

Portland State University

**PDXScholar**

---

Dissertations and Theses

Dissertations and Theses

---

1-1-2009

# An Eigenanalysis and Synthesis of Unitary Operators used in Quantum Computing Algorithms

Steven Randall Hutsell  
*Portland State University*

Follow this and additional works at: [https://pdxscholar.library.pdx.edu/open\\_access\\_etds](https://pdxscholar.library.pdx.edu/open_access_etds)

**Let us know how access to this document benefits you.**

---

## Recommended Citation

Hutsell, Steven Randall, "An Eigenanalysis and Synthesis of Unitary Operators used in Quantum Computing Algorithms" (2009). *Dissertations and Theses*. Paper 251.  
<https://doi.org/10.15760/etd.251>

This Dissertation is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: [pdxscholar@pdx.edu](mailto:pdxscholar@pdx.edu).

An Eigenanalysis and Synthesis  
of Unitary Operators used in Quantum  
Computing Algorithms

by

Steven Randall Hutsell

A dissertation submitted in partial fulfillment of the  
requirements for the degree of

Doctor of Philosophy  
in  
Electrical and Computer Engineering

Dissertation Committee:  
Garrison Greenwood, Chair  
Marek Perkowski  
Steven Bleiler  
Richard Tymerski  
Peter Leung

Portland State University  
2009

## Abstract

In this work we tackle the challenge of designing quantum unitary operators which represent solutions to optimization problems. We start with a novel method which combines an evolutionary algorithm known as an Evolution Strategy (ES) with a method to randomly generate unitary operators. With this new method, a quantum operator is represented for the first time using real-valued vectors and can be “evolved” or designed to meet certain target criteria. This criteria could be the solution to an optimization problem. With the ability to evolve quantum operators, we attempt to evolve various known single and multi-qubit quantum gates as well as quantum oracles. We evolve quantum operators which solve instance problems of a known NP-Hard problem and even attempt to evolve a generalized solution operator. We evolve multiple operators with varying size and investigate their properties through eigenanalysis methods as well as by synthesizing them into quantum logic gates using the quantum compiler Qubiter. We also present a new quantum logic algebra which offers a new way to represent quantum circuits and demonstrate its immediate uses in quantum computing.

To my wife Joann, my son Ethan, and my daughter Brooklyn.

In memory of Gary W. Hutsell and Edmundo L. Zevallos.

## Acknowledgments

Since deciding to pursue my doctoral degree, my wife and I have had two children and together we have lost two fathers. So, first and foremost, I'd like to thank my wife and children for being so very patient with me and showing such loving support. I thank my parents for their encouragement and for showing me the importance of education as well as supporting me financially as much as they could through my early college years. I have no doubt in my mind that my program would not have been successful without the mentorship and support of my advisor, Dr. Garry Greenwood. He provided me immeasurable help and some necessary prodding over the years which I greatly appreciated. It has been an honor working with Dr. Marek Perkowski who has been a rich source of knowledge and a key member of my committee. Both he and his students helped me gain a better understanding of quantum computing. I would like to thank Dr. George Lendaris who was willing to take me on as a research assistant while I was still an undergraduate. It was with his encouragement that I decided to pursue my doctoral degree and some of my fondest memories at PSU were of working with him at the Systems Science department. It was a pleasure working with Dr. Lee Casperson who was my original doctoral advisor

before leaving to North Carolina. He guided me through the initial path of my Ph.D program as well as overseeing my comprehensive examination. I'd especially like to thank Dr. Doug Hall who was not only a good mentor but, a long time ago, convinced me to enter the Electrical Engineering program at PSU. Lastly, I'd like to thank the professors on my doctoral committee for their support and their interest in my research.

