpreting permutation matrices as acting on the computational basis, e.g.

$$C_{i_1=0}(X)(|i\rangle\,|j\rangle) = |i\rangle\,\left|j+1-i^2 \mod 3\right\rangle.$$

then interpreting diagonal matrices as acting on the phase factors of the computational basis, e.g.

$$C_{i_1=0}(Z)(|i\rangle\,|j\rangle) = \omega_3^{j(1-i^2)}\,|i\rangle\,|j\rangle.$$

This allows us to compose permutation matrices and diagonal matrices respectively, and simplify the resulting polynomials mod 3. This technique is clearly powerful, given its success in proving the above claims in [3], but unfortunately when we try to do this in a mixed system, we get terms evaluated mod 3, nested inside terms evaluated mod 2, and vice versa. For example in a system with a qubit and a qutrit we could write

$$X_3\,|i\rangle\,|j\rangle = |i\rangle\,|j+1 \mod 3\rangle.$$

Further, by using a control value of 1 we can leverage that $2 \mod 2 = 0$ and write

$$C_{j=1}(X_2)\,|i\rangle\,|j\rangle = |i+j \mod 2\rangle\,|j\rangle.$$

Composing these we get

$$C_{j=1}(X_2)X_3 \left|i\right\rangle \left|j\right\rangle = \left|i + (j+1 \mod 3) \mod 2\right\rangle \left|j+1 \mod 3\right\rangle ,$$

which does not simplify, defeating the purpose of this analysis.

Another approach might be to inject our bits and trits into $\mathbb{Z}_6$, which may work with the correct construction, but if we respect the structure used in [3] and represent X_d as mapping $i \mapsto i + 1 \mod 6$, then we need to identify i with $i + d$ in order for X_d^d to be equivalent to I. At this point the problem of incompatibility between qubits and qutrits immediately reappears, since a trit will identify 0 with 3 for example, which can not be added to a qubit in a well defined way. If we represent X_3 as mapping $i \mapsto i + 2 \mod 6$ to achieve $X_3^3 = I$ directly, then we still have to identify the 0 trit with some odd integer, so the same conflict occurs. As a result there is no obvious way to represent operations acting on qubits and qutrits directly as polynomials, which suggests that at an intuitive level mixed systems have the potential for novel structural behaviour not present in non-mixed systems. Whether this novel structure exists, and whether it benefits or hinders practical computation remains to be seen.

Chapter 3

Universal Computation

A foundational result in quantum computation is that of universal computation, that certain combinations of quantum gate can be used to implement any quantum algorithm to some accuracy, given sufficient circuit depth. The resulting circuits are generally too long to use in practice, compared to compilation techniques that rely on specific properties of the algorithm in question, but the result is still useful since it proves that it's not impossible, i.e. its necessary and sufficient conditions provide a starting point for designing and using quantum computers in practice.

3.1 Universal Computation in Qubit Contexts

The two most widely useful results about universality of qubit computers are the result of [2], that a quantum computer with arbitrary operations from $U(2)$ on each individual qubit, and the controlled not $C(X)$ operation, one can exactly implement any unitary $U \in U(N)$, and the result of [6], which ports this result to fault tolerant computation by showing that with only two fault tolerant elementary gates one can approximate any single-qubit operation in $U(2)$, and hence with the addition of $C(X)$, which is also fault tolerant, one can fault tolerantly approximate any operation in $U(N)$. We shall describe the former of these results, and in doing so generalize it to the following:

Theorem 3.1. *In any mixed quantum computer with at least one qubit, one can achieve universal computation with either:*

- *Arbitrary qubit operations and arbitrary controlled increments* $C_{r_i=q_i}(X_{d_j})$

- *Arbitrary qubit operations and arbitrary controlled transpositions* $C_{r_i=q_i}(S_{p_i,p'_i})$

In the case of a computer with only qubits, this theorem is equivalent to the result of [2]. We shall now outline the series of techniques presented in the textbook [11], which collect the relevant techniques from [2] and its predecessors into a continuous sequence of increasingly powerful proofs of universal computation. We will treat this as a single proof, with each step of the proof decomposing an arbitrary unitary $U \in U(N)$ into a smaller set of basic operations. The first decomposition is of U into

$$U = \prod_{p=0}^{N-2} \prod_{q=p+1}^{N-1} U_{p,q},$$

giving $(N-1)(N-2)/2$ unitary matrices $U_{p,q}$, one for each distinct pair $p = p_n \dots p_1$, $q = q_n \dots q_1$, $p < q$. Specifically $U_{p,q}$ will be 'two level' unitaries, in that they only act on the two computational basis vectors $|p\rangle$ and $|q\rangle$, meaning there are some complex a, b, c, d so that

$$U_{p,q} = I + (a-1)\,|p\rangle\langle p| + b\,|p\rangle\langle q| + c\,|q\rangle\langle p| + (c-1)\,|q\rangle\langle q|\,.$$

For example in a system of two qubits, with $p = 1$ and $q = 2$, we have

$$U_{1,2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & a & b & 0 \\ 0 & c & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

This operation will have no effect on computational basis states $|0\rangle|0\rangle$ or $|1\rangle|1\rangle$, but will act on $|p_2\rangle|p_1\rangle$ and $|q_2\rangle|q_1\rangle$ in a similar manner to

$$U_2 = \begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

The proof that any unitary can be represented as a product of such two-level unitaries amounts to a kind of row reduction on the lower left triangle of the unitary, choosing the c value of each $U_{p,q}$ in order to eliminate each element one at a time. We won't present the exact details here, since this part requires no change in the case of a mixed quantum computer. The full procedure can of course be found in [11].

These two-level unitaries can then be implemented as a series of controlled operations $\{C_c(U_2)\}$, which can in turn be decomposed into 'basic' operations,

$C(X)$ along with arbitrary $U_2 \in U(2)$. This decomposition is done using a variety of techniques presented in [2]. This set of techniques provide an excellent starting point for reasoning about quantum computation at the level of physical qubits, but in order to work with logical qubits one can go a step further and approximately implement all of $U(2)$ using only two basic gates with known fault tolerant implementations. This result was shown in [6], and shall inform our later discussion in Section 3.4 of minimal gate sets in quantum computers consisting only of qubits and qutrits.

3.2 Representing Two-Level Unitaries With Control Operations

Once we have decomposed a unitary into two-level unitaries $U_{p,q}$, acting on computational basis states $|p\rangle$ and $|q\rangle$, our next goal will be to represent this two-level unitary as a concrete quantum circuit consisting of various controlled operations. First, we must choose any qubit in the system, which will be indexed by an integer j satisfying $d_j = 2$. Now define $C_c(U_j)$ to be the control operation applying

$$U_2 = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

to qubit j, so long as every other object in the quantum system is in state q_i. In our notation this means

$$c = \{|r_n\rangle \dots |r_1\rangle \mid r_i = q_i \text{ whenever } i \neq j\}.$$

Now $C_c(U_j)$ will also be a two-level unitary, acting on

$$p' = q_n \dots q_{j+1} 0 q_{j-1} \dots q_1,$$
$$q' = q_n \dots q_{j+1} 1 q_{j-1} \dots q_1.$$

Formally,

$$C_c(U_j) = I + (a-1)\,|p'\rangle\langle p'| + b\,|p'\rangle\langle q'| + c\,|q'\rangle\langle p'| + (c-1)\,|q'\rangle\langle q'|\,.$$

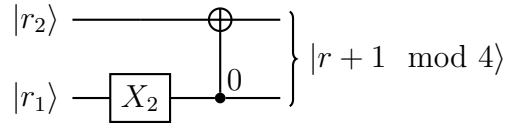
Consider our $U_{1,2}$ example. We could choose $j = 1$. Then $p'_2 = q'_2 = q_2 = 1$, so p' and q' would have binary expansions 10 and 11, the integers 2 and 3 respectively. This gives

$$C_c(U_j) = C_{r_2=1}(U_j) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{bmatrix}.$$

At this point we can fairly easily see how to implement $U_{1,2}$ as a concrete quantum circuit, so long as we can map $|p\rangle = |1\rangle$ to $|p'\rangle = |2\rangle$ and $|q\rangle = |2\rangle$ to $|q'\rangle = |3\rangle$. There are two permutation matrices that will do this:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

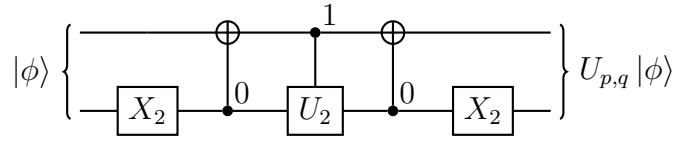
We choose the latter of these, which happens to map any $|r\rangle$ to $|r+1 \mod 4\rangle$, and can be represented by the following circuit:



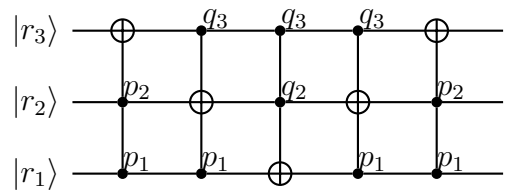
As a matrix expression this is $P = (I \otimes X_2)C_{r_1=0}(X_2 \otimes I)$. Since each term in this circuit is self-inverse, we can reverse the circuit to implement P^{-1} as well. Then in order to apply $U_{p,q}$ to $|r\rangle$, we first apply P, then $C_c(U_j)$, then P^{-1}, which is the following similarity transformation:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & a & b & 0 \\ 0 & c & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

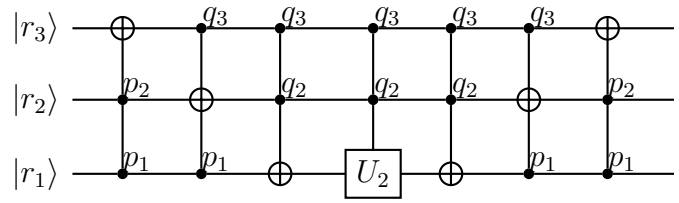
Reading from right to left we can get a concrete quantum circuit for this operation.



To generalize this we must describe a process for generating and implementing P in any qubit computer. The process given in [11] is to implement the transposition $S_{p,p'}$ mapping $|p\rangle$ to $|p'\rangle$. For them this will do as a permutation P since they assume $q'_j = q_j$, as opposed to our assumption that $q'_j = 1$. Consider as an extreme example among cases $n = 3$, and $j = 1$, where all three bits need to be inverted, i.e. $p_3 \neq q_3$, $p_2 \neq q_2$, $p_1 = q_1$.



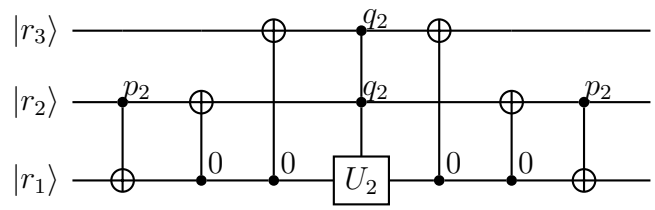
In the circuit for $PC_c(U_j)P^{-1}$ the latter half of the above sequence of operations would have no effect, since it would be transforming the action of $C_c(U_2)$ on basis vectors that it doesn't do anything to, so in fact the full circuit simplifies to a circuit that looks much like the above.



The first half of this is essentially the P we originally wanted, mapping $|p\rangle$ to $|p'\rangle$ and $|q\rangle$ to $|q'\rangle$. This process is fine for showing universal computation in the abstract, but is hard to generalize to mixed contexts, and is generally very wasteful. Instead we shall do something much simpler, using only $C(X)$. The first step shall be to choose a smaller permutation P_1 so that p_j maps to 0 and q_j maps to 1. There are four cases to consider:

- $p_j = 0$, $q_j = 1$ already,
- $p_j = 1$, $q_j = 0$,
- $p_j = q_j = 0$,
- $p_j = q_j = 1$.

In the first case we can set $P_1 = I$, and do nothing, and in the second case we can set $P = X_j$, but in the last two cases we must pick some k so that $p_k \neq q_k$. Then if we are in the third case we set $P = C_{r_k=q_k}(X_j)$, and in the fourth case $P = C_{r_k=p_k}(X_j)$. Now $P_1 |q\rangle = |q'\rangle$, so all that remains is to map each remaining p_i to q_i without changing $|q\rangle$. This is simple to do with $C_{r_j=0}(X_i)$, repeated once for each $i \neq j$ with $p_i \neq q_i$. Applied to $|p\rangle$ this will change p_j to 0, setting the control value, so that the remaining bits can be set to $|q_i\rangle$, and of course applied to $|q\rangle$ this will change q_j to 1, making no other changes since the control bit has been set incorrectly. We can apply this process to our extremal three-qubit example to get a circuit that is relatively straight forward.



In this case our circuit is the same length, or even longer if more optimization had been applied, but the simplification from three Toffoli gates to three $C(X)$ gates is dramatic once we start representing Toffoli gates in terms of $C(X)$, and as the number of bits increases this difference will increase quadratically, so it is a significant improvement. Of course arguments about universal computation are not intended to be efficient anyway, and the real purpose of this approach is to generalize to mixed systems. The way that we do this is straight forward. We still choose j to be some qubit, meaning $d_j = 2$ still. P_1 is chosen by the same process as before as well, but if k needs to be chosen it can be an arbitrary control digit whether binary or otherwise. Now to change the remaining digits p_i to q_i, we still use controlled operations, but can choose whether we use $C_{r_j=0}(S_{p_i,q_i})$ to change p_i, or $C_{r_j=0}(X_{d_i})^{q_i-p_i}$, depending on which basis of Theorem 3.1 we are aiming to use.

We have now written $U_{p,q}$ as a combination of control operations, and so next is to use the techniques described in [2] to decompose these into operations with only a single control object. Of course with the above technique most of our operations are already in this form, but this doesn't eliminate any potential cases, since we still have an arbitrary $C_c(U_j)$ in the middle of the circuit.

3.3 Decomposing Control Operations

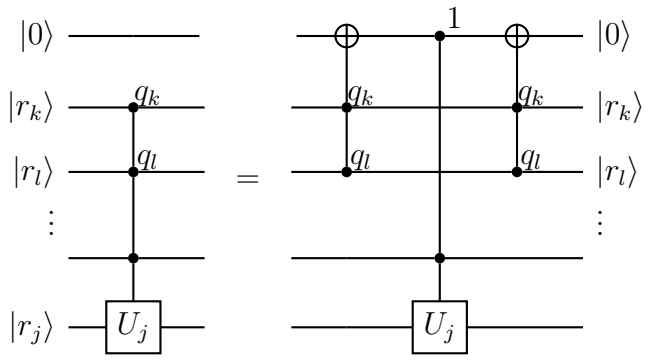
We would like to decompose $C_c(U_j)$ into elements of $U(2)$ and operations of the form $C(X_{d_i})$ or $C(S_{p_i,p'_i})$. In a binary computer this is done in three steps. The first is to introduce/require an additional $n-3$ auxiliary qubits to the quantum computer that are not intended to be affected by the original unitary in $U(N)$ or the control operation in question, and to inductively reduce the operation $C_c(U_j)$ into a series of gates with two control qubits. We index the new qubits $n+1$ through $2n-2$, which begin and end this process in the computational basis state $|0\rangle$. The key to this process is the Toffoli gate, which we recall is simply a controlled X operation with two control bits, $C_{r_k=q_k,r_l=q_l}(X_j)$.

Take I to be the set of indices i for which our control operation $C_c(U_j)$ has a

constraint $r_i = q_i$, so that

$$c = \{|r_1\rangle \dots |r_n\rangle \ \mid r_i = q_i \text{ whenever} i \in I\}$$

Then we shall perform induction on the size of I, implementing any control operation as $2\,|I|-4$ Toffoli gates, sandwiching a final single- or double-controlled operation $C_{r_k=q_k, r_l=q_l}(U_j)$. We draw this as a circuit equation, for reference.



In total this implements an operation with $|I|$ control bits in terms of one with $|I|-1$ control bits. Formally we have introduced an auxiliary qubit r_i, and reduced I to

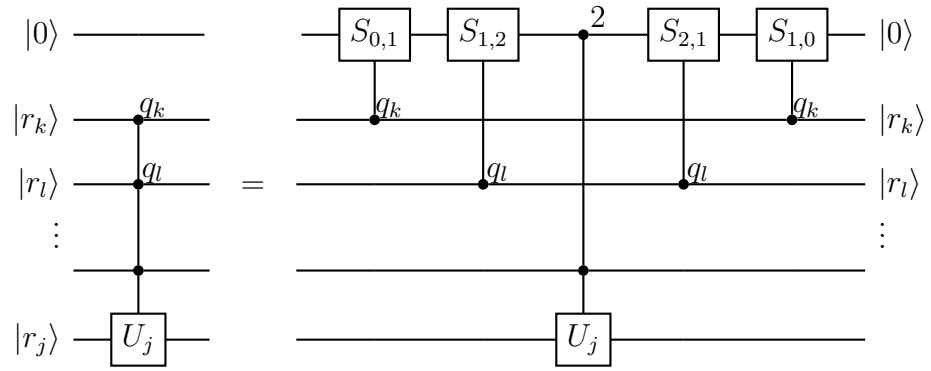
$$I' = (I \backslash \{k, l\}) \cup \{i\}$$

Correspondingly, set $q_i = 1$ and reduce the condition set c to

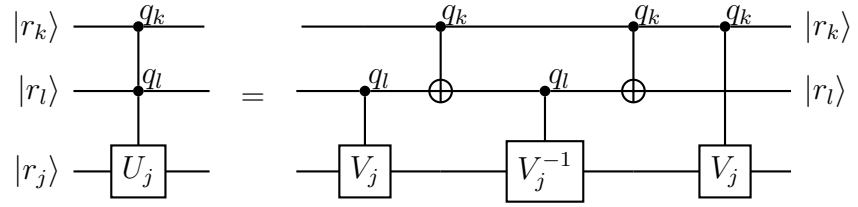
$$c' = \{|r_1\rangle \dots |r_n\rangle \ \mid r_i = q_i \text{ whenever} i \in I'\}.$$

This returns us to the form we started in, with an operation $C_{c'}(U_j)$, but with one less control qubit. Eventually we will reach a base case where $|I| = 2$, in which case $C_c(U_j)$ is already a double-controlled gate sandwiched by $2\,|i|-4 = 0$ Toffoli gates. This means that inductively we end up with the result that we wanted.