## Contents

<b>Abstract</b>	<b>i</b>
<b>Dedication</b>	<b>ii</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>List of Tables</b>	<b>x</b>
<b>List of Figures</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem Statement . . . . .	3
1.2 Research Overview . . . . .	5
1.3 Primary Contributions . . . . .	7
1.4 Dissertation Organization . . . . .	11
<b>2 Background</b>	<b>13</b>
2.1 Quantum Mechanics . . . . .	13
2.1.1 A Quantum of History . . . . .	13
2.1.2 Dirac's Notation . . . . .	15
2.1.3 Hilbert Space . . . . .	16
2.1.4 Tensor and Kronecker Products . . . . .	17

2.1.5	Quantum States . . . . .	19
2.1.6	Quantum Operators . . . . .	20
2.1.7	Quantum Observables and Measurements . . . . .	22
2.1.8	Spectral Decomposition . . . . .	25
2.2	Quantum Computing . . . . .	27
2.2.1	Qubits . . . . .	28
2.2.2	Quantum Gates . . . . .	32
2.2.3	Quantum Circuits . . . . .	49
2.2.4	Quantum Algorithms . . . . .	56
2.3	Evolutionary Algorithms . . . . .	77
2.3.1	Elements of an Evolutionary Algorithm . . . . .	79
2.3.2	Evolution Strategies . . . . .	86
<b>3</b>	<b>Related Research</b>	<b>93</b>
3.1	Evolutionary Algorithms in	
	Quantum Computing . . . . .	93
3.1.1	Application of a Genetic Algorithm for	
	Quantum Logic Synthesis . . . . .	94
3.2	Random Unitary Matrices . . . . .	100
3.3	Eigenanalysis Methods and Separability . . . . .	101
3.4	Quantum Logic Synthesis . . . . .	104
3.4.1	Quantum Gate Universality . . . . .	104
3.4.2	Computation Bounding . . . . .	105
3.4.3	Uniformly Controlled Gate Decomposition . . . . .	105
3.4.4	Cosine–Sine Decomposition . . . . .	106
3.4.5	Quantum Compilers . . . . .	109



<b>4</b>	<b>Quantum Logic Algebra</b>	<b>111</b>
4.1	Introduction . . . . .	111
4.2	Key Definitions . . . . .	114
4.3	The Hutsell Method . . . . .	116
4.4	Slice Examples . . . . .	119
4.5	Canonical Construction . . . . .	119
4.6	Creating the Method . . . . .	131
4.7	Examples . . . . .	134
4.8	Extension to Multi-valued Logic . . . . .	138
<b>5</b>	<b>Evolving Unitary Operators</b>	<b>140</b>
5.1	Constructing the ES . . . . .	142
5.1.1	Representation of an Individual . . . . .	143
5.2	Optimizing the ES Parameters . . . . .	150
5.2.1	Parent and Offspring Size Selection . . . . .	151
5.2.2	Mutation and Recombination . . . . .	154
5.2.3	Using a Complex versus Probabilistic Target Vector . . . . .	159
5.2.4	Initial and Reset $\sigma$ Value . . . . .	160
5.2.5	Fitness Calculation Methods . . . . .	162
5.2.6	Training Behavior . . . . .	168
5.2.7	Best Convergence Results . . . . .	171
5.2.8	Simulation Run-Time . . . . .	172
5.3	Evolving Known Quantum Gates . . . . .	175
5.3.1	Evolving the Hadamard Gate . . . . .	179
5.3.2	Evolving Known Single Qubit Gates . . . . .	180
5.3.3	2-qubit Oracle for Deutsch's Problem . . . . .	184

5.4	Single Problem Instances . . . . .	186
5.4.1	3-Node ISP Instance . . . . .	189
5.4.2	4-Node ISP Instance . . . . .	196
5.4.3	Single Instance Conclusions . . . . .	202
5.5	General Problem Solving . . . . .	202
5.5.1	Encoding General Graphs and ISP Solutions . . . . .	203
5.5.2	ES Input Methodology . . . . .	206
5.5.3	Tuning the ES Parameters for General Problems . . . . .	207
5.5.4	Test and Fitness for the Generalized 3 Node ISP . . . . .	211
<b>6</b>	<b>Eigenanalysis</b>	<b>214</b>
6.1	Eigenvalue Placement . . . . .	215
6.2	Degeneracy . . . . .	216
6.3	Interpretation of Eigenvalues . . . . .	224
6.4	Adjusting Eigenvalues . . . . .	231
<b>7</b>	<b>Logic Synthesis Results</b>	<b>241</b>
7.1	Compiling and Running Qubiter . . . . .	242
7.2	Qubiter Input Files . . . . .	243
7.3	Qubiter Output Files . . . . .	245
7.4	Synthesis Results . . . . .	249
7.5	Input, Loaded, and Result Matrix Errors . . . . .	257
7.6	Synthesis Runtimes, Limitations, and Memory Usage . . . . .	261
7.7	Synthesis Conclusions . . . . .	261