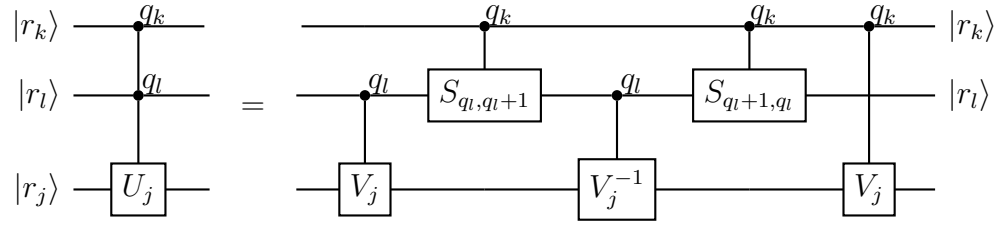
This decomposition into Toffoli gates acting on auxiliary qubits generalizes to mixed logic without any modification, so long as auxiliary qubits are available. Alternatively one can copy [10] which introduces auxiliary objects of dimension $d > 2$, and uses singly-controlled permutations gates, though controlled increments will do as well. We will assume $d = 3$ but it is easy to optimize the number of auxiliary objects when d is larger. Again we draw this transformation as a circuit equation for reference.



$S_{0,1}$ in the above diagram can be replaced with X_3 and $S_{1,0}$ with X^{-1} to achieve the same effect, depending on the desired basis. If we have arbitrarily many qutrits or higher available, then we can ignore the Toffoli gates altogether, and only use the $C(X)$ or $C(S_{p_i,p'_i})$ gates available to us to reduce $C_c(U_k)$ to a singly-controlled $C_{r_i=2}(U_j)$, but for generality we shall continue as if Toffoli gates need to be used as well. Given $V_j^2 = U_j$, [2] presents a circuit which we can use to implement $C_{r_k=q_k,r_l=q_l}(U_j)$ acting only on qubits.



This decomposes our many doubly-controlled gates into singly-controlled gates, but interestingly all of the Toffoli gates acting on auxiliary qubits will become $C(\sqrt{X_2})$[1] rather than $C(X)$. So long as object j is still a qubit, this proof will generalise to a mixed quantum computer by simply inserting the controlled transpositions available to us in place of the controlled increments.

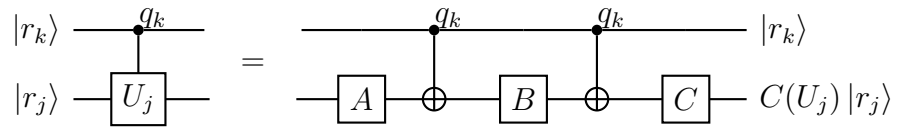


Once again S_{q_l,q_l+1} and S_{q_l+1,q_l} can be replaced with X_{d_l} and $X_{d_l}^{-1}$ depending

[1] If one wants a square root of X, then $(H^{-1}\sqrt{Z}H)^2 = H^{-1}ZH = X$ is a simple example, with $\sqrt{Z}\,|i\rangle = \omega_4^i\,|i\rangle$.

on the desired basis.

In fact [2] generalizes this implemention of doubly-controlled gates directly to arbitrary $C_c(U_j)$, so this approach could be directly generalized to a mixed context by replacing all X_2 operations with increment/decrement respectively, avoiding the discussion of auxiliary qubits or qutrits altogether, but requiring greater circuit depth in exchange for the narrower circuit width. This gives a total of three distinct ways of implementing $C_c(U_j)$ in terms of singly-controlled operations $C_{r_k=q_k}(U_j)$, acting on qubits. In all three cases we now need only demonstrate how to implement these in terms of single-qubit unitaries and $C_{r_k=q_k}(X_2)$, which in this case is the same operation as that notated by $C_{r_k=q_k}(S_{0,1})$. The qubit construction referenced in [2] works for this purpose without modification.



Here $A, B, C \in U(2)$ are chosen so that $ABC = I$ and $AXBXC = U_j$. This can be done in general, via the spherical geometry of $U(2)/U(1)$, as shown in [2]. Combining all of the steps just described we have proven Theorem 3.1, generalizing the universality result of [2] and [11] to mixed quantum computers with at least one qubit, and with significantly fewer operations than [11] seems to have used, even in the case of a computer with only qubits available. □

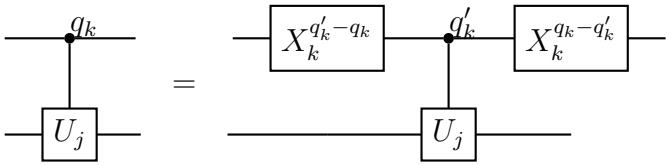
3.4 Analysing Small Permutations

We have generated $U(N)$ with a finite set of two-object operations along with arbitrary qubit operations from $U(2)$. We can then apply the result of [6], that any element of $U(2)$ can be approximated using the operations

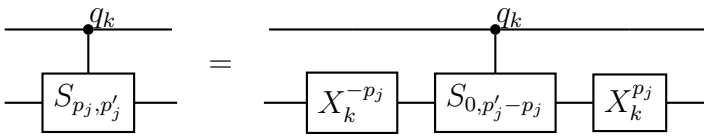
$$H_2 = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \qquad T = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{\sqrt{2}}(1+i) \end{bmatrix}.$$

We will come back in Section 4.2 to talk briefly about how this is done, since it involves an aspect of algebraic number theory that will be useful to us. The result was then used to argue that by the decompositions described in [2], the gate set H_2, T, and $C_2(X_2)$ is universal, a significant result since this gate set can also be implemented fault tolerantly. We can directly map this result to mixed quantum computers via Theorem 3.1, to get two alternative

finite gate sets, $\{H_2, T, C_{r_k=q_k}(X_j)\}$, and $\{H_2, T, C_{r_k=q_k}(S_{p_j,p'_j})\}$. We will now discuss ways that this operator set can be reduced further, with particular mind to the simplest case of quantum computers that only have qubits and qutrits. First we can fix $q_k = d_k - 1$, and generate other control values via a circuit equivalence.



This introduces each X_d to our generator set, but removes each $C_{r_k=q_k}(\dots)$ apart from $q_k = d_k - 1$. Further we already have $X_2 = H^{-1}T^4H$, so we only need X_d for $d > 2$. Additionally, we can fix p_j to 0 by a similar equivalence.



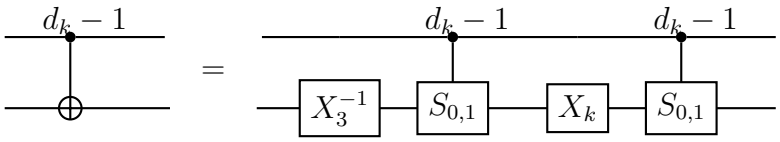
where the difference $p'_j - p_j$ is evaluated mod d_j. Finally, we can optionally exchange p_j and p'_j to sure that the difference $p'_j - p_j$ is always at most $d_j/2$. For example $S_{2,1}$ acting on a qutrit is equivalent to $S_{1,2}$, which by the above can be reduced to $X_3 S_{1,2} X_3^{-1}$. This means ultimately our generator sets are $\{H_2, T, X_{d_i}, C_{r_k=d_k-1}(X_k)\}$ and $\{H_2, T, X_{d_i}, C_{r_k=d_k-1}(S_{0,p'_k})\}$, with $d_i > 2$, $p'_k \leq d_k/2$. In the qubit-qutrit case these will both have seven elements, the increment basis

$$\{H_2, T, X_3, C_2(X_2), C_2(X_3), C_3(X_2), C_3(X_3)\},$$

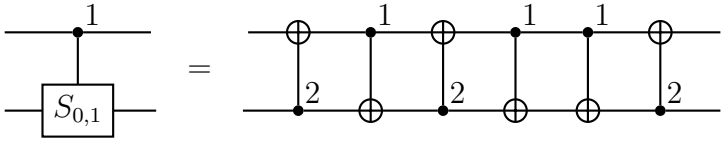
and the transposition basis

$$\{H_2, T, X_3, C_2(X_2), C_2(S_{0,1}), C_3(X_2), C_3(S_{0,1})\}.$$

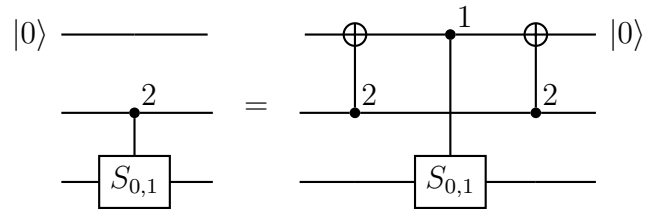
We will now show how these two bases are directly equivalent to each other, though generating the increment basis from the transposition basis will be much less expensive than the other way around. Five of the operations are common between the two bases, so the only difference is that one contains $C_2(X_3)$ and $C_3(X_3)$, where the other contains $C_2(S_{0,1})$ and $C_3(S_{0,1})$. In general $C_d(A)C_d(B) = C_d(AB)$, so setting $A = S_{0,1}$ and $B = S_{1,2}$ we get $C_d(X_3)$. We can write this as an explicit quantum circuit in the transposition basis.



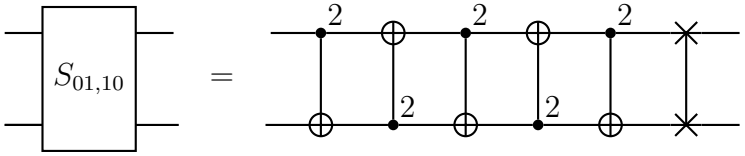
Since d_k is arbitrary in this construction, we have implemented both of the controlled increment operations $C_2(X_3)$ and $C_3(X_3)$. The reverse is more contrived; we will later find by brute force that X_2, X_3, $C_2(X_3)$, and $C_3(X_2)$ generate all permutations on an $N = 6$ composite system, and in particular $C_2(S_{0,1})$ becomes a convoluted sequence of controlled increments.



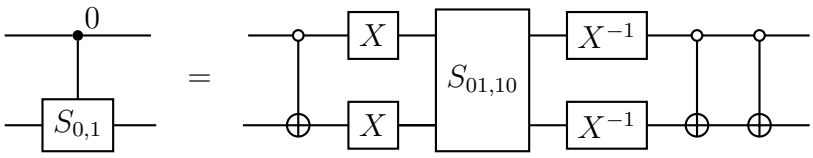
We cannot directly generate $C_3(S_{0,1})$ using $C_3(X_3)$ due to parity, since $C_3(S_{0,1})$ is an odd number of transpositions and $C_3(X_3)$ is an even number of transpositions. One thing we can do is introduce an auxiliary qubit, and use $C_2(S_{0,1})$ as above:



Another option we have is to introduce the Clifford SWAP and SUM operations, and appeal to the equivalences shown in [3]. $S_{01,10}$ in a two qutrit system was shown to be implementable by five $C_3(X_3)$ gates and a SWAP.

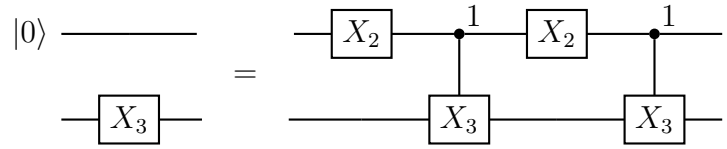


Then $C_{r_2=0}(S_{0,1})$ is identical to $S_{00,01}$, which is a reflection in $\mathbb{C}^9$ just like $S_{01,10}$, and in fact in [3] these operations were transformed to each other using Clifford operations X_3 and SUM.



These equivalences are theoretically interesting, but at this algebraic level, the transposition basis appears to be significantly more efficient than the increment basis. Despite this, it is hard to speculate on what a good basis would be without properly considering the physical constraints and trade-offs of a specific quantum computer. For example, it would not be surprising if $C_2(X_3)$ were more favourable to implement fault tolerantly than $C_2(S_{0,1})$, due to its resemblance to the SUM operation between two qutrits, but it is also possible that neither end up being implemented exactly, and that a different basis is used which approximately implements both of the bases we have presented.

Another decomposition that we have not discussed until now is the implementation of X_3 in terms of $C_2(X_3)$ and X_2.



We do not use the abbreviated $\oplus$ notation in this case to make it clear what is happening in this circuit – we are applying X_3 conditionally, once for every possible condition, which has the same effect as unconditionally applying X_3. In theory this reduces both of our generator sets to only need six gates, but in practice this is absurd, since X_3 is a Pauli operation which will be one of the first and simplest operations to implement on a fault tolerant quantum computer, and we have implemented it in terms of a controlled operation with no known fault tolerant implementation.

In any case we have a set of seven gates that appear useful to implement and work with. All of the examples we have seen of universal computation for non-mixed systems of qubits or qutrits have involved some number of Clifford gates, and a single non-Clifford gate, but in order to generalize the standard binary model of quantum computation we find we require four control gates, only one of which is Clifford, together with the non-Clifford T, for a total of four non-Clifford gates. An interesting avenue of future research would be the question of what the minimum number of non-Clifford gates is for achieving universal computation in mixed systems like this.

Chapter 4

FINITE AND INFINITE GROUPS OF MATRICES

In Section 1.4.1 we outlined the Pauli matrices, and defined the related Weyl-Heisenberg and Clifford groups, including the matrix H_d satisfying $H_d^{-1} Z_d H_d = X_d$. In [6] we saw that H was one of three gates needed for universal computation in a binary quantum computer, but the other two are also directly related to the Pauli matrices. $C(X)$ is of course the controlled version of the X gate, and T when applied four times is equivalent to Z.

$$T^4 = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{\sqrt{2}}(1+i) \end{bmatrix}^4 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = Z.$$

It is for this reason that we are interested in the Pauli and Clifford groups of other number systems. First of all, the Clifford group formed by taking the normaliser of the Weyl-Heisenberg group lets us generalize the algebraic relationships of H_2 to higher dimensions, as well as the algebraic relation $D_2 X_2 D_2^{-1} = i X_2 Z_2 = Y_2$. In Section 4.1 we describe the results of [14], which describes how the Clifford group will behave in quantum computers with multiple objects of different dimension.

Additionally the T gate, which does not directly appear in the Clifford group, proves essential to the generalization of universal computation to other quantum systems. In the qubit case $T_2 H_2^{-1} T_2 H_2$ is an infinite order matrix, which was proven in [6], so in Section 4.2 we outline the technique used to show this, since this is a significant property for a quantum operation to have, mainly because of its importance in the universality of T and H, but also because of the algebraic fact that the Clifford group by itself has no operations like this.

This will guide us in Section 4.3 when we investigate different finitely generated groups to see which ones do and do not contain infinite order operations.

4.1 On Clifford Groups

The paper [14] describes the Clifford group associated with quantum systems with just a single object, as well as the Clifford group associated with arbitrary composite quantum systems, with multiple objects each of different dimension, generalising an earlier paper [9] which showed the same for composite systems with exactly two objects.

Tolar removes the scalar factors from both the Weyl-Heisenberg group, and the Clifford group, not just by taking the group quotient $G/U(1)$, but also by looking at the group conjugation action, mapping the Weyl-Heisenberg group to itself under the map

$$\mathrm{Ad}_A(X^j Z^k) = A X^j Z^k A^{-1}.$$

These conjugation maps form a group under function composition, and are isomorphic to the corresponding quotient groups $H(n)/U(1)$ and $N(H(n))/U(1)$. When applied to the Weyl-Heisenberg group associated with a single quantum object of arbitrary dimension, it was shown that $N(H(n))/U(1)$ was isomorphic to

$$(\mathbb{Z}_N \times \mathbb{Z}_N) \rtimes \mathrm{SL}(2, \mathbb{Z}_N).$$

This isomorphism by itself is not new, and can be used to generate the Clifford group (up to global phase) with four matrices X_N, Z_N, H_N, and D_N. In the special cases $N = 2$ and $N = 3$ only two generators are needed, which we displayed in Section 1.4.1.

The more significant result of the paper however, was the extension to Clifford groups for arbitrary composite systems. The Weyl-Heisenberg group of a composite system $C^{d_1} \otimes \cdots \otimes C^{d_n}$ is taken to be the product of Weyl-Heisenberg groups acting on each individual system, written

$$H(d_1, \ldots, d_n) = \{A_1 \otimes \cdots \otimes A_n \mid A_i \in H(d_i)\} \cong H(d_1) \times \cdots \times H(d_n).$$

While this composite Weyl-Heisenberg group consists only of Kronecker products, its normaliser can contain operations that are not Kronecker products,

for example

$$\text{CNOT} = C(X) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \in N(H(2,2))$$

While Tolar has shown a number of results for the Clifford groups of mixed systems, the most important of these for us was that in a bipartite system $\mathbb{C}^m \otimes \mathbb{C}^n$ with $gcd(m,n) = 1$, the quotient of the Clifford group is simply the direct product of the corresponding single-object Clifford groups:

$$N(H(m,n))/U(1) \cong N(H(m))/U(1) \times N(H(n))/U(1)$$

This means that the Clifford group is *not* capable of acting dependently between quantum objects, making it much less powerful than in non-mixed systems.

Additionally, [14] showed that if m and n have square or cubic divisors in common the corresponding Clifford groups are novel in a way unlike any non-mixed quantum system. This is interesting in the broader context of mixed logic, but will not be relevant to our later discussion of $\mathbb{C}^2 \otimes \mathbb{C}^3$.