<b>8</b>	<b>Summary and Major Findings</b>	<b>263</b>
<b>9</b>	<b>Future Work</b>	<b>266</b>
9.1	Further Evolving Unitary Matrices . . . . .	266
9.2	Applications of our Quantum Logic Algebra . . . . .	271
9.3	Expanding The Eigenanalysis . . . . .	277
9.4	Better Logic Synthesis . . . . .	280
	<b>References</b>	<b>282</b>
	<b>Appendices</b>	<b>292</b>
<b>A</b>	<b>Compiling and Running the ES</b>	<b>292</b>
A.1	Compiling the ES . . . . .	293
A.2	Running the ES . . . . .	293
A.3	The ES File Format . . . . .	294
A.4	The ES Output . . . . .	296
<b>B</b>	<b>ES Source Code</b>	<b>300</b>
B.1	The Main ES Code: ES.CPP . . . . .	300
B.2	The Header Files . . . . .	325
B.2.1	MATRIX.H . . . . .	325
B.2.2	UNITARY.H . . . . .	347
B.2.3	PARSE_ARGS.H . . . . .	350
B.2.4	RANDOM.H . . . . .	352

## List of Tables

2.1	Dirac's Notation . . . . .	15
4.1	Substitution table for $\#$ symbols. Substitutions are made from left to right in each term. . . . .	118
4.2	Encoding table to simplify $P$ . . . . .	126
5.1	Average, Min, and Max Run-Times (in seconds) for $\mu = 10$ , $\lambda = 500$ ES Simulations . . . . .	178
5.2	Summary Table of 4-Node ISP Single Instance Training Results.	198
5.3	Number of qubits required to encode an $n$ node graph for general ISP solutions. . . . .	205
5.4	Input state encoding for 3-node ISP graphs. . . . .	206
5.5	3-Node ISP graph encoding with single and multiple solution state encodings. . . . .	207
5.6	Comparison of target solution for 3 Node ISP Single Solution. .	211
5.7	Comparison of target solution for 3 Node ISP Single Solution. .	212
7.1	Table of Qubiter output files. . . . .	246
7.2	Average total synthesized gates versus our predicted total gates.	251

## List of Figures

1.1	Evolving $U$ to solve for a known output vector . . . . .	6
2.1	Bloch Sphere representation of a Qubit . . . . .	31
2.2	CNOT gate drawn using an $X$ gate . . . . .	39
2.3	Common drawing of a CNOT using XOR symbol . . . . .	39
2.4	General gate operating on qubit 1 and controlled by qubit 0 . . .	41
2.5	General two qubit gate . . . . .	41
2.6	Toffoli Gate . . . . .	42
2.7	Fredkin Gate . . . . .	44
2.8	Example Three Qubit Quantum Circuit . . . . .	50
2.9	Common Single Qubit Circuit Identities . . . . .	54
2.10	Controlled- $Z$ Identity . . . . .	54
2.11	Common Two-Qubit Circuit Identities . . . . .	57
2.12	Simple two-input reversible quantum operator, $U_f$ . . . . .	58
2.13	Quantum Circuit for Deutsch's Algorithm . . . . .	60
2.14	Quantum Circuit for the Deutsch-Jozsa Algorithm . . . . .	62
2.15	Quantum Fourier Transform Circuit . . . . .	65
2.16	Structure of a Quantum Oracle . . . . .	71
2.17	The Grover Iterate: HZHO . . . . .	73
2.18	Zero State Phase Shift Circuit . . . . .	74