4.2 Infinite-Order Gates

In Section 3.4 we stated the result of [6] that universal computation in binary quantum computers can be achieved with just three distinct gates, and the last step was to decompose unitary operations acting on one qubit, into a sequence of H and T operations. The technique for doing this was the main result of [6], and involved constructing a pair of operations which each rotate the Bloch sphere by irrational portions of a full 2π rotation, through different orthogonal planes. Irrational portions of a full turn can be iterated to provide arbitrary approximations of any rotation in the respective plane, and so one can approximate any rotation of the Bloch sphere as three successive rotations each made using just H and T.

One significant algebraic fact about the Clifford group is that up to scale factors it is finite. Contrasting this we see that any group containing both T and H must not be finite, seeing as they can be used to calculate irrational rotations, i.e. infinite order rotations. This means the technique by which these rotations were shown to be irrational/infinite order are of algebraic significance on their own, even before considering the particular application of

universal computation. The technique they used for showing these rotations are irrational was to generate an expression for the angle of rotation, $2\pi\theta$ say, then to show that the unit complex number $e^{i2\pi\theta}$ is not a root of unity, i.e. that θ is rational. This can be done by leveraging the algebraic number theory of cyclotomic polynomials, with the following theorem proven in an appendix of their paper:

Theorem 4.1 (The Cyclotomic/Rational Number Theorem)**.** *For any $\theta \in \mathbb{R}$, θ is rational if and only if $e^{i2\pi\theta}$ is the root of an irreducible polynomial with rational coefficients.*

This analysis works well in the Bloch sphere all unitary matrices will represent a single rotation in some plane, and when dealing with more complicated geometries we can still apply this kind of analysis if we look at the eigenvalues of a given matrix instead. Given a unitary matrix A, we already know that its eigenvalues are unit complex numbers $e^{i2\pi\theta}$, so we characterise as follows.

Proposition 4.2. *Given a unitary matrix A the following are equivalent:*

1. *A is finite order,*
2. *Every eigenvalue of A is finite order under multiplication, i.e. is a root of unity,*
3. *Every eigenvalue of A is a root of a cyclotomic polynomial.*

Proof. By diagonalising A according to the results described in Section 1.1.4 we can write $A = \sum_i e^{i2\pi\theta_i} |v_i\rangle \langle v_i|$, where $e^{i2\pi\theta_i}$ is the eigenvalue corresponding to the normalised eigenvector $|v_i\rangle$. Then if 1. is true we have $A^n = I$, which means $A^n |v_i\rangle = e^{2\pi n\theta_i} |v_i\rangle = |v\rangle$, and hence $n\theta_i \in \mathbb{Z}$, and hence $\theta_i \in \mathbb{Q}$, giving 2. In reverse if 2. is true then we can take n to be the greatest common divisor of each θ_i and raise A to this power to get

$$A^n = \left(\sum_i e^{i2\pi\theta_i} |v_i\rangle \langle v_i| \right)^n = \sum_i e^{i2\pi n\theta_i} |v_i\rangle \langle v_i| = \sum_i |v_i\rangle \langle v_i| = I.$$

Finally 2. and 3. are equivalent by Theorem 4.1. □

Now the eigenvalues of A will be roots of the characteristic polynomial $det(A - \lambda I)$, so if the characteristic polynomial has rational coefficients then we can work directly with this to show that A is of finite or infinite order.

It is interesting to note that $e^{i2\pi\theta}$ and $e^{-i2\pi\theta}$ are both eigenvalues of the rotation

$$R(2\pi\theta) = \begin{bmatrix} \cos(2\pi\theta) & -\sin(2\pi\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix},$$

so in this way [6] was already dealing with eigenvalues, and we simply generalize this to rotations in higher dimensional spheres than the Bloch sphere. One significant difference between their use and ours is that the Bloch sphere is designed to remove global phase factors, and so an operation like $e^{i2\pi\theta}I$ will act trivially on the Bloch sphere, whereas under our eigenvalue analysis this operation might have infinite order despite being trivial in practice. This means that if we want to show that a matrix has infinite order up to global phase, then we must show that some scalar multiple of it has some eigenvalues which are roots of unity, and others which are not. For example let θ be irrational, so that the unit complex number $e^{i2\pi\theta}$ is not a root of unity, then

$$A = \begin{bmatrix} e^{i2\pi\theta} & 0 \\ 0 & e^{-i2\pi\theta} \end{bmatrix}$$

will be infinite order, but so will any non-zero scalar multiple λA, since if $\lambda e^{i2\pi\theta}$ is a root of unity, then $\lambda = e^{i2\pi(\phi-\theta)}$ with ϕ rational, meaning that $\lambda e^{-i2\pi\theta}$ still won't be a root of unity, and vice versa.

4.3 Programmatic Search for Finite Groups

The Clifford group is a significant set of operations for performing fault tolerant computation in quantum computers, and is particularly interesting in the way that adding a single finite order matrix T_2 to it can generate an infinite order group capable of universal computation. Motivated by this we generated groups using various Clifford and non-Clifford operations on the qubit-qutrit system $\mathbb{C}^2 \otimes \mathbb{C}^3$, and found which combinations were finite order and which were infinite order. We saw in [14] that the Clifford group only introduces operations that act dependently between objects when the dimensions of those objects have a common factor, and 2 and 3 do not share a common factor, so the Clifford group in a binary-ternary quantum computer has the same behaviour as two separate Clifford groups, one acting on $\mathbb{C}^2$ and the other on $\mathbb{C}^3$. This means the operations of primary interest to us are the controlled operations discussed in Section 3.4. We consider the ten operations

$$X_2, Z_2, H_2, D_2, X_3, Z_3, H_3, S_3, C_2(X_3), C_3(X_2),$$

as elements of $U(6)$, acting on qubit-qutrit pairs $|p\rangle |q\rangle$.

As an example consider the Pauli operations

$$X_2 = X_2 \otimes I_3 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}, \quad X_3 = I_2 \otimes X_3 = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix},$$

$$Z_2 = Z_2 \otimes I_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \omega_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \omega_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \omega_2 \end{bmatrix}, \quad Z_2 = Z_2 \otimes I_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \omega_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \omega_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \omega_2 \end{bmatrix}.$$

Further, the non-Clifford operations $C_2(X_3)$ and $C_3(X_2)$ look like the following:

$$C_2(X_3) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \qquad C_3(X_2) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

and different combinations of permutation matrix, starting with $C_2(X_3)$ and $C_3(X_2)$, to create different generators of subgroups of $U(6)$. We implement a breadth first search in the groups generated by these sets using the C programming language, and find some combinations which yield finite groups, and others that cannot be fully enumerated without crashing due to integer overflow, which we will present and discuss.

The first group we will discuss is the one generated by removing H_2 and H_3 from the generator set, and adding both $C_2(X_3)$ and $C_3(X_2)$ in their place. All of these generators sit inside of the generalized symmetric group $S(6,6)$ defined in Definition A.32, and so the group generated by them must also be a subgroup of $S(6,6)$. $S(6,6)$ has exactly $6!6^6$ elements, making it large but finite.

In fact X_2, X_3, $C_2(X_3)$ and $C_3(X_2)$ will turn out to generate the whole symmetric group on six elements, so having them all would be quite powerful for

computation, but now we run into a different problem, which is that our group has a way of doing nontrivial things with entangled states, but no longer any way of creating states other than simple multiples of the computational basis! This generalized permutation group provides essential operations, but isn't much more powerful than the Pauli group without H_2 or H_3, so our primary focus will be understanding the trade-offs between different combinations of H_2, H_3, $C_3(X_2)$, and $C_2(X_3)$.

4.3.1 Representing Complex Matrices in C

It would be standard to use some algebraic program or package such as Magma to solve this problem, but because our goal is simple enough to implement in a handful of sittings, it was just as easy to implement it from scratch in C, yielding a program that is simple, high performance, and easy to customize. The full source code of this program is given in Appendix C, and was tested using the Tiny C Compiler, but run using the Clang compiler with optimization level 2. We will now outline the techniques used, since they are of independent interest.

The first obstacle in not using an algebraic package is the exact representation of numbers whose binary expansion never terminates or repeats, for example the sixth root of unity

$$\omega_6 = e^{\frac{i\pi}{3}} = \frac{\sqrt{3}}{2} + \frac{1}{2}i.$$

To solve this we aim to represent the field extension

$$\text{Num} = \mathbb{Q}[\sqrt{-1}, \sqrt{2}, \sqrt{3}] = \left\{ \sum_{j,k,l=0}^{1} n_{jkl}\ i^j \left(\sqrt{2}\right)^k \left(\sqrt{3}\right)^l \;\middle|\; n_{jkl} \in \mathbb{Q} \right\}$$

using an array of 32-bit integers, eight signed integers representing the numerator in $\mathbb{Z}[\dots]$, and one unsigned integer for the denominator.

We have a few basic operations to define in Num, the most significant of which are multiplication, and inverses. We implement these algorithms naively, and since these are the most fundamental building blocks of our program, this will lead to very bad performance without optimization, (in fact we found the program runs four times worse, twenty seconds vs five seconds) but when the search is going to take numerous seconds anyway, the additional second or two of optimization is less painful than it would be in other projects.

For multiplication in Num, we implement multiplication in $\mathbb{Z}[X, Y, Z]$, iterat-

ing six variables over the range $0, 1$ according to the equality

$$\left(\sum_{j,k,l=0}^{1} a_{jkl} X^j Y^k Z^l\right)\left(\sum_{j,k,l=0}^{1} b_{jkl} X^j Y^k Z^l\right)$$
$$= \sum_{j_1,k_1,l_1=0}^{1} \sum_{j_2,k_2,l_2=0}^{1} a_{j_1k_1l_1} b_{j_2k_2l_2} X^{j_1+j_2} Y^{k_1+k_2} Z^{l_1+l_2},$$

then we make three reductions, according to the rules $x^2 = -1$, $y^2 = 2$, and $z^2 = 3$, returning us to Num. When calculating the product of two elements of Num we generally also divide out any common factors of the nine integers in our representation, so that each number has a unique 36 byte representation in memory.

To calculate the inverse of $a \in$ Num, we set $x_0 = a$ and $y_0 = 1$, make repeated modifications of the form $x_{j+1} = x_j c_j$, $y_{j+1} = y_j c_j$. In this way we eventually get $x_k = 1$, while preserving the identity $x_j = a y_j$, meaning $y_k = a^{-1}$. We set c_0 to the denominator of x_0 so that $x_1 \in \mathbb{Z}[i, \sqrt{2}, \sqrt{3}]$, then remove each radical by setting c_k to the conjugate of x_k. That is c_1 is x_1 but with i replaced by $-i$, so that $x_2 \in \mathbb{Z}[\sqrt{2}, \sqrt{3}]$. Repeating for $\sqrt{2}$ gives $x_3 \in \mathbb{Z}[\sqrt{3}]$, and then repeating for $\sqrt{3}$ gives $x_4 \in \mathbb{Z}$. Now we can set $c_4 = 1/x_4$ to get $x_5 = 1$ and hence $y_5 = a^{-1}$. For example if $a = \omega_6 = (\sqrt{3} + i)/2$, then the process looks like

$$\begin{array}{lll}
x_0 = (\sqrt{3} + i)/2 & y_0 = 1 & c_0 = 2, \\
x_1 = \sqrt{3} + i & y_1 = 2 & c_1 = \sqrt{3} - i, \\
x_2 = 4 & y_2 = 2\sqrt{3} - 2i & c_2 = 4, \\
x_3 = 16 & y_3 = 8\sqrt{3} - 8i & c_3 = 16, \\
x_4 = 256 & y_4 = 128\sqrt{3} - 128i & c_4 = 1/256, \\
x_5 = 1 & y_5 = (\sqrt{3} - i)/2. &
\end{array}$$

In practice we could have stopped earlier, and in fact for any root of unity it will be enough to multiply by the complex conjugate, but checking for these conditions of early termination is very likely to be slower for a CPU than to simply finish the calculation as we have above.

Next we represent a complex matrix as a 6×6 array of Num, taking a total of 1296 bytes per matrix. Matrix multiplication is defined in the usual way,

by setting $c_{ik} = \sum_j a_{ij} b_{jk}$, but reduced/coset matrix multiplication was also defined, where the matrix is scaled so that the first non-zero entry of the matrix is exactly 1, giving a consistent representative for each coset in $U(6)/U(1)$. We also represent a permutation in $\mathcal{S}_6$ as six bytes, representing a lookup table of a map $\{1 \dots 6\} \to \{1 \dots 6\}$.

4.3.2 The Search Algorithm

We now have an implementation of two groups, $GL(6, \mathrm{Num})$, and $\mathcal{S}_6$, so now we can implement the search itself. The search has six non-optional parameters:

- `gen_len` the amount of 'letters' in the generator set,
- `gen`, the array of letters, each of type `void*`,
- `names`, string data to print when describing new words that were found,
- `elem_size`, the size in memory of an element of the group,
- `compose`, a binary operation acting on group elements,
- `print_elem`, a procedure to display the group element (matrix or permutation) that has been generated.

The search then generates an array of `PathNode`s, (named after the geometric interpretation of groups as 'Cayley graphs') which are tuples containing:

- the word length of the corresponding element generated,
- a pointer to an earlier `PathNode`, called the predecessor; may be `NULL`,
- the leftmost letter of the word, which when added to the predecessor gives the current word in question,
- a `void*` pointer to the unique group element that was found, called `result`.

The procedure also prints each object as it generates them, for example:

```
X3 D2 X2:
0, 0, 0, 0, 0, -i,
0, 0, 0, -i, 0, 0,
```

```
0, 0, 0, 0, -i, 0,
0, 0, 1, 0, 0, 0,
1, 0, 0, 0, 0, 0,
0, 1, 0, 0, 0, 0,
```

The algorithm itself is a breadth first search, where the array of `PathNode`s is also used as the queue of elements not yet searched. The array is initialized to contain the letters themselves, as words of length 1 with no predecessor, and a pointer `curr` is initialized to the start of the array. Then new group elements are generated via `compose(gen[i], curr->result)`, checked for uniqueness, and if they are unique they are appended to the array. Once `curr` reaches the end of the array, we will have proven that the group is finite, and have enumerated all of its elements.

Checking uniqueness becomes quite slow, the biggest group that was exhaustively generated had 165888 matrix elements, meaning the minimum number of matrix comparisons needed would have been 13759331328, which is like comparing sixteen terabytes of data, one kilobyte at a time. In order to overcome this we also implement an open addressed hash map, using a technique called Robin Hood hashing [7]. The hash map takes the `void*` corresponding to the group element in question, and generates a hash by literally shuffling bits around using the XOR and SHIFT operations offered in C. The hash is then used as an offset into an additional array, called the hash table, where another pointer to the group element will be stored. When a new group element is calculated, rather than search for that group element within a whole megabyte of existing elements, we simply calculate its hash, and check if the element we are looking for is in the correct location in the hash table.

Things are complicated when two group elements correspond to the same location in the hash table, which is where the lookup algorithm must 'probe' for other locations near the offset associated with the hash, to store the new entries there instead. Robin hood hashing/probing is a small optimization on this concept, allowing entries to be moved again after being written the first time, to make room for other entries that would otherwise end up too far from their original offset; minimizing this 'probe distance' from the original offset of a hash entry makes lookups faster, especially as the hash table becomes full. Another technical detail is the fact that since we never remove entries from the hash table, and only run the program once before exiting, we don't need to allocate, reallocate, ore remove entries from the hash table; we simply use a global block of memory, and implement lookup and insert procedures, and we are done.

With this technology in place, all that remains is to wire it together, with definitions for `compose`, and `print_elem`. For permutations, we compose x

and y by setting `xy[i] = x[y[i]]`, and for matrices we multiply, and divide out the first non-zero scale factor, essentially implementing the quotient group $U(6)/U(1)$. We also check the order of the matrix, as a simple heuristic for detecting infinite order groups; If we can calculate the order without causing an integer overflow then we continue generating the group, but if any of the 324 integers in our matrix representation are above some threshold, we assume the order is infinite, represented by `-1`, and print this fact, to indicate that the search is unlikely to terminate. We actually continue regardless of the order, since it might be worth knowing which words of similar size have infinite order as well.

4.3.3 Results

Call the six generators of the Clifford group other than H_2 and H_3 the reduced generator set R. Then $\langle R, H_2, H_3\rangle$ is a finite Clifford group, and $\langle R, C_2(X_3), C_3(X_2)\rangle$ is a subgroup of the generalized symmetric group and therefore finite as described previously, but additionally, $\langle R, H_2, C_3(X_2)\rangle$ and $\langle R, H_3, C_2(X_3)\rangle$ are finite.

The last two generators we could consider, $\langle R, H_2, C_2(X_3)\rangle$, and $\langle R, H_3, C_3(X_2)\rangle$, both trigger an integer overflow when calculating the order of the products $C_2(X_3)H_2$, and $C_3(X_2)H_3$.

The exact value of the former product is

$$C_2(X_3)H_2 = \frac{1}{\sqrt{2}}\begin{bmatrix}1&0&0&0&0&0\\0&1&0&0&0&0\\0&0&1&0&0&0\\0&0&0&0&0&1\\0&0&0&1&0&0\\0&0&0&0&1&0\end{bmatrix}\begin{bmatrix}1&0&0&1&0&0\\0&1&0&0&1&0\\0&0&1&0&0&1\\1&0&0&-1&0&0\\0&1&0&0&-1&0\\0&0&1&0&0&-1\end{bmatrix}$$
$$= \frac{1}{\sqrt{2}}\begin{bmatrix}1&0&0&1&0&0\\0&1&0&0&1&0\\0&0&1&0&0&1\\0&0&1&0&0&-1\\1&0&0&-1&0&0\\0&1&0&0&-1&0\end{bmatrix}.$$

For a 6×6 matrix this seems fairly innocuous, and the other doesn't look

much worse

$$C_3(X_2)H_3 = \frac{1}{\sqrt{3}}\begin{bmatrix}1&0&0&0&0&0\\0&1&0&0&0&0\\0&0&0&0&0&1\\0&0&0&1&0&0\\0&0&0&0&1&0\\0&0&1&0&0&0\end{bmatrix}\begin{bmatrix}1&1&1&0&0&0\\1&\omega_3&\omega_3^2&0&0&0\\1&\omega_3^2&\omega_3&0&0&0\\0&0&0&1&1&1\\0&0&0&1&\omega_3^2&\omega_3\\0&0&0&1&\omega_3&\omega_3^2\end{bmatrix}$$
$$= \frac{1}{\sqrt{3}}\begin{bmatrix}1&1&1&0&0&0\\1&\omega_3&\omega_3^2&0&0&0\\0&0&0&1&\omega_3^2&\omega_3\\0&0&0&1&1&1\\0&0&0&1&\omega_3&\omega_3^2\\1&\omega_3^2&\omega_3&0&0&0\end{bmatrix},$$

but if we raise the simpler of these to the fourth power, it becomes quite chaotic.

$$(C_2(X_3)H_2)^4 = \frac{1}{4}\begin{bmatrix}2&-1&3&0&-1&1\\3&2&-1&1&0&-1\\-1&3&2&-1&1&0\\-1&1&0&3&-1&2\\0&-1&1&2&3&-1\\1&0&-1&-1&2&3\end{bmatrix}.$$

The second power is $C_2(X_3)H_2C_2(X_3)H_2$, which resembles the infinite order term from [6], in that it is a non-Clifford matrix, times the Fourier transform of that matrix; when we ask Wolfram Mathematica for its characteristic polynomial we get

$$1/4(\lambda-1)^2(4\lambda^4+2\lambda^3-3\lambda^2+2\lambda+4).$$

This rational polynomial has an irreducible factor that is cyclotomic and another that is non-cyclotomic, which by the reasoning used in [6] tells us that as we expected, one of its eigenvalues is a root of unity, and another is not, meaning no scale multiple of this matrix is finite order, and so $\langle R, H_2, C_2(X_3)\rangle$ really is an infinite group. Further $C_2(X_3)H_2$ is also infinite order, since two of its eigenvalues will square to roots of unity, and are therefore also roots of unity, whereas the other four will square to non-roots of unity, and are therefore also not roots of unity.

Inspired by this, we calculate

$$C_3(X_2)H_3^{-1}C_3(X_2)H_3 = \frac{1}{3}\begin{bmatrix} 2 & \omega_6 & \omega_6^2 & 1 & \omega_6^4 & \omega_6^5 \\ \omega_6^2 & 2 & \omega_6 & \omega_6^5 & 1 & \omega_6^4 \\ \omega_6^4 & \omega_6^5 & 1 & \omega_6 & \omega_6^2 & 2 \\ 1 & \omega_6^4 & \omega_6^5 & 2 & \omega_6 & \omega_6^2 \\ \omega_6^5 & 1 & \omega_6^4 & \omega_6^2 & 2 & \omega_6 \\ \omega_6 & \omega_6^2 & 2 & \omega_6^4 & \omega_6^5 & 1 \end{bmatrix},$$

but can only conjecture that this is infinite order, given the complexity of dense, non-rational characteristic polynomials. Further, there are 720 different permutations in $\mathcal{S}_6$, twelve of which are already in the Clifford group, and we have shown manually that one of the remaining 708 is infinite order when multiplied with H_2. It is quite difficult to prove that a matrix is infinite order using this method, especially when it isn't sparse the way H_2 is, and often the characteristic polynomial has $\sqrt{2}$ or $\sqrt{3}$ coefficients which would require deeper number theory than we have presented here, so we return now to the search for matrices that we can prove are finite order. We can write another C program extending the existing codebase to exhaustively generate all 6^6 tables of integers, and generate matrices for the 720 that are permutations. Then we attempt to calculate the order of PH_2 and PH_3, and report when either or both have a finite order that we could calculate. Other than the twelve Clifford permutations, the only P for which both PH_2 and PH_3 were finite order were the twelve matrices of the form

$$X_2^a C_2(S_{p,q}) X_2^b \text{ where } a, b \in \{0, 1\}, p, q \in \{0, 1, 2\}, p \neq q.$$

This seems promising, but in fact such P will still have $\langle P, H_2, H_3\rangle$ infinite order, since $PH_3^2 = PS_{1,2}$ will not be in the above form, meaning $\langle P, H_2, H_3\rangle$ is still infinite order, and so the only time where $\langle P, H_2, H_3\rangle$ can be finite is if $P \in \mathcal{C}_2$ anyway, meaning P is simply the Kronecker product of a qubit operation and a qutrit operation.

The final result of this program is that we can use it to check that

$$\langle X_2, C_3(X_2), X_3, C_2(X_3)\rangle = \mathcal{S}_6,$$

This is how we know that

$$C_2(S_{p,q}) \in \langle X_2, C_3(X_2), X_3, C_2(X_3)\rangle$$

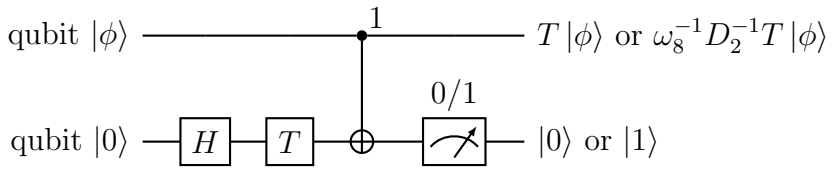
which is where we found the word for $C_2(S_{p,q})$ given in Section 3.4. Additionally since we have $C_2(X_3) = C_2(S_{0,1})X_3C_2(S_{0,1})X_3$, we can infer that

$$\langle X_2, C_3(X_2), X_3, C_2(S_{0,1})\rangle = \mathcal{S}_6$$

also. These results are interesting but it is important not to forget about the qubit-qubit and qutrit-qutrit control gates which were essential for universal computation, which are concealed by this myopic discussion of $U(6)$.

4.3.4 Non-Clifford States

The primary motivation for the result of [6], that $H, T, C_2(X_2)$ is universal, was that all three of these operations can be implemented fault tolerantly. H and $C_2(X)$ had known fault tolerant implementations known prior to this, but the algorithm presented to implement T was done in two steps, the first was to fault tolerantly develop an approximation of the same state $\frac{1}{\sqrt{2}}(|0\rangle + \omega_8 |1\rangle)$ that would be produced by $TH|0\rangle$, and the second step was to use this *state* in a $C_2(X_2)$ gate in order to simulate a T gate. If we use the T operation directly we can draw a diagram demonstrating most of how this is done.



If $|\phi\rangle = a|0\rangle + b|1\rangle$ then after the control operation we will have the state

$$\begin{aligned}
&\frac{1}{\sqrt{2}} C_2(X_2)(|\phi\rangle \otimes (|0\rangle + \omega_8 |1\rangle)) \\
&= \frac{1}{\sqrt{2}} C_2(X_2)(a|0\rangle|0\rangle + a\omega_8 |0\rangle|1\rangle + b|1\rangle|0\rangle + b\omega_8 |1\rangle|1\rangle) \\
&= \frac{1}{\sqrt{2}} (a|0\rangle|0\rangle + a\omega_8 |0\rangle|1\rangle + b|1\rangle|1\rangle + b\omega_8 |1\rangle|0\rangle).
\end{aligned}$$

If we measure $|0\rangle$ in the second qubit then we get $a|0\rangle + b\omega_8 |1\rangle$ in the first qubit, which is $T|\phi\rangle$. On the other hand if we measure $|1\rangle$ then we get $a\omega_8 |0\rangle + b|1\rangle$, and so if we measure $|1\rangle$ we might also be able to conditionally apply $D_2 = \sqrt{Z}$ to get

$$a\omega_8 |0\rangle + b\omega_4 |1\rangle = \omega_8 (a|0\rangle + b\omega_8 |1\rangle),$$

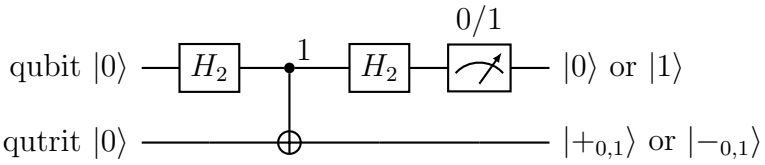
which is also $T|\phi\rangle$ up to global phase.

This is a powerful technique that has been generalised to many other fault tolerant gate implementations, including the P_9 operation presented in [3] which we discussed in Section 2.4. The key insight is that even if you can only perform Clifford operations fault tolerantly, and measure fault tolerantly, then

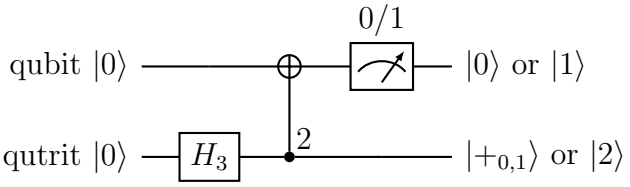
you can still produce the effects of non-Clifford operations by generating non-Clifford states such as the $\frac{1}{\sqrt{2}}(|0\rangle + \omega_8 |1\rangle)$ state shown above. Our operations $C_2(X_3)$ and $C_3(X_2)$ are not Clifford, and as such can produce numerous non-Clifford states, including states that resemble the qubit $|+\rangle$ state, but in a qutrit instead, which we notate as

$$|+_{p,q}\rangle = \frac{1}{\sqrt{2}}(|p\rangle + |q\rangle), \qquad |-_{p,q}\rangle = \frac{1}{\sqrt{2}}(|p\rangle - |q\rangle).$$

First, we present a circuit to consistently produce states of this form using $C_3(X_2)$.



Then, using $C_2(X_3)$ (or $C_2(S_{0,1})$) we can produce these states with probability 2/3, using a different circuit.



Using these or other such non-Clifford states it may become possible for $C_2(X_3)$ or $C_3(X_2)$ to implement each other, but since neither of these operations is Clifford it is hard to say whether this is of any use in fault tolerant contexts specifically. The embedded qubit states may also be interesting for purely ternary calculations, if the $C_2(X_3)$ or $C_3(X_2)$ operations were to be made available fault tolerantly by future research. In all of these cases there is novelty to be explored in these embedded qubit states, and even more motivation to explore coding schemes and fault tolerant operations for mixed quantum systems.

Chapter 5

Conclusion

Quantum computation is a computational paradigm that is still rich with novelty, and while the simplicity and familiarity of binary computation has produced significant results thus far, there are many natural yet unexplored techniques currently being developed for ternary and higher computational systems. In particular mixed-logic that utilises binary and ternary quantum data at the same time, while even less familiar, appears to be just as novel and worth exploring. We have given an introduction to quantum computation from elementary linear algebra up to quantum computation in both binary and non-binary forms. We now summarise the original results we found in the preceding chapters.

In Chapter 2 we generalised the theoretical results of [3], finding that existing algorithms for ternary arithmetic already utilise binary quantum data, and wrote very simple mixed-logic quantum circuits that perform the same calculations while storing this binary data directly within qubits instead of qutrits. We found that in addition to the known properties of the two algorithms considered, one involved fewer qubit-qutrit gates than the other, a new dimension to consider when looking at mixed logic algorithms. Further we briefly discussed techniques that had been used to represent certain qutrit gates as polynomials, which we found do not generalise to mixed contexts in any straight-forward way.

In Chapter 3 we generalised the foundational techniques presented in [2, 11], in order to achieve universal computation by implementing arbitrary unitary operations with circuits consisting of basic gates that act on one or two quantum objects at a time. We found that having a qubit available simplifies the argument of this very abstract technique, compared to systems that have qutrits or higher, but no qubits, in line with the intuition that qubits are sim-

ple and familiar, while qutrits are more novel and potentially powerful. We and also found that the increased diversity of available gates in ternary and higher logic systems required deeper understanding of the universality argument itself, resulting in a proof that is significantly more efficient than the one it was based on, even asymptotically, and even in quantum computers that are purely binary.

Both of these inquiries demonstrated that mixed binary-ternary quantum logic is capable of exploiting the power and novelty of ternary quantum logic and the simplicity and familiarity of binary quantum logic simultaneously. Additionally the basic gates that were used in both of these constructions prompted a programmatic search in Chapter 4 for finitely generated groups with different combinations of single-object and multiple-object gates, where we found equivalences between increment and transposition based ternary logic, and also found that the discrete Fourier transform and the mixed-logic controlled increment have strange behaviour when composed. Regardless of whether the control object is the qubit or the qutrit, the Fourier transform can be applied to the target object to get a finite order operation, but when applied to the control object, the resulting operation is infinite order and produces a variety of novel quantum states in the qubit-qutrit system. This hints at the existence of a very rich variety of unexplored techniques both using these peculiar compositions of gates, and using the resulting quantum states.

Overall we found that mixed logic does seem to combine the strengths of both binary and ternary quantum logic when it comes to algorithm design, and found significant unexplored potential in terms of the basic gates it offers. We now summarise the many questions directly prompted by this research.

5.1 Questions for Future Research

Given how young and novel the field of non-binary quantum computation is, there are numerous distinct topics that are brought into attention by the foundational research done in this thesis. Of primary interest are

1. further algorithm design using basic mixed logic gates,
2. designs of concrete quantum computers capable of performing mixed logic,
3. smaller gate sets capable of universal computation in a mixed quantum computer,

4. schemes for encoding binary, non-binary and mixed data on binary, non-binary and mixed quantum computers,

5. fault tolerant implementation of mixed-logic gates,

6. algebraic and geometric exploration of the strange way that even the simplest mixed-logic gates behave, as compared to the the simple cyclic behaviour available in basic non-mixed logic gates,

7. an investigation into more ways of injecting qubit data into qutrit systems, and the quantum information and quantum computation implications of this, including exploration of the consequences this might have if mixed-logic gates can be implemented fault tolerantly.

Appendix A

Other Definitions and Propositions

A.1 Algebraic Structures

The definitions and propositions in this section and also in Section A.2 are assumed knowledge, and are thus not explained in great detail, but are given anyway, so as to make this thesis more complete and less ambiguous.

Definition A.1 (Binary Operation)**.** A *binary operation* is a map of the form $A \times B \to C$, where A, B, and C, are some sets.
In particular, when we say $\cdot$ is a *binary operation on* A we mean that it is a map of the form $A \times A \to A$. We write $x \cdot y$ infix to mean the image of (x, y) under the map.

Definition A.2 (Associativity)**.** A binary operation $\cdot$ on a set A is *associative* if, for every $a, b, c \in A$, $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.

The point of associativity, of course, is to identify $a \cdot (b \cdot c)$ with $(a \cdot b) \cdot c$, writing both $a \cdot b \cdot c$.

Definition A.3 (Commutativity)**.** A binary operation $\cdot$ on a set A is *commutative* if, for every $a, b \in A$, $a \cdot b = b \cdot a$.

Definition A.4 (Identity)**.** Given a binary operation $\cdot$ on a set A we say that $e \in A$ is an *identity* if, for every $a \in A$, $a \cdot e = e \cdot a = a$.

Proposition A.1 (Uniqueness of Identities)**.** *If e_1 and e_2 are identities of a binary operation $\cdot$ on A, then $e_1 = e_2$.*

Proof. By definition of e_2, $e_1 = e_1 \cdot e_2$, then by definition of e_1, $e_1 \cdot e_2 = e_2$. By transitivity this gives $e_1 = e_2$. □

This proposition tells us that when a set has an identity, it follows that it has only one identity, which we can give a name. We will not dwell on any other foundational propositions of abstract algebra like this.

Definition A.5 (Group)**.** A pair $(A, \cdot)$ is a *group* if $\cdot$ is an associative binary operation on A, with some identity e, and, for every $a \in A$ there is a unique inverse y, satisfying $x \cdot y = y \cdot x = e$. If $\cdot$ is commutative then $(A, \cdot)$ is said to be a *commutative group*, or an *Abelian group*.

Definition A.6. Given a positive integer n, the *symmetric group* $\mathcal{S}_n$ is the set of bijections, or *permutations*, that map $\{0, 1, \ldots, n-1\} \to \{0, 1, \ldots, n-1\}$.

Proposition A.2. *For any positive integer n, $(\mathcal{S}_n, \circ)$ is in fact a group.*

Proof. Function composition $\circ$ is always associative.
Define id to be the map $i \mapsto i$. Then $x \circ \mathrm{id} = \mathrm{id} \circ x = x$, giving us an identity element.
Finally since $x \in \mathcal{S}_n$ is bijective, we can use the function inverse $y = x^{-1}$ to get $xy = yx = \mathrm{id}$. □

From this point on if we talk about a set A which has only one relevant way of being interpreted as a group, or more generally has only one way of being interpreted as an algebraic structure $(A, \alpha_1, \ldots, \alpha_n)$, then we will treat A as if it is itself the group, etc. For example if we talk about the group $\mathbb{Z}$, we mean $(\mathbb{Z}, +)$, not any other group which happens to be well defined.

Definition A.7. [Induced operations] Given a set A which is a subset of a set B, and a binary operation $\cdot : B \times B \to B$, we say that $\cdot$ *induces a binary operation in* A if for every $a_1, a_2 \in A$, the product $a_1 \cdot a_2$ is also in A.

The induced operation being referred to in this phrase is $\cdot$ with the restricted domain $\cdot|_A : A \times A \to B$, since it will now be a well defined operation on A in that $\cdot|_A : A \times A \to A$.

Definition A.8 (Subgroups)**.** Given a group $(B, \cdot)$, or more generally just a binary operation $\cdot$ on a set B, and a set A which is a subset of B, we say that A is a *subgroup* of B under $\cdot$, if $\cdot$ induces a binary operation on A and this binary operation forms a group $(A, \cdot)$.

Definition A.9 (Distributivity)**.** A binary operation $\cdot$ on a set A is said to *distribute over* a binary operation $+$ on the same set A, if for every $a, b, c \in A$, the identities

$$a \cdot (b + c) = a \cdot b + a \cdot c,$$

and

$$(b + c) \cdot a = b \cdot a + c \cdot a$$

both hold. More generally if $\cdot$ is a binary operation $A \times B \to C$, and $+_A, +_B, +_C$ are binary operations on A, B, and C respectively, then $\cdot$ *distributes over* these $+$ operations if for every $a_1, a_2 \in A$, $b_1, b_2 \in B$, the identities

$$a_1 \cdot (b_1 +_B b_2) = a_1 \cdot b_1 +_C a_2 \cdot b_2,$$

and

$$(a_1 +_A a_2) \cdot b_1 = a_1 \cdot b_1 +_C a_2 \cdot b_1$$

both hold.