2.19	Grover's Algorithm . . . . .	76
2.20	Grover Iterations rotate an initial solution to the desired solution	77
2.21	The canonical EA . . . . .	78
3.1	Example GA Encoding for 5 Qubit Individual . . . . .	96
3.2	Quantum Logic Synthesis of a 3 qubit system . . . . .	104
3.3	Uniformly Controlled Gate Decomposition . . . . .	106
3.4	Quantum Multiplexor selecting between block operators from CSD110	
4.1	Parallel Logic Example: $S_1 = X \otimes I \otimes Z$ and $S_2 = Y \otimes I$ , thus $U = S_2 \times S_1$ . . . . .	112
4.2	Example of Difficult Logic Slices: $U = S_4 \times S_3 \times S_2 \times S_1$ . . . .	113
4.3	Generalized control gate . . . . .	113
4.4	Splitting Slices: $S_1 = S_{1b} \times S_{1a}$ . Alternatively, we could have done $S_1 = S_{1a} \times S_{1b}$ since $S_{1a}$ and $S_{1b}$ are independent . . . . .	117
4.5	Method Examples . . . . .	119
4.6	Refined Slices: $S_1 = S_{1c} \times S_{1b} \times S_{1a}$ . $S_{1a}$ , $S_{1b}$ , and $S_{1c}$ are in- dependent. $S_{1a} = ID_0II + XD_1II$ , $S_{1b} = ID_0II + ID_1YI$ , and $S_{1c} = IIIZ$ . $S_1 = ID_0IZ + XD_1YZ$ . . . . .	123
4.7	Canonical Representation Example: $S = P + P_c \times C = (I - P_c) +$ $P_c \times C$ ( <i>shown</i> ) . . . . .	129
4.8	Method legal slice versus canonical slice: $S = S_2 \times S_1 = S_1 \times S_2$	132
4.9	The quantum circuit for Example 1. . . . .	134
4.10	The quantum circuit for Example 2. The open circle is the neg- ative control. . . . .	136
4.11	The quantum circuit for Example 3. . . . .	136

4.12 Quantum circuit for Example 4. . . . .	138
4.13 Generalized two-qudit controlled gate example . . . . .	139
5.1 ES Flow Chart . . . . .	144
5.2 $\mu$ vs. $\lambda$ , 4 Qubits, $(\mu + \lambda)$ , Probabilistic, 70% Target . . . . .	152
5.3 $\mu$ vs. $\lambda$ , 4 Qubits, $(\mu + \lambda)$ , Probabilistic, 99% Target . . . . .	152
5.4 $\mu$ vs. $\lambda$ , 5 Qubits, $(\mu + \lambda)$ , Probabilistic, 70% Target . . . . .	152
5.5 $\mu$ vs. $\lambda$ , 4 Qubits, Average over all runs . . . . .	153
5.6 $\mu$ vs. $\lambda$ , 5 Qubits, Average over all runs . . . . .	153
5.7 Comparison of $(\mu, \lambda)$ to $(\mu + \lambda)$ for 3, 4, and 5 qubits . . . . .	154
5.8 5 Qubit Mutation/Recombination Sweep at 70% Target . . . . .	156
5.9 4 Qubit Mutation/Recombination Sweep at 70% Target . . . . .	156
5.10 4 Qubit Mutation/Recombination Sweep at 90% Target . . . . .	157
5.11 5 qubit recombination versus 100% mutation . . . . .	157
5.12 4 qubit recombination versus 100% mutation . . . . .	157
5.13 3 qubit recombination versus 100% mutation . . . . .	157
5.14 Best case 50% Mutation for 70% Target . . . . .	158
5.15 Best case 50% Mutation for 90% Target . . . . .	159
5.16 Probability vs. Complex Target for 3/4/5 Qubit (10+500) ES . . . . .	160
5.17 Initial/Reset $\sigma$ Sensitivity for a 2-5 Qubit (10+500) ES . . . . .	161
5.18 1 Qubit Fitness Landscape plot for Fitness Equation 5.0.3 . . . . .	164
5.19 1 Qubit Fitness Landscape plot for Fitness Equation 5.2.4 . . . . .	165
5.20 Fitness comparison for 4 Qubit, Probabilistic, 70% Target ES . . . . .	166
5.21 Fitness comparison for 4 Qubit, Probabilistic, 99% Target ES . . . . .	167
5.22 Fitness comparison for 5 Qubit, Probabilistic, 70% Target ES . . . . .	167
5.23 Fitness comparison for 5 Qubit, Probabilistic, 70% Target ES . . . . .	168