A simple example of a distributive operation is multiplication $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$, distributing over addition. An example of the more general distributive operator $A \times B \to C$ is matrix multiplication, taking an $l \times m$ matrix and an $m \times n$ matrix, and producing an $l \times n$ matrix.

Definition A.10 (Additivity, Multiplicativity)**.** To aid intuition, and to guide notation, when $\cdot : A \times B \to C$ distributes over $+_A, +_B, +_C$, and these binary operations form commutative groups $(A, +_A)$, $(B, +_B)$, $(C, +_C)$, then we call these groups *additive*, and call $\cdot$ *multiplicative*.

We denote binary operations as $+$ whenever it is obvious that they form an additive group, and omit subscripts such as $+_A$ whenever clear. We also denote the identity of an additive group as 0, and the inverse of x as $-x$. We can then extend summation notation to the additive groups, writing $a_1 + a_2 + \cdots + a_n$ as $\sum_{i=1}^{n} a_n$. Further if $\cdot$ is an associative binary operation on A, and is multiplicative, or simply *not* additive, then we can write products $a_1 \cdot a_2 \cdot \cdots \cdot c_n$ as $a_1 a_2 \dots a_n$, or even $\prod_{i=1}^{n} a_i$. Further, if $(A, \cdot)$ is in fact a group, then we can write the inverses of x as x^{-1}.

Proposition A.3. *Given an additive group $(A, +)$, summations can be swapped in the sense that*

$$\sum_i \sum_j a_{ij} = \sum_j \sum_i a_{ij}$$

This can be proven using induction, but when you unpack the notation these two sums are clearly just reordered versions of each other. This allows us to identify these two sums, and optionally combine the quantifiers as $\sum_{i,j} a_{ij}$.

Proposition A.4. *Given a multiplicative operation $\cdot : A \times B \to C$, distribution can be generalized to*

$$(\sum_i a_i) \cdot (\sum_j b_j) = \sum_{i,j} a_i b_j$$

This is once again proven inductively.

One important special case of multiplicative operations is that of rings.

Definition A.11 (Rings)**.** A triple $(R, \cdot, +)$ is said to be a *ring* if $(R, +)$ is a commutative group, $\cdot$ is an associative binary operation $R \times R \to R$, and $\cdot$ distributes over $+$.

Clearly whenever $(R, \cdot, +)$ is a ring, then $\cdot$ will be multiplicative and $(R, +)$ will be an additive group, and so all rings inherit the notation of additive identities 0 and additive inverses $-x$.

Definition A.12 (Field, Multiplicative group)**.** A ring $(F, \cdot, +)$ is said to be a *field* if $\cdot$ is commutative and $F\backslash\{0\}$ is a subgroup of F under $\cdot$. In this case we call $(F\backslash\{0\}, \cdot)$ the *multiplicative group* of F, which will of course be a commutative group.

Central examples of fields include $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$. In all of these examples, the identity of the corresponding multiplicative group is 1, so just as we denote all additive identities as 0, we will denote the multiplicative identity of any field F as 1.

Definition A.13 (Subfield)**.** Given a field $(F, \cdot, +)$, and a set A which is a subset of F, A is said to be a *subfield* of F under $\cdot$ and $+$ if $\cdot$ and $+$ both induce binary operations on A and $(A, \cdot, +)$ is a field.

Definition A.14 (Linear Groups)**.** Given a field $(F, \cdot, +)$, and a positive integer n, the *general linear group* $GL(n, F)$ is the set of $n \times n$ matrices with determinant not equal to 0. The *special linear group* $SL(n, F)$ is the set of $n \times n$ matrices with determinant equal to 1.

Proposition A.5. *The general linear group $GL(n, F)$ is in fact a group under matrix multiplication, and the special linear group $SL(n, F)$ is a subgroup of this.*

This follows from the multiplicative property of the determinant.

Related to fields are another important case of a multiplicative operation, the vector space.

Definition A.15 (Vector Space)**.** Given a field $(F, \cdot_F, +_F)$, a triple $(V, \cdot_V, +_V)$ is said to be a *vector space over the field* F if all of the following hold:

- $(V, +_V)$ is a commutative group,

- $\cdot_V : F \times V \to V$ distributes over $+_F$ and $+_V$
- for any $a, b \in F$, $v \in V$, the identity $a \cdot_V (b \cdot_V v) = (a \cdot_F b) \cdot_V v$
- for any $v \in V$, $1 \cdot_V v = v$.

We also refer to elements of V as *vectors*, elements of F as *scalars*, and products of the form $a \cdot_V v$ as *scalar multiplication*.

Since $\cdot_F$ and $\cdot_V$ are 'associative' in the above sense, we can write abv or more broadly $a_1 a_2 \dots a_n v$ without ambiguity, once again identifying all of the possible interpretations of these expressions with each other. Further, since $\cdot_V$ and $\cdot_F$ are of different types, and $+_V$ and $+_V$ are also of different types, we drop all of these subscripts without ambiguity.

Proposition A.6. *For any non-negative integer n, and field $(F, \cdot, +)$, the set of column vectors F^n forms a vector space.*

Specifically, F^n is the set

$$\left\{ \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} \;\middle|\; a_i \in F \right\},$$

and vector addition and scalar multiplication are defined in the usual way

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix} = \begin{bmatrix} a_0 + b_0 \\ a_1 + b_1 \\ \vdots \\ a_{n-1} + b_{n-1} \end{bmatrix}$$

$$a \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix} = \cdot \begin{bmatrix} ab_0 \\ ab_1 \\ \vdots \\ ab_{n-1} \end{bmatrix}.$$

Proving that these operations have the required properties is a simple matter of applying the field properties of F coordinate-wise to each of the elements of the column vectors defined by these operations. Note also that F^0 will be the set of empty column vectors, but since we inherit the language of additive groups, we can denote this as vector space as the trivial group $\{0\}$.

A.2 Finite Dimensional Vector Spaces

For all of the following definitions suppose that $(F, \cdot, +)$ is an arbitrary field, and $(V, \cdot, +)$ is an arbitrary vector space over F.

Definition A.16. Given a finite set of vectors $S = \{u_0, \ldots, u_{n-1}\}$ with $u_i \in V$, and a set of scalars $a_0, \ldots, a_{n-1}$, the sum

$$v = \sum_i a_i u_i.$$

is said to be a *linear combination* of $u_0, \ldots u_{n-1}$. Then the *span* of S is the set of all such linear combinations, written $\operatorname{span} S$ or $\operatorname{span}\{u_0, \ldots, u_{n-1}\}$. If $\operatorname{span} S = V$ then S is said to be *spanning*.

Definition A.17. A finite set of vectors $S = \{u_0, \ldots, u_{n-1}\}$ is said to be *linearly independent* if the only linear combination of its elements

$$v = \sum_i a_i u_i.$$

with $v = 0$ is the trivial $0 = a_0 = a_1 = \cdots = a_{n-1}$. If S is both spanning and linearly independent, then S is said to be a *basis* of V.

Proposition A.7. *If $S = \{u_i\}$ is linearly independent, and two linear combinations $\sum_i a_i u_i$ and $\sum_i b_i u_i$ are equal, then each a_i is equal to the corresponding b_i.*

Proof. We have

$$\sum_i a_i u_i = \sum_i b_i u_i,$$

and so by subtracting one side from the other,

$$\sum_i a_i u_i - b_i u_= 0.$$

Now S is linearly independent, so since this linear combination gives the zero vector, it must be the trivial linear combination

$$(a_i - b_i) = 0 \forall i,$$

but undoing the subtraction gives

$$a_i = b_i \forall i.$$

□

Now, if every linear combination of a set of vectors will be the unique such linear combination, then we can extract the corresponding scalars in a well defined way.

Definition A.18. If S is a finite set of vectors $\{u_0, \ldots, u_{n-1}\}$, then the *coordinates* of a vector $v \in \operatorname{span} S$ are the unique scalars a_i that give

$$v = \sum_i a_i u_i.$$

In particular if S is a basis of V then every vector $v \in V$ will have unique coordinates.

A crucial example of all of the above concepts is F^n, equipped with a simple and natural basis.

Definition A.19. The *canonical basis* of the set of column vectors F^n is the set of vectors u_i, whose ith entry is 1 and the rest are 0.[1]

$$u_0 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, u_1 = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, u_{n-1} = \ldots, \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

Proposition A.8. *The canonical basis of F^n is in fact a basis of F^n.*

Proof. We can prove $\{u_i\}$ is spanning by reading the coordinates directly out of $v \in F^n$,

$$v = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \sum_i a_i u_i.$$

Then since this formula is arbitrary in the scalars a_i, we can read it in reverse to see that every linear combination of u_i will be of the form

$$v = \sum_i a_i u_i = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix},$$

[1] Later in Section 1.1.2 these will be written $|i\rangle$

so if $v = 0$, then

$$\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix},$$

meaning a_i are all 0 and the linear combination was in fact trivial. This means $\{u_i\}$ is linearly independent, and spanning, which by definition makes it a basis of F^n. □

Bases tell us something very important about the structure of V, as evidenced by an established fact about bases of vector spaces.

Proposition A.9 (Dimension Theorem for Finite Vector Spaces)**.** *Whenever finite sets S_1 and S_2 are both bases of V, they must have the same size.*

Definition A.20. When V has at least one finite set $S = \{u_0, \ldots, u_{n-1}\}$ that is a basis for V, we say that V is *finite-dimensional*, and define the *dimension* $\dim(V)$ to be the unique size of any basis of V. Since this includes S this means $\dim(V) = |S|$.

A.3 Inner Products, Hilbert Spaces

An important operation in the geometry of vectors is the inner product, which simultaneously indicates the magnitude of vectors, and the angle that different vectors take to each other. While vector spaces are easy to generalize to any field F, for inner products we require specific operations defined on $\mathbb{C}$, which restricts generality a lot. Note that we write the complex conjugate of $a \in \mathbb{C}$ as a^*.

Definition A.21. Given a vector space V over a subfield $\mathbb{F}$ of $\mathbb{C}$, a map $(,) : V \times V \to F$ is called an *inner product* if for every $u, v, w \in V$, and every $a, b \in \mathbb{F}$, the following identities hold:

- conjugate symmetry: $(u, v) = (v, u)^*$ (where a^* is the complex conjugate of a),
- right linearity: $(u, av + bw) = a(u, v) + b(u, w)$,
- positive definite: (v, v) real, and strictly positive whenever $v \neq 0$.

We also say that $(V, \cdot, +, (,))$ is an *inner product space* under $\mathbb{F}$.

Proposition A.10. *Every inner product $(,)$ is left conjugate linear, i.e. for every $u, v, w \in V$, and every $a, b \in \mathbb{F}$ the identity*

$$(au + bv, w) = a^*(u, w) + b^*(v, w).$$

Proof.

$$\begin{aligned}(au + bv, w) &= (w, au + bv)^* \\ &= (a(w, u) + b(w, v))^* \\ &= a^*(w, u)^* + b^*(w, v)^* \\ &= a^*(u, w) + b^*(v, w).\end{aligned}$$

□

Proposition A.11. *If for every scalar $a \in \mathbb{F}$ we have $a^* \in \mathbb{F}$, then the map*

$$\left(\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}, \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} \right) = \begin{bmatrix} a_0^* & a_1^* & \cdots & a_n^* \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} = \sum_{i=0}^{n-1} a_i^* b_i$$

is an inner product in $\mathbb{F}^n$.

Once again, this is straight-forward to prove using the algebraic properties of $\mathbb{C}$. This is not the only inner product one can define for $\mathbb{F}^n$, but it is the one we will be using.

Definition A.22 (Norm)**.** Given an inner product space $(V, \cdot, +, (,))$ over a field $\mathbb{F}$, the *norm* or *length* of a vector $v \in V$ is the non-negative real number

$$\|v\| = \sqrt{(v, v)}.$$

Proposition A.12. *For any vector v in an inner product space V, the inner product with zero is $(v, 0) = 0$.*

Proof. By basic group arithmetic we have $0 = 0 + 0$, which means

$$(v, 0) = (v, 0 + 0) \qquad = (v, 0) + (v, 0),$$

subtracting $(v, 0)$ from both sides gives $(v, 0) = 0$. □

Proposition A.13. *The length of a vector v in an inner product space V is 0 if an only if the vector is itself 0.*

Proof. We have $\|0\| = \sqrt{(0,0)} = 0$ by Theorem A.12. and inversely we have $\|v\| = \sqrt{(v,v)} > 0$ whenever $v \neq 0$. □

Definition A.23 (Orthogonality, Orthonormality)**.** In an inner product space $(V, \cdot, +, (,))$ over a field $\mathbb{F}$, we say that a pair of non-zero vectors $u_0, u_1 \in V$ are *orthogonal* if $(u, v) = 0$, and we say that a set of non-zero vectors $u_0, \dots u_k \in V$ are *orthogonal* if every distinct pair of vectors is orthogonal. Further we say this set is orthonormal if every vector in the set has length 1.

Proposition A.14. *If a set S of non-zero vectors $u_0, \dots, u_k \subset V$ is orthogonal then S is also linearly independent.*

Proof. Suppose that v is a linear combination $\sum_i a_i u_i$. Then for each $0 \leq j \leq k$,

$$\begin{aligned}(u_j, v) &= \left(u_j, \sum_i a_i u_i\right) \\ &= \sum_i a_i \left(u_j, u_i\right) \\ &= 0 + 0 + \cdots + 0 + a_j \left(u_j, u_j\right) + 0 + \cdots + 0 \\ &= a_j \left\|u_j\right\|^2\end{aligned}$$

If we let $v = 0$ then by Theorem A.12 we get $0 = a_j \left\|u_j\right\|^2$, and by Theorem A.13 we know that $\left\|u_j\right\|^2$ is non-zero, so every a_j will be zero. □

The converse does not hold, but we can always use the Gram–Schmidt process to construct an orthonormal set from a linearly independent set.

Proposition A.15 (Gram–Schmidt)**.** *For every finite dimensional vector space $(V, \cdot, +)$ over the field $\mathbb{R}$ or $\mathbb{C}$ with inner product $(,)$, there is a basis which is orthonormal in the inner product space $(V, \cdot, +, (,))$.*

This process leads to great generality in ones discussion of inner product spaces in the abstract, but in practice many inner product spaces already have an obvious orthonormal basis, one that may be more algebraically convenient than one that comes out of the Gram-Schmidt process anyway.

Proposition A.16. *In an inner product space $(V, \cdot, +, (,))$ over the field $\mathbb{F}$ with orthonormal basis $S = u_0, \ldots, u_{n-1}$, the inner products (u_i, v) will be exactly the coordinates of v in the basis S, meaning $v = \sum_{i=0}^{n-1}(u_i, v)u_i$.*

Proof. Let the coordinates of v be a_i, so that

$$v = \sum_{i=0}^{n-1} a_i v.$$

Then calculate

$$\begin{aligned}(u_i, v) &= \left(u_i, \sum_j a_j u_j\right) \\ &= \sum_j a_j \left(u_i, u_j\right) \\ &= 0 + \cdots + 0 + a_i \left(u_i, u_i\right) + 0 + \cdots + 0 \\ &= a_i \left\|u_i\right\|^2 \\ &= a_i.\end{aligned}$$

□

Definition A.24. A map between two inner product spaces, $\phi : U \to V$ is said to *preserve inner products* if for every $u_1, u_2 \in U$, $(u_1, u_2) = (\phi(u_1), \phi(u_2))$.

Such maps essentially show that U exists embedded inside of V. If they are surjective then they show that U is structurally equivalent to V.

Proposition A.17. *Given an inner product space $(V, \cdot, +, (,))$ over a field $\mathbb{F}$ with an orthonormal basis $S = u_0, u_1, \ldots, u_{n-1}$, the map $\phi : V \to \mathbb{F}^n$,*

$$\phi(v) = \begin{bmatrix}(u_0, v) \\ (u_1, v) \\ \cdots \\ (u_n, v)\end{bmatrix}$$

preserves inner products.

Proof. From Theorem A.16 we have $u = \sum_i (u_i, u)$ and $v = \sum_j (u_i, v)$, which gives

$$\begin{aligned}
(u, v) &= \left(\sum_{i=0} (u_i, u) u_i, \sum_{j=0} (u_i, v) u_i \right) \\
&= \sum_{i=0} (u_i, u)^* \left(u_i, \sum_{j=0} (u_j, v) u_j \right) \\
&= \sum_{i=0} (u_i, u)^* \sum_{j=0} (u_j, v) \, (u_i, u_j) \\
&= \sum_{i=0} (u_i, u)^* \left(0 + \cdots + 0 + (u_i, v) \, (u_i, u_i) + 0 + \cdots + 0 \right) \\
&= \sum_{i=0} (u_i, u)^* (u_i, v) \\
&= \left(\begin{bmatrix} (u_0, u) \\ (u_1, u) \\ \cdots \\ (u_n, u) \end{bmatrix}, \begin{bmatrix} (u_0, v) \\ (u_1, v) \\ \cdots \\ (u_n, v) \end{bmatrix} \right) \\
&= (\phi(u), \phi(v))
\end{aligned}$$

□

Definition A.25 (Hilbert Spaces)**.** In an inner product space V, an infinite sequence of vectors v_i is said to be *Cauchy* if for every $\epsilon > 0$, no matter how small, there is some N after which any $n, m > N$ will satisfy $\|v_n - v_m\| < \epsilon$. The sequence is said to be convergent if there is some vector $v \in V$ so that again, for every $\epsilon > 0$ there is an N after which any $n > N$ will satisfy $\|v_n - v\| < \epsilon$. If every Cauchy sequence in V is convergent then $(V, \cdot, +, (,))$ is said to be a *complete* inner product space, also known as a *Hilbert* space.