5.24	Best Individual per Generation, 5 Qubits, 70% Target . . . . .	169
5.25	Best Individual per Generation, 4 Qubits, 70% Target . . . . .	170
5.26	Best Individual per Generation, 3 Qubits, 70% Target . . . . .	171
5.27	Best Individual per Generation, 2 Qubits, 70% Target . . . . .	172
5.28	Best Individual per Generation, 5 Qubits, 99% Target . . . . .	173
5.29	Best Individual per Generation, 4 Qubits, 99% Target . . . . .	174
5.30	Best Individual per Generation, 3 Qubits, 99% Target . . . . .	175
5.31	Best Individual per Generation, 2 Qubits, 99% Target . . . . .	176
5.32	Best Fitness Convergence for 70% and 99% Targets . . . . .	176
5.33	Average, Min, and Max Run–Times . . . . .	177
5.34	General quantum circuit for Deutsch’s problem (2 qubits). The blocks labeled with an $H$ are Hadamard gates while the block labeled with $U$ is the unitary operator representing the oracle to be evolved. . . . .	185
5.35	An Independent Set Problem instance. Both $V'_1 = \{2, 4, 7, 9\}$ and $V'_2 = \{1, 6, 9\}$ are independent sets, but neither one is globally optimum. . . . .	187
5.36	An ISP instance of three nodes. . . . .	187
5.37	3–Node ISP with multiple solutions. . . . .	189
5.38	Eigenvalue placement for 3–Node ISP with single solution $V_1$ . . .	194
5.39	Eigenvalue placement for 3–Node ISP with single solution $V_2$ . . .	194
5.40	Eigenvalue placement for 3–Node ISP with both solutions $V_1$ and $V_2$ . . . . .	195
5.41	Comparison of 3–Node ISP $V_1$ (X’s) superimposed with $V_2$ (O’s) against $V_1 + V_2$ eigenvalue plots. . . . .	195



5.42	4-Node ISP with multiple solutions. . . . .	196
5.43	4-Node ISP Single-Solution Instance Eigenvalue placements. . .	199
5.44	4-Node ISP Multi-Solution Instance Eigenvalue placements. . .	200
5.45	4-Node ISP instance eigenvalue placement comparison. $V_1$ (X's), $V_2$ (O's), $V_3$ (Squares) . . . . .	201
5.46	Encoding a 4-node graph for a general ISP. . . . .	206
5.47	3-Node General ISP Fitness given $\mu$ . . . . .	208
5.48	3-Node General ISP Fitness given $\lambda$ . . . . .	209
5.49	3-Node General ISP Fitness given a reset/initial $\sigma$ . . . . .	210
5.50	3-Node General ISP Fitness for Probability/Complex and $(\mu + \lambda)$ vs. $(\mu, \lambda)$ mode. . . . .	210
6.1	Eigenvalue plots for Complex Target, 70% Target Fitness . . . .	217
6.2	Eigenvalue plots for Probabilistic Target, 70% Target Fitness . .	218
6.3	Eigenvalue plots for Complex Target, 99% Target Fitness . . . .	219
6.4	Eigenvalue plots for Probabilistic Target, 99% Target Fitness . .	220
6.5	Best 5 qubit, probability mode, 70% target, pairs close in value to be made degenerate are circled in blue. . . . .	235
6.6	Adjusted 5-qubit operator eigenvalue placement. Squares indi- cate our newly formed degenerate pairs. . . . .	239
6.7	5-qubit probability distribution of original operator (blue) versus one with adjusted eigenvalues to add degeneracy (red). . . . .	240
7.1	Average Number of Synthesized Gates . . . . .	250
7.2	Breakdown of Gates used in Synthesis . . . . .	251
7.3	1 qubit, complex mode, 70% target synthesized circuit. . . . .	253

7.4	1 qubit, complex mode, 99% target synthesized circuit. . . . .	253
7.5	1 qubit, probability mode, 70% target synthesized circuit. . . . .	253
7.6	1 qubit, probability mode, 99% target synthesized circuit. . . . .	253
7.7	2 qubit, complex mode, 70% target synthesized circuit. . . . .	254
7.8	5 qubit, complex mode, 99% target synthesized circuit. . . . .	254
7.9	2 qubit, probability mode, 70% target synthesized circuit. . . . .	254
7.10	2 qubit, probability mode, 99% target synthesized circuit. . . . .	254
7.11	3 qubit, complex mode, 70% target synthesized circuit. . . . .	255
7.12	3 qubit, complex mode, 99% target synthesized circuit. . . . .	255
7.13	3 qubit, probability mode, 70% target synthesized circuit. . . . .	255
7.14	3 qubit, probability mode, 99% target synthesized circuit. . . . .	255
7.15	4 qubit, complex mode, 70% target synthesized circuit. . . . .	256
7.16	4 qubit, complex mode, 99% target synthesized circuit. . . . .	256
7.17	4 qubit, probability mode, 70% target synthesized circuit. . . . .	256
7.18	4 qubit, probability mode, 99% target synthesized circuit. . . . .	256
7.19	5 qubit, complex mode, 70% target synthesized circuit. . . . .	257
7.20	5 qubit, probability mode, 70% target synthesized circuit. . . . .	257
7.21	Average Error between the Evolved $U$ and the Loaded $U$ . . . . .	259
7.22	Average Error between the Loaded $U$ and the Synthesized $U$ . . . . .	260
9.1	Simple Random Unitary Model Example. . . . .	267
9.2	Reduced Simple Random Unitary Model. $E_2$ appeared not to be needed as $q_1$ always remained in the same state. . . . .	268
9.3	Barenco et al's decomposition of $\wedge_2(U)$ where $V^2 = U$ . . . . .	271
9.4	Reconstructed circuit from a quantum algebraic expression. . . . .	274
9.5	General circuit showing $F$ following by a CNOT. . . . .	275

9.6	General circuit where the function of the upper circuit is separated by from the function of the lower qubit, followed by a CNOT. . . . .	276
-----	---	-----

## **Chapter 1**

### **Introduction**

The real challenge in solving optimization problems is to create algorithms and techniques that can solve realistically sized problems within a reasonable amount of computational time. Most of these algorithms formulate an optimization problem as a search problem (i.e., the problem solutions reside in an abstract solution space and two solutions are neighbors if they differ by a small perturbation of a problem parameter). Any algorithm that “solves” an optimization problem is therefore a search algorithm that explores the solution space landscape.

Unfortunately, many real-world optimization problems require such huge computational resources that brute force search methods are useless; they simply take too much time to find the optimal solution. This has led researchers to use search heuristics that yield an acceptable compromise: a possibly lower quality answer but with minimal search effort. Recently an entirely new approach has surfaced with potentially enormous consequences. This new approach is called quantum computing and it relies on the principles of quantum mechanics to find problem solutions.

We are interested in solving optimization problems which have their solutions encoded as binary strings. This covers a broad class of problems including

many of which are NP-hard. In principle, a classical computer takes an initial solution binary string and, using logic operations, transforms it into the final solution binary string. The specific logical operations are dictated by the search algorithm steps. Since any logical operation can be implemented with logic gates, one could physically implement the search algorithm as a logic circuit composed of interconnected elementary logic gates.

This classical system perspective has been adopted by many developers of quantum computing search algorithms. Quantum mechanical systems evolve according to Schrödinger’s equation (i.e., the initial system state is transformed into a final state by a series of unitary operations). Since problem solutions are encoded in quantum computers as a set of qubits, these unitary operators are usually defined as elementary quantum “gates” (e.g., a controlled–NOT gate). Although different optimization problems may use qubits to represent solutions, each optimization problem instance requires an entirely new “quantum circuit”. This is because the qubit states that represent the optimal solution to one type of optimization problem will mostly not be the same for the optimal solution to a different optimization problem. Consequently, a new quantum circuit is required for each optimization problem, which makes it imperative that an efficient quantum-gate synthesizer be available.

The construction of practical quantum computers depends on the availability of quantum circuits because it is only through these circuits can experimenters develop and demonstrate principles at the quantum level. Quantum circuit synthesizers are therefore vital and their design continues to be a hot research area [1, 2]. One of the key metrics used to evaluate these synthesizers is the number of quantum gates they require to implement a unitary operation. Un-

fortunately, few synthesizers are publicly available. These unitary operators will directly manipulate state amplitudes. Specifically, each distinct state represents a unique problem solution and our goal is therefore to amplify the amplitude of the one state that encodes the globally optimum solution to an optimization problem while attenuating the amplitudes of all other states.