Proposition A.18. *For every non-negative integer n, the inner product spaces $\mathbb{R}^n$ and $\mathbb{C}^n$ are complete.*

The proof amounts to showing that a Cauchy sequence in either of these spaces is coordinate-wise Cauchy, then finding a limit of each of these coordinates using the completeness of $\mathbb{R}$.

A.4 Group Definitions

Definition A.26. A relation $\sim$ between elements of a set S is called an *equivalence relation* if it satisfies the following three properties, for any $a, b, c \in S$:

- Reflexivity: $a \sim a$,
- Symmetry: $a \sim b \iff b \sim a$,
- Transitivity: $a \sim b, b \sim c \implies a \sim c$.

Equivalence relations are a powerful way of identifying different elements of a set with each other. The set quotient is an even more powerful way of exhibiting the structure of the equivalence relation as a concrete set.

Definition A.27 (Set Quotient)**.** Given an equivalence relation $\sim$ on a set S, and an element $x \in S$, the *equivalence class* of x is the subset of S written

$$[x] = \{y \mid x \sim y\}$$

The *set quotient* of S with respect to $\sim$ is the set of equivalence classes

$$S/\sim = \{[x] \mid x \in S\}.$$

Proposition A.19. *For any equivalence relation $\sim$ on a set S, $S/\sim$ partitions S.*

Proposition A.20. *In an equivalence relation $\sim$ on a set S, any two elements $x, y \in S$ will satisfy $x \sim y$ exactly when $[x] = [y]$.*

Sometimes set quotients of algebraic structures will themselves have an algebraic structure.

Definition A.28. Given an equivalence relation $\sim$ on a group G, we say that *G induces a binary operation on $G/\sim$* if the binary operation

$$[x] \cdot [y] \mapsto [xy]$$

is well defined, i.e. if this product does not depend on the choice of x and y, so that whenever $x' \in [x]$ and $y' \in [y]$ we have $[xy] = [x'y']$.

Proposition A.21. *If a group G induces a binary operation on $G/\sim$, then $(G/\sim, \cdot)$ is also a group.*

Proof. We have associativity by $([x][y])[z] = [xyz] = [x]([y][z])$.
We have an identity by $[e][x] = [x][e] = [x]$.
We have inverses by $[x][x^{-1}] = [xx^{-1}] = [e]$. □

Definition A.29. A subgroup H of a group G is said to be a *normal subgroup* if, for every $x \in H$, $z \in G$, the group product $xzx^{-1} \in G$.

As an example of a normal subgroup, take G to be any subgroup of $GL(n, \mathbb{R})$, and H to be the set of 'scalars' in G, that is the set $\{\lambda I \mid \lambda \in \mathbb{C}\} \cap G$. Since scalars are commutative, it is straight-forward that $g\lambda I g^{-1} = \lambda g g^{-1} = \lambda I \in H$.

Proposition A.22. *If a group G induces a binary operation on some set quotient $G/\sim$, then the equivalence class $[e]$ is a normal subgroup.*

Proof. First, suppose that $e \sim x$ and $e \sim y$, then

$$[e] = [e][e] = [x][y] = [xy],$$

meaning $e \sim xy$ as well, so the binary operation of G is induced (in the sense of Definition A.7) in $[e]$.

Now by reflexivity we have $e \sim e$, which gives the identity element $e \in [e]$, and if $e \sim x$ then

$$[x^{-1}] = [e][x^{-1}] = [x][x^{-1}] = [xx^{-1}] = [e],$$

meaning $e \sim x^{-1}$, which gives inverses x^{-1} in $[e]$. This makes $[e]$ a subgroup of G.

Finally, given $x \in [e]$ and $z \in S$ then

$$[zxz^{-1}] = [z][x][z^{-1}] = [z][e][z^{-1}] = [zez^{-1}] = [e],$$

meaning $zxz^{-1} \in [e]$ as well, making $[e]$ normal. □

Not only is $[e] \in G/\sim$ a normal subgroup of G, but this set allows us to characterize $\sim$ even further. In group theory one can write any equation $g_1 = g_2$ as $g_1 g_2^{-1} = e$, which in the set quotient $G/\sim$ becomes $[xy^{-1}] = [e]$, i.e. $xy^{-1} \sim e$. We now draw this out formally, in terms of $[e]$ as a set.

Proposition A.23. *If a group G induces a binary operation on some set quotient $G/\sim$, and $x, y \in G$, then $x \sim y$ exactly when $xy^{-1} \in [e]$.*

Proof. If we have $x \sim y$, then equivalently $[x] = [y]$, which we can now manipulate algebraically. Clearly $[x][y^{-1}] = [y][y^{-1}]$, meaning $[xy^{-1}] = [e]$, and hence $xy^{-1} \sim e$. By symmetry we have $e \sim xy^{-1}$, and so $xy^{-1} \in [e]$.

Conversely if $xy^{-1} \in [e]$, then $[xy^{-1}] = [e]$, and in much the same way

$$[x] = [xy^{-1}y] = [xy^{-1}][y] = [e][y] = [y],$$

giving $x \sim y$. □

This result motivates us to reverse the formulation we have given, by constructing a set quotient out of a normal subgroup rather than constructing a normal subgroup out of a set quotient. First we must check that we have an equivalence relation.

Proposition A.24. *If H is a normal subgroup of the group G, or in fact just a subgroup of G, then $x \sim y \iff xy^{-1} \in H$ is an equivalence relation.*

Proof. $xx^{-1} = e \in H$ gives reflexivity.
If $xy^{-1} \in H$ then its inverse yx^{-1} will be in H as well, giving symmetry.
If $xy{-}1, yz^{-1} \in H$, then their product xz^{-1} will be as well, giving transitivity.
□

Now if H is in fact normal, we can induce a binary operation.

Proposition A.25. *If H is a normal subgroup of G then G induces a binary operation on the set quotient $G/\sim$ given by the equivalence relation*

$$x \sim y \iff xy^{-1} \in H$$

.

Proof. Suppose that $x' \in [x]$ and $y' \in [y]$, i.e. that $xx'^{-1} \in H$ and $yy'^{-1} \in H$. We would like to show that $[xy] = [x'y']$.

Since H is normal, and yy'^{-1} is in H, $x'\left(yy'^{-1}\right)x'^{-1}$ will be in H as well. Now the product of xx'^{-1} with this will be $xyy'^{-1}x'^{-1}$, which must be in H also. This is exactly $xy\left(x'y'\right)^{-1}$, meaning $xy \sim x'y'$, and so $[xy] = [x'y']$ as required. □

We have now characterised a special case of set quotients directly in terms of the concepts of group theory. Given the foundational power of set quotients, it might be little surprise that this characterisation we have presented is the basis of many other group theoretic concepts and techniques, and as such is given a name and taken as the standard way of building a group out of a set quotient.

Definition A.30. If H is a normal subgroup of G then the *quotient group* G/H is the group $(G/\sim, \cdot)$, where $\sim$ is the equivalence relation given in Theorem A.24, and $\cdot$ is the binary operation induced by G on $G/\sim$.

Definition A.31 (Cosets)**.** Given a subgroup H of a group G, and elements $g_1, g_2 \in G$, we define the *two-sided coset* g_1Hg_2 to be the set

$$\{g_1hg_2 \mid h \in H\}.$$

When $g_1 = e$ we say that $eHg_2 = Hg_2$ is a *right coset* of H, and when $g_2 = e$ we say that $g_1He = g_1H$ is a *left coset* of H.

Proposition A.26. *If H is a normal subgroup of G, and $g \in G$, then every left coset gH is equal to the corresponding right coset Hg, and the group quotient G/H consistes exactly of these cosets of H.*

As an example of a group with a normal subgroup, consider the following finite group generalising the set of permutation matrices:

Definition A.32 (Generalised Symmetric Group)**.** For each pair of positive integers m, n, define $S(m, n)$ to be the set of unitary matrices $P \in U(n)$ satisfying

$$Pu_i = \exp\left(\frac{i2\pi r_i}{m}\right) u_{\sigma(i)}$$

for every canonical basis u_i, where σ is some permutation in $\mathcal{S}_n$, and $r_0, \ldots, r_{n-1}$ are some set of integers.

Proposition A.27. *For any positive integers m, n, $S(m, n)$ is a subgroup of $U(n)$.*

Proof. Clearly $I \in S(m, n)$ by setting $\sigma = \mathrm{id}$, $r_i = 0$.
If $P_1 \in S(m, n)$ via σ_1 and r_i, and $P_2 \in S(m, n)$ via σ_2 and s_i, then

$$P_1P_2u_i = \exp\left(\frac{i2\pi s_i}{m}\right) P_1u_{\sigma_2(i)} = \exp\left(\frac{i2\pi(r_{\sigma_2(i)} + s_i)}{m}\right) u_{\sigma_1(\sigma_2(i))},$$

so set $r'_i = r_{\sigma_2(i)} + s_i$, and $\sigma = \sigma_1 \circ \sigma_2$, giving $P_1P_2 \in S(m, n)$. Additionally if we take the equation

$$P_1u_i = \exp\left(\frac{i2\pi r_i}{m}\right) u_{\sigma_1(i)},$$

and rearrange, we get

$$P_1^{-1}u_{\sigma_1(i)} = \exp\left(\frac{-i2\pi r_i}{m}\right) u_i$$

So set $r'_i = -r_{\sigma_1^{-1}(i)}$, and $\sigma = \sigma_1^{-1}$ to get $P_1^{-1} \in S(m, n)$. □

Proposition A.28. *The set $\Delta(m,n)$ of diagonal matrices in $S(m,n)$ form a normal subgroup of $S(m,n)$.*

Proof. If $P_1, P_2 \in S(m,n)$ are diagonal, then $\sigma_1 = \sigma_2 = \mathrm{id}$, so by the above formulas we get $\sigma = \sigma_1 \circ \sigma_2 = \mathrm{id}$ or $\sigma = \sigma_1^{-1} = \mathrm{id}$, giving P_1P_2 and P_1^{-1} diagonal. This tells us that $\Delta(m,n)$ is a subgroup of $S(m,n)$.

Further if P is some other matrix in $S(m,n)$ via σ and q_i, and u_i is a canonical basis vector in $\mathbb{C}^n$, then

$$\begin{aligned} P^{-1}P_1Pu_i &= \exp\left(\frac{i2\pi\left(q_i\right)}{m}\right) P^{-1}P_1u_{\sigma(i)} \\ &= \exp\left(\frac{i2\pi\left(q_i + r_{\sigma(i)}\right)}{m}\right) P^{-1}u_{\sigma(i)} \\ &= \exp\left(\frac{i2\pi\left(r_{\sigma(i)}\right)}{m}\right) u_i, \end{aligned}$$

clearly a diagonal matrix, so $\Delta(m,n)$ is normal. □

This group also contains $\mathcal{S}_n$ as a subgroup, of course, but this will not be normal.

If H is a normal subgroup of N and N is a subgroup of G, we might say "H is normal in N" and "N is in G". Despite this terminology, H is not necessarily a normal subgroup in G. With this subtlety in mind we can find exactly such a group N, given any subgroup H of G.

Definition A.33. Given a subgroup H of a group G, the *normaliser* of H, written $N_G(H)$ or simply $N(H)$ is the set

$$\{g \mid g \in G,\ ghg^{-1} \in H \forall h \in H\}.$$

Proposition A.29. *Given a subgroup H of a group G, the normaliser $N_G(H)$ is the maximal subgroup of G with H as a normal subgroup.*

As an example, consider the normaliser of $\mathcal{S}_n$ in the generalised symmetric group.

Proposition A.30. *The normaliser $N_{S(m,n)}(\mathcal{S}_n)$, containing $\mathcal{S}_n$ and contained in $S(m,n)$, is exactly the set of scalar multiples of permutation matrices,*

$$\{\exp\left(\frac{i2\pi s}{m}\right) P \mid s \in \mathbb{Z}, P \in \mathcal{S}_n\}.$$

We can also define equivalence classes *between* groups, based on whether they have the same structure under group multiplication.

Definition A.34. Given two groups G_1 and G_2, a *homomorphism* is a map $\phi : G_1 \to G_2$ satisfying $\phi(x)\phi(y) = \phi(xy)$, for any $x, y \in G_1$. If ϕ is bijective then it is also called an *isomorphism*, and if there is at least one such isomorphism then G_1 and G_2 are said to be *isomorphic*, represented infix as $G_1 \cong G_2$.

Proposition A.31. *The relation $\cong$ between groups, of being isomorphic, is an equivalence relation.*

Proposition A.32. *If ϕ is a homomorphism between G_1 and G_2, then the image $\phi(G_1)$ is a subgroup of G_2, and if ϕ is injective then $G_1 \cong \phi(G_1)$.*

Definition A.35. If F is a field and n is a positive integer, then a matrix $P \in GL(n, F)$ is said to be a *permutation matrix* if there is a permutation $\sigma \in \mathcal{S}_n$ so that

$$Pu_i = u_{\sigma(i)},$$

where u_i is any of the n canonical basis vectors defined in Definition A.19.

By Theorem A.32 the set of permutation matrices form a group isomorphic to the symmetric group $\mathcal{S}_n$, which allows us to identify these two groups, treating $\mathcal{S}_n$ as a subgroup of $GL(n, F)$.

Definition A.36. If G is a group and ϕ is an isomorphism from G to itself, then ϕ is said to be an *automorphism*. The set of automorphisms is written $\mathrm{Aut}(G)$.

Proposition A.33. *For any group G, $(Aut(G), \circ)$ is a group.*

Definition A.37. Given a group $(G_1, \cdot)$, and a subgroup G_2 of $Aut(G)$, the *semi-direct product* $G_1 \rtimes G_2$ is the group $(G_1 \times G_2, \cdot)$ with the product

$$(g_1, f_1) \cdot (g_2, f_2) = (g_1 \cdot f_1(g_2), f_1 \circ f_2).$$

Definition A.38. We say that a subgroup H of a group G is *generated by* a set $S \subset H$, if every $h \in H$ is of the form

$$h = \prod_{i=1}^{k} s_i,$$

for some choice of k and $s_1, \ldots, s_k$, where each s_i is either an element of S or the inverse of an element of S. Clearly the set of elements of G of this form is a subgroup of G, which we call *the subgroup generated by* S, and denote $\langle S \rangle$. If S is a finite set $\{s_1, \ldots, s_n\}$ then we can also write $\langle s_1, \ldots, s_n \rangle$ directly, and say that any group H generated by such a set is *finitely generated*.

Appendix B

Source Code

The following source code was used in the programmatic search in Section 4.3. It was written from scratch in the C programming language, and avoids using any external programs other than the standard `std*.h` libraries. There are three main `.c` source files, which share three `.h` files for common function definitions. Some explanation behind the design of these programs is given in Section 4.3, but this code was not designed for convenient use or reuse, and must be modified directly in order to reconfigure its behaviour. It is primarily included for reference and transparency. The source code is also available at `github.com/spiveeworks/quantum-binary-ternary/tree/master/src` and the data output by a few different configurations of `matgroup.c` can be found at `github.com/spiveeworks/quantum-binary-ternary/tree/master/data`

B.1 hashmap.h

```
#pragma once

#include<stdbool.h>
#include<stdint.h>
#include<string.h>

#ifdef MAX_PATH_COUNT
#define TABLE_LEN MAX_PATH_COUNT
#else
```

```
#define TABLE_LEN 100000
#endif

struct HashNode {
    uint64_t hash;
    void *key;
    void *val;
} table[TABLE_LEN] = {};

uint64_t hash_calc(void *val_v, size_t size) {
    char *val = (char*)val_v;
    uint64_t result = 0xFEDCBA987654321;
    for (size_t i = 0; i < size; i++) {
        char x = val[i];
        // just put the bits in different places,
           nothing special happening
        result = (result << 5U) + (result >> 2U) + (x <<
            17U) + x;
    }
    // final touch to give higher bits more effect, and
       to get correct range
    return (result + (result >> 32U)) % TABLE_LEN;
}

#define swap_cond(ind, curr_hash, new_hash) (((new_hash)
   +TABLE_LEN-(ind)-1)%TABLE_LEN < ((curr_hash)+
   TABLE_LEN-(ind)-1)%TABLE_LEN)

void* hash_lookup(void *key, size_t key_size) {
    uint64_t hash = hash_calc(key, key_size);
    size_t index = hash % TABLE_LEN;
    while (true) {
        if (table[index].key == NULL || swap_cond(index,
            table[index].hash, hash)) {
            return NULL;
        }
        if (memcmp(table[index].key, key, key_size) ==
           0) {
```

```
            return table[index].val;
        }
        index = (index + 1) % TABLE_LEN;
    }
}

void hash_insert(void *key, size_t key_size, void *val)
   {
    struct HashNode new_node;
    new_node.hash = hash_calc(key, key_size);
    new_node.key = key;
    new_node.val = val;
    size_t index = new_node.hash;
    bool done = false;
    while (!done) {
        if (table[index].key == NULL) {
            table[index] = new_node;
            done = true;
        } else if (swap_cond(index, table[index].hash,
           new_node.hash)) {
            struct HashNode swap = table[index];
            table[index] = new_node;
            new_node = swap;
        } else if (memcmp(table[index].key, new_node.key
           , key_size) == 0) {
            table[index] = new_node;
            done = true;
        }
        index = (index + 1) % TABLE_LEN;
    }
}
```