## 1.1 Problem Statement

**Can we develop a good method to generate quantum unitary operators?**

The ability to generate the unitary matrix describing a quantum computer is a large challenge. Typically, the matrix is given *a priori* and generated by hand or constructed through specific applications of known elementary quantum gates or smaller quantum circuits, or even from an equivalent classical logic circuit. However, these methods have traditionally focused on very specific problems. The ability to easily design the unitary operator of a quantum computer is a significant weakness in quantum computing research.

**Can we evolve quantum sub-circuits and new elementary quantum gates?**

Many quantum algorithms have similar structures and only a sub-circuit or one or more elementary gates may need to be re-designed to implement the solution. The design of a quantum sub-circuit suffers the same limitations as general operators, that is, there is no good general design technique. Can a generalized method be used to design the quantum sub-circuit or discover (or even re-discover) elementary quantum gates?

## **Can quantum operators evolved for particular solutions tell us more about how to design better quantum operators?**

The quantum computing research community is constantly searching for problems which can be solved with a quantum computer more efficiently than its classical counterpart. Some instances have been found as we detail in this dissertation. The formulation of a quantum algorithm is complicated and we're starting to gain a better understanding how to re-use those formulations to potentially evolve newer quantum algorithms. If we had a method of evolving a quantum operator which solved for various instance problems, could an analysis of these operators give us clues on the general design of quantum operators?

In general, there are no clear rules for the design of the quantum unitary operator. For specific problems, certain criteria can sometimes be deduced. But what we're really after is a generalized approach. Could an eigenanalysis of a large selection of evolved quantum operators which solve instance problems give us insight into the design of general quantum operators?

## **Can we evolve generalized quantum operators which can solve problems based on input criteria?**

Most quantum algorithms are instance problems. They generally rely on a given oracle (or some other determined circuit) which has the solution criteria pre-programmed. The input to the quantum computer is typically static (that is, it's a known state). By having a generalized method, could we design a quantum operator which would yield a desired target for various input states?

## **Is there a quantum algebraic logic like boolean logic for classical logic?**

Lastly, another severely weak aspect in quantum computing is a fundamental algebraic representation of quantum circuits. For example, what is the classical counterpart of boolean logic in quantum computing? Typically, quantum circuits are expressed using more mathematically intensive representations and are almost invariably at the state-transition level. The culprit has traditionally been controlled gates which no longer allow us to represent a slice of a quantum circuit as simple Kronecker product of its gates and wires. A lack of a quantum logic algebra has been a hindrance in many ways. It increases the complexity of the mathematical description of a quantum circuit by representing the output state of the circuit based on an initial state rather than the transfer function (that is, the unitary operator itself). Such representations are complicated and operating on them is typically a mathematical exercise and not well suited for software implementations. Secondly, it prevents us from developing a truly diverse set of operations that would allow us to simplify quantum circuits, map them from one structure to another, and derive characteristics using known mathematical rules and laws. Is there a quantum logic algebra?

## **1.2 Research Overview**

We used evolutionary algorithms—i.e., algorithms that conduct searches using the principles of Darwinian evolution found in Nature—to design unitary operators as used in quantum computing. Our first objective was to design the evolutionary algorithm (EA) that would — given input and target state conditions — evolve one or more candidate unitary operators which satisfy the instance solutions within a certain error margin. The design of the EA is an



integral and fundamental aspect of this research. One of the fundamental aspects of our research was to determine a method of randomly generating unitary matrices based on a set of parameters. Rigorous exploration was made on the various parameters of the EA to determine the best configuration for operating on quantum operator problems of varying complexity.