B.2 generator.h

```
#pragma once

#include<stdbool.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define MAX_PATH_COUNT 1000000
#include "hashmap.h"

typedef unsigned char u8;

typedef struct { size_t i; } GenIndex;
typedef struct { size_t n; } PathLength;

typedef struct PathNode {
    PathLength len;
    struct PathNode *pred;
    // we're using the convention x . y = x after y
    // so by last we mean leftmost
    GenIndex last;
    void *result;
} PathNode;

typedef struct {
    size_t path_count;
    PathNode *paths;
} PathList;

enum SearchStrategy {
    SEARCH_BREADTH_FIRST,
    SEARCH_DEPTH_FIRST,
};

PathList gen_paths(
    size_t gen_len, void** gen, char **names, size_t
```

```
    elem_size,
    void (*compose)(void*,void*,void*), void (*
    print_elem)(void*),
    bool (*print_cond)(void*), enum SearchStrategy
    strategy
) {
    static PathNode paths[MAX_PATH_COUNT];
    PathNode *next_node = paths;
    size_t path_count = 0;

    struct PathNode *curr = paths;
    GenIndex next_generator = {0};

    for (size_t i = 0; i < gen_len; i++) {
        next_node->len.n = 1;
        next_node->pred = NULL;
        next_node->last.i = i;
        next_node->result = gen[i];
        hash_insert(next_node->result, elem_size,
            next_node);
        next_node++;
        path_count++;
    }

    while (true) {
        bool done = false;
        while (!done && next_generator.i == gen_len) {
            switch (strategy) {
                case SEARCH_BREADTH_FIRST:
                    next_generator.i = 0;
                    curr += 1;
                    done = curr == next_node;
                    break;
                case SEARCH_DEPTH_FIRST:
                    if (curr->pred == NULL) {
                        done = true;
                    } else {
                        next_generator.i = curr->last.i
```

```
                    + 1;
                curr = curr->pred;
            }
            break;
        }
    }
    if (done) { break; }

    u8 result[elem_size];
    compose(result, gen[next_generator.i], curr->
        result);
    if (hash_lookup(result, elem_size) != NULL) {
        next_generator.i += 1;
        continue;
    }
    if (path_count >= MAX_PATH_COUNT) {
        printf("Path List is full!\n");
        exit(1);
    }
    next_node->len.n = curr->len.n + 1;
    next_node->pred = curr;
    next_node->last = next_generator;
    next_node->result = malloc(elem_size);
    memcpy(next_node->result, result, elem_size);
    hash_insert(next_node->result, elem_size,
        next_node);
    if (print_cond == NULL || print_cond(next_node->
        result)){
        PathNode *printing = next_node;
        while (printing) {
            printf(" %s", names[printing->last.i]);
            printing = printing->pred;
        }
        printf(":\n");
        print_elem(next_node->result);
        printf("\n");
    }
    switch (strategy) {
```

```
                case SEARCH_BREADTH_FIRST:
                    next_generator.i += 1;
                    break;
                case SEARCH_DEPTH_FIRST:
                    curr = next_node;
                    next_generator.i = 0;
                    break;
            }
            next_node++;
            path_count++;
        }

        return (PathList){path_count, paths};
    }
```

B.3 matrix_quadratic.h

```
#pragma once

#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

#define range(i, n) for (size_t i = 0; i < (n); i++)

int gcd(int x, int y) {
    while (y != 0) {
        int rem = x % y;
        x = y;
        y = rem;
    }
    if (x < 0) {
        x = -x;
    }
    return x;
}

// stupidly slow when x is big, but intended use case is
    around int_sqrt(36)=6
unsigned int_sqrt(unsigned x) {
    unsigned i = 1;
    while (i*i <= x) {
        i++;
    }
    return i-1;
}
unsigned int_quadpart(unsigned x) {
    unsigned i = 2;
    unsigned result = 1;
    while (i*i <= x) {
        if (x % (i*i) == 0) {
            result *= i;
            x /= i*i;
```

```
        } else {
            i++;
        }
    }
    return result;
}

// this type represents the field Q[sqrt(-1), sqrt(2),
   sqrt(3)]
// @Performance change all of the num_ procedures to use
    pointers? it got big
typedef struct num {
    int c[2][2][2]; // coefficients
    unsigned de; // denominator
} num;

const int MONOMIAL_SQUARES[2][2][2] = {
    {{1,3},{2,6}},
    {{-1,-3},{-2,-6}},
};

num num_from_int(int nu) {
    num out = {};
    out.c[0][0][0] = nu;
    out.de = 1;
    return out;
}

bool num_eq(num x, num y) {
    if (x.de == 0 || y.de == 0) {
        return false;
    }
    range(i, 2) {
        range(j, 2) {
            range(k, 2) {
                if (x.c[i][j][k] * y.de != y.c[i][j][k]
                   * x.de) {
                    return false;
```

```
                }
            }
        }
    }
    return true;
}

num num_reduce(num x) {
    int d = x.de;
    range(i, 2) {
        range(j, 2) {
            range(k, 2) {
                d = gcd(d, x.c[i][j][k]);
            }
        }
    }
    x.de /= d;
    range(i, 2) {
        range(j, 2) {
            range(k, 2) {
                x.c[i][j][k] /= d;
            }
        }
    }
    return x;
}

void num_print(num x) {
    int terms = 0;
    range(i, 2) {
        range(j, 2) {
            range(k, 2) {
                if (x.c[i][j][k] != 0) {
                    terms += 1;
                }
            }
        }
    }
```

```
if (terms == 0) {
    printf("0");
} else if (terms >= 2 && x.de != 1) {
    printf("(");
}
bool first_term = true;
range(i, 2) {
    range(j, 2) {
        range(k, 2) {
            int c = x.c[i][j][k];
            if (c != 0) {
                if (c > 0 && !first_term) {
                    printf("+");
                }
                if (c < 0) {
                    printf("-");
                    c = -c;
                }
                if (c != 1 || (i == 0 && j == 0 && k
                    == 0)) {
                    printf("%d", c);
                }
                if (j == 1 || k == 1) {
                    printf("sqrt(%d)",
                        MONOMIAL_SQUARES[0][j][k]);
                }
                if (i == 1) {
                    printf("i");
                }
                first_term = false;
            }
        }
    }
}
if (terms >= 2 && x.de != 1) {
    printf(")");
}
if (x.de != 1) {
```

```
        printf("/%u", x.de);
    }
}

num num_add(num x, num y) {
    num out;
    out.de = x.de * y.de;
    range(i, 2) {
        range(j, 2) {
            range(k, 2) {
                out.c[i][j][k] = x.c[i][j][k] * y.de + y
                    .c[i][j][k] * x.de;
            }
        }
    }
    return out;
}

bool debug = false;
num num_mul(num x, num y) {
    int out_c[3][3][3] = {};
    range(i1, 2) {
        range(j1, 2) {
            range(k1, 2) {
                range(i2, 2) {
                    range(j2, 2) {
                        range(k2, 2) {
                            out_c[i1+i2][j1+j2][k1+k2]
                                +=
                                x.c[i1][j1][k1] * y.c[i2
                                    ][j2][k2];
                        }
                    }
                }
            }
        }
    }
    range(i, 3) {
```

```
        range(j, 3) {
            out_c[i][j][0] += out_c[i][j][2] *
              MONOMIAL_SQUARES[0][0][1];
            out_c[i][j][2] = 0;
        }
    }
    range(i, 3) {
        range(j, 3) {
            out_c[i][0][j] += out_c[i][2][j] *
              MONOMIAL_SQUARES[0][1][0];
            out_c[i][2][j] = 0;
        }
    }
    range(i, 3) {
        range(j, 3) {
            out_c[0][i][j] += out_c[2][i][j] *
              MONOMIAL_SQUARES[1][0][0];
            out_c[2][i][j] = 0;
        }
    }
    num out;
    range(i, 2) {
        range(j, 2) {
            range(k, 2) {
                out.c[i][j][k] = out_c[i][j][k];
            }
        }
    }
    out.de = x.de * y.de;
    return out;
}

num num_conj_i(num x) {
    range(j, 2) {
        range(k, 2) {
            x.c[1][j][k] = -x.c[1][j][k];
        }
    }
```

```
    return x;
}

num num_conj_2(num x) {
    range(i, 2) {
        range(k, 2) {
            x.c[i][1][k] = -x.c[i][1][k];
        }
    }
    return x;
}

num num_conj_3(num x) {
    range(i, 2) {
        range(j, 2) {
            x.c[i][j][1] = -x.c[i][j][1];
        }
    }
    return x;
}

num num_mod_sq(num x) {
    return num_mul(x, num_conj_i(x));
}

num num_inv(num x) {
    num out = {};
    out.c[0][0][0] = (int)x.de;
    out.de = 1;
    x.de = 1;

    num conj_i = num_conj_i(x);
    x = num_mul(x, conj_i);
    out = num_mul(out, conj_i);

    num conj_2 = num_conj_2(x);
    x = num_mul(x, conj_2);
    out = num_mul(out, conj_2);
```

```
    num conj_3 = num_conj_3(x);
    x = num_mul(x, conj_3);
    out = num_mul(out, conj_3);

    out.de = x.c[0][0][0];
    return out;
}

num num_inv_sqrt(num x) {
    range(i, 2) {
        range(j, 2) {
            range(k, 2) {
                bool zero = x.c[i][j][k] == 0;
                bool is_const = i == 0 && j == 0 && k ==
                    0;
                if (!zero && !is_const) {
                    printf("Tried to take invsqrt of non
                        -rational number\n");
                    exit(1);
                }
            }
        }
    }
    int x_nu = x.c[0][0][0];

    int i = 0;
    if (x_nu < 0) {
        x_nu = -x_nu;
        i = 1;
    }

    int d = gcd(x_nu, (int)x.de);
    x_nu /= d;
    x.de /= d;

    num out = {};
    out.de = int_quadpart(x_nu);
```

```
x_nu /= (int)out.de * (int)out.de;
int out_nu = (int)int_quadpart(x.de);
x.de /= out_nu * out_nu;

int j = 0;
if (x_nu % 2 == 0) {
    j = 1;
    out.de *= 2;
    x_nu /= 2;
}
if (x.de % 2 == 0) {
    j = 1;
    x.de /= 2;
}

int k = 0;
if (x_nu % 3 == 0) {
    k = 1;
    out.de *= 3;
    x_nu /= 3;
}
if (x.de % 3 == 0) {
    k = 1;
    x.de /= 3;
}

if (x.de != 1) {
    printf("Tried to take invsqrt of multiple of %d
       /%d\n", x_nu, x.de);
    exit(1);
}
if (x_nu != 1) {
    printf("Tried to take invsqrt of multiple of %d\
       n", x_nu);
    exit(1);
}

out.c[i][j][k] = out_nu;
```

```
    return out;
}

/* My matrices are just pointers to num
 * (assumed to be pointers to an array of size n*n)
 * C allows statically declared arrays, and we would
    like `num x[4][4];` to
 * declare a usable matrix, and `x[i][j]` to extract an
    appropriate term
 * this means `x[i]` will yield a `num[4]` which by
    convention ought to be a
 * row vector, and so the equivalent index from a flat
    array will be
 * `x[i*n + j]`
 *
 * we could use a newtype like `typedef struct {num *mp
    ;} mat;`, possibly add a
 * size field as well, but this does not save much
    trouble, since we aren't yet
 * using vectors, and usually share size between
    multiple objects.
 *
 * I will probably come to regret this if I continue to
    use this code.
 */

void mat_zero(int n, num *out) {
    range(i, n) {
        range(j, n) {
            out[i*n+j] = num_from_int(0);
        }
    }
}

// performs matrix multiplication
void mat_mul(int n, num *out, num *x, num *y) {
    range(i, n) {
```

```
        range (k, n) {
            num sum = num_from_int(0);
            range (j, n) {
                sum = num_reduce(num_add(sum, num_mul(x[
                    i*n+j], y[j*n+k])));
            }
            out[i*n+k] = sum;
        }
    }
}

// if X = A (x) B = mat_kronecker(m, n, X, A, B)
// then X[KRON_INDEX(m, n, i1, j1, i2, j2)] = A_{i1 j1}*
    B_{i2 j2}
#define KRON_INDEX(m, n, i1, j1, i2, j2) (((i1)*(n)+(i2)
    )*(m)*(n)+((j1)*(n)+(j2)))

// takes the Kronecker product of an m x m matrix and an
    n x n matrix
// out must be an mn x mn matrix
void mat_kronecker(int m, int n, num *out, num *x, num *
    y) {
    range(i1, m) {
        range(i2, n) {
            range(j1, m) {
                range(j2, n) {
                    out[KRON_INDEX(m, n, i1, j1, i2, j2)
                        ] =
                        num_mul(x[i1*m+j1], y[i2*n+j2]);
                }
            }
        }
    }
}

void mat_ident(int n, num *out) {
    mat_zero(n, out);
    range(i, n) {
```

```
        out[i*n+i].c[0][0][0] = 1;
    }
}

// initializes the permutation matrix that maps basis
   vector i to i+1
void mat_shift(int n, num *out) {
    mat_zero(n, out);
    range(i, n) {
        out[((i+1)%n)*n+i].c[0][0][0] = 1;
    }
}

void mat_clock(int n, num *out, num *phase, size_t
   phase_len) {
    mat_zero(n, out);
    range(i, n) {
        out[i*n + i] = phase[i%phase_len];
    }
}

void mat_fourier(int n, num *out, num *phase, size_t
   phase_len) {
    num scale = num_inv_sqrt(num_from_int(n));
    range(i, n) {
        range(j, n) {
            num coord = phase[i*j%phase_len];
            out[i*n+j] = num_reduce(num_mul(scale, coord
               ));
        }
    }
}

// |control>|j> \mapsto |i>(y|j>)
// |else>|j> \mapsto |else>|j>
void mat_control_target(int m, int n, num *out, int
   control, num *y) {
    mat_ident(m*n, out);
```

```
    range(i, n) {
        range(j, n) {
            out[KRON_INDEX(m, n, control, control, i, j)
                ] = y[i*n+j];
        }
    }
}

// |i>|control> \mapsto (x|i>)|control>
// |i>|else> \mapsto |i>|else>
void mat_target_control(int m, int n, num *out, num *x,
    int control) {
    mat_ident(m*n, out);
    range(i, m) {
        range(j, m) {
            out[KRON_INDEX(m, n, i, j, control, control)
                ] = x[i*m+j];
        }
    }
}

// generates a diagonal matrix which along with the
    previous 3 functions give a
// basis for the Clifford quotient group
void mat_cliff_diag(int n, num *out, num *phase, size_t
    phase_len) {
    mat_zero(n, out);
    range(i, n) {
        out[i*n+i] = phase[i*(n-i)%phase_len];
    }
}

// |i> \mapsto |\sigma(i)>
void mat_perm(int n, num *out, const unsigned char *perm
    ) {
    mat_zero(n, out);
    range(i, n) {
        out[perm[i]*n+i].c[0][0][0] = 1;
```

```
    }
}


void mat_reduce(int n, num *x) {
    range (i, n) {
        range (j, n) {
            x[i*n+j] = num_reduce(x[i*n+j]);
        }
    }
}

void mat_print(int n, num *x, char*sepcol, char*seprow)
   {
    range (i, n) {
        range (j, n) {
            num_print(x[i*n+j]);
            printf("%s", sepcol);
        }
        printf("%s", seprow);
    }
}

#define SMALLEST_ROOT 24
bool roots_initialized = false;
num roots[SMALLEST_ROOT];

num root_of_unity(int r, int d) {
    if (SMALLEST_ROOT * r % d != 0) {
        printf("Cannot calculate root of unity, %d does
           not divide %d\n",
            d, SMALLEST_ROOT * r);
        exit(1);
    }

    if (!roots_initialized) {
        num x = num_from_int(1);
        num y = num_from_int(0);
```

```
            y.c[0][1][0] = 1;
            y.c[0][1][1] = 1;
            y.c[1][1][0] = -1;
            y.c[1][1][1] = 1;
            y.de = 4;

            range(i, SMALLEST_ROOT) {
                if (i > 0 && num_eq(x, num_from_int(1))) {
                    printf("Error while initializing roots:\
                       n");
                    num_print(y);
                    printf("raised to %lu is already 1\n", i
                       );
                    exit(1);
                }
                roots[i] = x;
                x = num_reduce(num_mul(x, y));
            }
            if (!num_eq(x, num_from_int(1))) {
                printf("Error while initializing roots:\n");
                num_print(y);
                printf("raised to %d is only ",
                   SMALLEST_ROOT);
                num_print(x);
                printf("\n");
                exit(1);
            }

            roots_initialized = true;
        }
        return roots[SMALLEST_ROOT * r / d % SMALLEST_ROOT];
    }

    bool mat_is_ident(int n, num *x) {
        range(i, n) {
            range(j, n) {
                if (i != j && !num_eq(x[i*n+j], num_from_int
                   (0))) {
```

```
                return false;
            }
            if (i == j && !num_eq(x[i*n+j], num_from_int
               (1))) {
                return false;
            }
        }
    }
    return true;
}

const int MAX_PRECISION = 100;

int mat_calc_order(int n, num *x, bool print) {
    if (mat_is_ident(n, x)) { return 0; }
    num y[n][n];
    range(i, n) {
        range(j, n) {
            y[i][j] = x[i*n+j];
        }
    }
    int order = 1;
    while (!mat_is_ident(n, &y[0][0])) {
        num z[n][n];
        mat_mul(n, &z[0][0], x, &y[0][0]);
        bool overprecise = false;
        range(i, n) {
            range(j, n) {
                y[i][j] = num_reduce(z[i][j]);
                if (y[i][j].de > MAX_PRECISION) {
                    overprecise = true;
                }
                range(a, 2) {
                    range(b, 2) {
                        range(c, 2) {
                            int val = y[i][j].c[a][b][c
                               ];
                            if (val > MAX_PRECISION || -
```

```
                                    val > MAX_PRECISION) {
                                    overprecise = true;
                                }
                            }
                        }
                    }
                }
            }
            order += 1;
            if (overprecise) {
                if (print) {
                    mat_print(n, x, ", ", "\n");
                    printf("\nraised to power %d:\n", order)
                        ;
                    mat_print(n, &y[0][0], ", ", "\n");
                }
                return -1;
            }
        }
        return order;
}

enum Gate {
    GATE_IDENT,
    GATE_SHIFT,
    GATE_CLOCK,
    GATE_FOURIER,
    GATE_CLIFF_DIAG,
    GATE_TRANSPOSE,
    GATE_CONTROL_0,
    GATE_CONTROL_1,
    GATE_CONTROL_N,
};

void initialize_single(int n, num *out, enum Gate gate)
    {
    switch(gate) {
        case GATE_IDENT:
```

```
{
    mat_ident(n, out);
    break;
}
case GATE_SHIFT:
{
    mat_shift(n, out);
    break;
}
case GATE_CLOCK:
{
    num phase[n];
    range(i, n) { phase[i] = root_of_unity(i, n)
        ; }
    mat_clock(n, out, phase, n);
    break;
}
case GATE_FOURIER:
{
    num phase[n];
    range(i, n) { phase[i] = root_of_unity(i, n)
        ; }
    mat_fourier(n, out, phase, n);
    break;
}
case GATE_CLIFF_DIAG:
{
    num phase[2*n];
    range(i, 2*n) { phase[i] = root_of_unity(i,
        2*n); }
    mat_cliff_diag(n, out, phase, 2*n);
    break;
}
case GATE_TRANSPOSE:
{
    unsigned char p[n];
    range(i, n) {
        p[i] = i;
```

```
                }
                p[0] = 1;
                p[1] = 0;
                mat_perm(n, out, p);
                break;
            }
            case GATE_CONTROL_0:
            case GATE_CONTROL_1:
            case GATE_CONTROL_N:
            {
                printf("Tried to initialize a single gate
                   with a control specifier\n");
                exit(1);
            }
            default:
            {
                printf("Unknown gate specifier %d\n", gate);
                exit(1);
            }
        }
    }

    void initialize(int m, int n, num *out, enum Gate gate_a
       , enum Gate gate_b) {
        num a[m][m];
        if (gate_a < GATE_CONTROL_0) {
            initialize_single(m, &a[0][0], gate_a);
        }
        num b[n][n];
        if (gate_b < GATE_CONTROL_0) {
            initialize_single(n, &b[0][0], gate_b);
        }
        if (gate_a < GATE_CONTROL_0 && gate_b <
           GATE_CONTROL_0) {
            mat_kronecker(m, n, out, &a[0][0], &b[0][0]);
        } else if (gate_b < GATE_CONTROL_0) {
            int control;
            switch(gate_a) {
```

```
            case GATE_CONTROL_0:
                control = 0;
                break;
            case GATE_CONTROL_1:
                control = 1;
                break;
            case GATE_CONTROL_N:
                control = m - 1;
                break;
            default:
                printf("Invalid control specifier %d\n",
                    gate_a);
                exit(1);
        }
        mat_control_target(m, n, out, control, &b[0][0])
            ;
    } else if (gate_a < GATE_CONTROL_0) {
        int control;
        switch(gate_b) {
            case GATE_CONTROL_0:
                control = 0;
                break;
            case GATE_CONTROL_1:
                control = 1;
                break;
            case GATE_CONTROL_N:
                control = n - 1;
                break;
            default:
                printf("Invalid control specifier %d\n",
                    gate_b);
                exit(1);
        }
        mat_target_control(m, n, out, &a[0][0], control)
            ;
    } else {
        printf("Tried to initialize a gate with only
            control specifiers\n");
```

```
            exit(1);
        }
        mat_reduce(m*n, out);
    }
```

B.4 matgroup.c

```
#include "generator.h"
#include "matrix_quadratic.h"

#define MAT_DIM_A 2
#define MAT_DIM_B 3
#define MAT_DIM (MAT_DIM_A * MAT_DIM_B)
const size_t MAT_SIZE = (sizeof(num)*MAT_DIM*MAT_DIM);

void mat_mul_reduce(void *out, void *x, void *y) {
    mat_mul(MAT_DIM, out, x, y);
    mat_reduce(MAT_DIM, out);
}

void mat_mul_phase(void *out_v, void *x, void *y) {
    mat_mul(MAT_DIM, out_v, x, y);
    // think of this as a pointer to num[MAT_DIM][
       MAT_DIM]
    num (*out)[MAT_DIM] = out_v;
    size_t pi = 0, pj = 0;
    mat_reduce(MAT_DIM, out_v);
    while (num_eq(out[pi][pj], num_from_int(0))) {
        if (pi < MAT_DIM) {
            pi += 1;
        } else if (pj < MAT_DIM) {
            pj += 1;
            pi = 0;
        } else {
            break;
        }
    }
    num phase = out[pi][pj];
    if (!num_eq(phase, num_from_int(0))) {
        // normalise
        phase = num_mul(phase, num_inv_sqrt(num_mod_sq(
           phase)));
        // invert
```

```
        phase = num_reduce(num_inv(phase));
        range(i, MAT_DIM) {
            range(j, MAT_DIM) {
                out[i][j] = num_mul(out[i][j], phase);
            }
        }
        mat_reduce(MAT_DIM, out_v);
    }
}

void mat_print_default(void *x) {
    mat_print(MAT_DIM, x, ", ", "\n");
}

bool mat_is_diag(void *x_v) {
    num (*x)[MAT_DIM] = x_v;
    range(i, MAT_DIM) {
        range(j, MAT_DIM) {
            if (i != j && !num_eq(x[i][j], num_from_int
               (0))) {
                return false;
            }
        }
    }
    return true;
}

bool mat_print_order_ret_true(void *x) {
    int ord = mat_calc_order(MAT_DIM, x, true);
    if (ord == -1) {
        return true;
        //exit(0);
    } else {
        //printf("Order = %d\n", ord);
        return false;
    }
    return true;
}
```

```
void find_group() {
    char *gate_names[10] = {};
    num gates[10][MAT_DIM][MAT_DIM];
    int gate_count = 0;

    gate_names[gate_count] = "X2";
    initialize(MAT_DIM_A, MAT_DIM_B, &gates[gate_count
       ][0][0], GATE_SHIFT, GATE_IDENT);
    gate_count += 1;

    gate_names[gate_count] = "X3";
    initialize(MAT_DIM_A, MAT_DIM_B, &gates[gate_count
       ][0][0], GATE_IDENT, GATE_SHIFT);
    gate_count += 1;

    gate_names[gate_count] = "Z2";
    initialize(MAT_DIM_A, MAT_DIM_B, &gates[gate_count
       ][0][0], GATE_CLOCK, GATE_IDENT);
    gate_count += 1;

    gate_names[gate_count] = "Z3";
    initialize(MAT_DIM_A, MAT_DIM_B, &gates[gate_count
       ][0][0], GATE_IDENT, GATE_CLOCK);
    gate_count += 1;

    gate_names[gate_count] = "H2";
    initialize(MAT_DIM_A, MAT_DIM_B, &gates[gate_count
       ][0][0], GATE_FOURIER, GATE_IDENT);
    gate_count += 1;

    gate_names[gate_count] = "H3";
    initialize(MAT_DIM_A, MAT_DIM_B, &gates[gate_count
       ][0][0], GATE_IDENT, GATE_FOURIER);
    gate_count += 1;

    gate_names[gate_count] = "D2";
    initialize(MAT_DIM_A, MAT_DIM_B, &gates[gate_count
```

```
    ][0][0], GATE_CLIFF_DIAG, GATE_IDENT);
gate_count += 1;

gate_names[gate_count] = "D3";
initialize(MAT_DIM_A, MAT_DIM_B, &gates[gate_count
    ][0][0], GATE_IDENT, GATE_CLIFF_DIAG);
gate_count += 1;

gate_names[gate_count] = "C2X3";
initialize(MAT_DIM_A, MAT_DIM_B, &gates[gate_count
    ][0][0], GATE_CONTROL_1, GATE_SHIFT);
//gate_count += 1;

gate_names[gate_count] = "C3X2";
initialize(MAT_DIM_A, MAT_DIM_B, &gates[gate_count
    ][0][0], GATE_SHIFT, GATE_CONTROL_1);
//gate_count += 1;

gate_names[gate_count] = "C2S3";
initialize(MAT_DIM_A, MAT_DIM_B, &gates[gate_count
    ][0][0], GATE_CONTROL_1, GATE_TRANSPOSE);
gate_count += 1;

void *gen[gate_count];
range(i, gate_count) {
    gen[i] = gates[i];
    printf("%s:\n", gate_names[i]);
    mat_print(MAT_DIM, (num*)gates[i], ", ", "\n");
    printf("\n");
}

PathList paths = gen_paths(
    gate_count, gen, gate_names, MAT_SIZE,
    mat_mul_phase, mat_print_default,
        mat_print_order_ret_true,
       SEARCH_BREADTH_FIRST
);
printf("Total of %lu gates generated\n", paths.
```

```
        path_count);
}

int main() {
    find_group();
}
```

B.5 permutation.c

```
#include "generator.h"

typedef u8 *Map;
typedef u8 const *ConstMap;

void print_cycle_decomposition(ConstMap m, u8 num) {
    bool printed[num];
    for (u8 i = 0; i < num; i++) {
        printed[i] = false;
    }
    bool err = false;
    bool any_printed = false;
    for (u8 i = 0; i < num; i++) {
        if (m[i] == i || printed[i]) { continue; }
        any_printed = true;
        printf("(%u", i);
        printed[i] = true;
        u8 d = i;
        while (!printed[m[d]]) {
            d = m[d];
            if (num > 10) {
                printf(",");
            }
            printf("%u", d);
            printed[d] = true;
        }
        if (m[d] != i) {
            err = true;
            if (num > 10) {
                printf(",");
            }
            printf("%u", m[d]);
        }
        printf(")");
    }
    if (!any_printed) {
```

```
            printf("()");
        }
        if (err) {
            printf("WARNING non-cycle was generated\n");
        }
}

#define NUM 6

void compose_permutations(void *out_v, void *x_v, void *
    y_v) {
    u8 *out = out_v;
    u8 *x = x_v;
    u8 *y = y_v;
    for (size_t i = 0; i < NUM; i++) {
        out[i] = x[y[i]];
    }
}

void print_permutation(void *x) {
    print_cycle_decomposition(x, NUM);
}

void findgen() {
#define MAPS_LEN 8
    struct {
        char *name;
        u8 map[6];
        bool include;
    } maps[MAPS_LEN] = {
        {"cx3", {0,1,2,4,5,3}, false},
        {"cx2", {0,4,2,3,1,5}, false},
        {"x3", {1,2,0,4,5,3}, true},
        {"s3", {1,0,2,4,3,5}, false},
        {"x2", {3,4,5,0,1,2}, true},
        {"x2x3", {4,5,3,1,2,0}, false},
        {"g6", {1,2,3,4,5,0}, false},
        {"cs3", {0,1,2,4,3,5}, true},
```

```
    };
    size_t gen_len = 0;
    void* gen[MAPS_LEN];
    char* gen_names[MAPS_LEN];
    for (size_t i = 0; i < MAPS_LEN; i++) {
        if (maps[i].include) {
            gen[gen_len] = maps[i].map;
            gen_names[gen_len] = maps[i].name;
            gen_len++;
        }
    }
    PathList paths = gen_paths(gen_len, gen, gen_names,
      NUM, compose_permutations, print_permutation,
      NULL, SEARCH_BREADTH_FIRST);
    size_t sn = 1;
    for (size_t i = 1; i <= NUM; i++) {
        sn *= i;
    }
    printf("Hit %u out of %u permutations\n", paths.
      path_count, sn);
    printf("Max word length was %d\n", paths.paths[paths
      .path_count - 1].len.n);
}

void findcliffordish() {
    u8 shift[NUM][NUM];
    u8 x2x3[NUM] = {4,5,3,1,2,0};
    for (u8 x = 0; x < NUM; x++) {
        shift[0][x] = x;
    }
    for (u8 i = 1; i < NUM; i++) {
        for (u8 x = 0; x < NUM; x++) {
            shift[i][x] = x2x3[shift[i-1][x]];
        }
        printf("x2x3^%u = ", i);
        print_cycle_decomposition(shift[i], NUM);
        printf("\n");
    }
```

```
u8 m[NUM] = {};
bool done = false;
while (!done) {
    u8 minv[NUM] = {255, 255, 255, 255, 255, 255};
    for (u8 x = 0; x < NUM; x++) {
        minv[m[x]] = x;
    }
    bool invertible = true;
    for (u8 y = 0; y < NUM; y++) {
        if (minv[y] == 255) {
            invertible = false;
        }
    }
    if (invertible) {
        u8 result[NUM];
        for (u8 x = 0; x < NUM; x++) {
            result[x] = minv[x2x3[m[x]]];
        }
        for (u8 i = 0; i < NUM; i++) {
            bool match = true;
            for (u8 x = 0; x < NUM; x++) {
                if (result[x] != shift[i][x]) {
                    match = false;
                }
            }
            if (match) {
                print_cycle_decomposition(minv, NUM)
                    ;
                printf(" x2x3 ");
                print_cycle_decomposition(m, NUM);
                printf(" = x2x3^%u\n", i);
            }
        }
    }

    for (u8 x = 0; x < NUM; x++) {
        m[x] += 1;
        if (m[x] < NUM) { break; }
```

```
            m[x] = 0;
            if (x+1 == NUM) { done = true; }
        }
    }
}

int main() {
    findgen();
    //findcliffordish();
}
```

B.6 matorder.c

```
#include "matrix_quadratic.h"
#include <stdint.h>

typedef uint8_t u8;

#define MAT_DIM 6

void test_perm_fourier_order() {
    num h2[MAT_DIM][MAT_DIM];
    initialize(2, 3, &h2[0][0], GATE_FOURIER, GATE_IDENT
        );

    num h3[MAT_DIM][MAT_DIM];
    initialize(2, 3, &h3[0][0], GATE_IDENT, GATE_FOURIER
        );

    u8 m[MAT_DIM] = {};
    int total_ph2 = 0;
    int total_ph3 = 0;
    int total_ph23 = 0;
    bool done = false;
    while (!done) {
        bool invertible = true;
        range(i, MAT_DIM) {
            range(j, i) {
                if (m[i] == m[j]) {
                    invertible = false;
                }
            }
        }
        if (invertible) {
            num m_mat[MAT_DIM][MAT_DIM];
            mat_perm(MAT_DIM, &m_mat[0][0], m);
            num result[MAT_DIM][MAT_DIM];

            mat_mul(MAT_DIM, &result[0][0], &m_mat
```

```
    [0][0], &h2[0][0]);
int order_ph2 = mat_calc_order(MAT_DIM, &
    result[0][0], false);
mat_mul(MAT_DIM, &result[0][0], &h2[0][0], &
    m_mat[0][0]);
int order_h2p = mat_calc_order(MAT_DIM, &
    result[0][0], false);
if (order_ph2 != -1 && order_h2p != -1) {
    /*
    printf("P H2 has order %d, and H2 P has
        order %d, where P:\n", order_ph2,
        order_h2p);
    mat_print(MAT_DIM, &m_mat[0][0], "", "",
        "\n");
    printf("\n");
    */
    total_ph2 += 1;
}

mat_mul(MAT_DIM, &result[0][0], &m_mat
    [0][0], &h3[0][0]);
int order_ph3 = mat_calc_order(MAT_DIM, &
    result[0][0], false);
mat_mul(MAT_DIM, &result[0][0], &h3[0][0], &
    m_mat[0][0]);
int order_h3p = mat_calc_order(MAT_DIM, &
    result[0][0], false);
if (order_ph3 != -1 && order_h3p != -1) {
    /*
    printf("P H3 has order %d, and H3 P has
        order %d, where P:\n", order_ph3,
        order_h3p);
    mat_print(MAT_DIM, &m_mat[0][0], "", "",
        "\n");
    printf("\n");
    */
    total_ph3 += 1;
    if (order_ph2 != -1 && order_h2p != -1)
```

```
                    {
                        printf("ord(P H2) = %d, ord(H2 P) =
                            %d, ord(P H3) = %d, prd(H3 P) = %
                            d\n",
                            order_ph2, order_h2p, order_ph3,
                                order_h3p);
                        mat_print(MAT_DIM, &m_mat[0][0], "
                            ", "\n");
                        printf("\n");
                        total_ph23 += 1;
                    }
                }
            }

            for (u8 x = 0; x < MAT_DIM; x++) {
                m[x] += 1;
                if (m[x] < MAT_DIM) { break; }
                m[x] = 0;
                if (x+1 == MAT_DIM) { done = true; }
            }
        }
        printf("P H2 and H2 P were both finite order for %d
            distinct permutations P\n\n", total_ph2);
        printf("P H3 and H3 P were both finite order for %d
            distinct permutations P\n\n", total_ph3);
        printf("All four were finite order for %d distinct
            permutations P\n", total_ph23);
}

int main() {
    test_perm_fourier_order();
}
```

BIBLIOGRAPHY

[1] Quantum algorithm implementations for beginners, 2020.

[2] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John Smolin, and Harald Weinfurter. Elementary gates for quantum computation, 1995.

[3] Alex Bocharov, Shawn X. Cui, Martin Roeteller, and Krysta M. Svore. Improved quantum ternary arithmetics, 2016.

[4] Alex Bocharov, Xingshan Cui, Vadym Kliuchnikov, and Zhenghan Wang. Efficient topological compilation for weakly-integral anyon model, 2016.

[5] Alex Bocharov, Martin Roeteller, and Krysta M. Svore. Factoring with qutrits: Shor's algorithm on ternary and metaplectic quantum architectures, 2017.

[6] P. Oscar Boykin, Tal Mor, Matthew Pulver, Vwani Roychowdhury, and Farrokh Vatan. On universal and fault-tolerant quantum computing, 1999.

[7] Pedro Celis. Robin hood hashing, 1986.

[8] Shawn X. Cui and Zhenghan Wang. Universal quantum computation with metaplectic anyons, 2015.

[9] M. Korbelar and J. Tolar. Symmetries of the finite heisenberg group for composite systems, 2010.

[10] Ashok Muthukrishnan and C. R. Stroud Jr. Multi-valued logic gates for quantum computation, 2018.

[11] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. 2010.

[12] Peter W. Shor. Fault-tolerant quantum computation, 1997.

[13] Andrew Steane. Multiple particle interference and quantum error correction, 1995.

[14] J. Tolar. On clifford groups in quantum computing, 2018.