We evolved many quantum operators which satisfactorily solved problem instances. The problem had a known input which was, in fact, an equal superposition of all states (some call this *the superposition* state) and a known target. Figure 1.1 shows an example where we want to evolve  $U$  such that when multiplied by the input state (a superposition vector), the result is the target vector within a certain error margin.

$$\begin{pmatrix} u_{00} & u_{01} & u_{02} & \cdots \\ u_{10} & u_{11} & u_{12} & \cdots \\ u_{20} & u_{22} & u_{22} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \frac{1}{\sqrt{N}} \begin{pmatrix} 1 \\ 1 \\ 1 \\ \vdots \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \end{pmatrix}$$

Figure 1.1: Evolving  $U$  to solve for a known output vector

The instance problems are general and could — for example — represent a solution to an NP-hard problem. An eigenanalysis of the unitary operators was conducted which primarily focused on eigenvalue placement. The resultant unitary operators were synthesized into quantum gates using an open-source quantum synthesizer (Qubiter[3]) and analyzed.

We applied the EA to various fundamental tests. For instance, we tested its ability to evolve known quantum gates (both single and multiple qubit gates). This was in an effort to show both the strengths and weakness of the EA.

Another application of the EA to design quantum sub-circuits was then showcased by using it to design the quantum oracle for Deutsch's Problem.

We chose optimization test problems (and in particular, instances of the Independent Set Problem (ISP)) small enough so that their optimal solutions could be found via exhaustive (classical) search. We applied our EA to evolve multiple quantum operators which represented solutions to ISP instance problems. By knowing the ISP solution, it was possible to specify what the relative amplitudes of all states should be for the solution. This allowed us to easily determine if the unitary operators were correctly constructed. A similar eigenanalysis study and synthesis was conducted.

The EA was then used to evolve generalized solutions to ISP problems. The size was kept small for convergence reasons and to have more sensible encodings for the initial state (for larger ISP problems, the number of graph configurations becomes much greater than the number of solutions).

A discovery was made during the course of our research which was not part of the original proposal. While trying to develop a software method to read in quantum netlists, we invented a method to represent quantum circuits (and especially those with controlled-gates) using a simple quantum logic algebra. This new quantum logic algebra and its applications are described in detail in chapter 4.

### **1.3 Primary Contributions**

There was a lot of data and analysis produced from our research and great care was made to pick out the most important parts to be included in this dissertation. We would like to highlight the primary contributions which resulted in publications:

1. An Evolution Algorithm (EA) which can generate quantum unitary oper-

ators using real-valued vectors[4].

2. A method for representing and manipulating quantum circuits using our invented Quantum Logic Algebra[5].

We now review the primary contributions of original work found in this research.

### **The Evolutionary Algorithm**

The EA was published in [4] and covered in chapter 5. Its significant contribution was not only that it presented a method to automatically evolve quantum unitary operators, but also that it represented the quantum operators using real-valued vectors. It was also the first to apply a class of evolutionary algorithms known as an Evolution Strategy (ES) to quantum computing. This was made possible through a novel application of applying a known method for generating random unitary matrices[6] with our EA.

With this method, generalized unitary operators can be created at both the top-circuit level as well as the sub-circuit and elementary gate levels. There were two immediate highlights that came from this research using our EA. When we applied the EA to evolve known quantum gates, we found that the anti-symmetric limitation (described in section 5.3) of the method prevented it from hitting the same complex behavior of some gates. This was also true of the Hadamard gate. However, upon further inspection of the evolved operator for the Hadamard, the operator found turned out to be the well-known pseudo-Hadamard which was considered the superior gate for NMR based quantum circuits.

The second was when we applied this method to design the quantum oracle for Deutsch’s problem. The oracle is typically generated by hand and in seconds our EA was able to derive a valid oracle operator which was in fact different from Deutsch’s but had the same functionality.

While the EA was limited to 5 or less qubit studies, it was surprisingly quick to find valid solutions. This was especially true after the EA code matured and a decent understanding of the bounds of the EA parameters were understood which took numerous simulations. Valid solutions were found in only a handful of generations. This makes the EA extremely viable in terms of run-time. In fact, many aspects of the EA are desirable including its simplicity (that is, ease of mapping it to a problem), highly parallelizable, and memory efficient.

### **The Quantum Logic Algebraic Method**

The quantum logic algebraic method was published in [5] and described in chapter 4. This significant work allows the algebraic representation of more complex quantum circuits, and specifically those with controlled-gates. Until this work was published, there was no known method for representing the unitary operator of such circuits.

With this method also came the ability to apply mathematical rules and transformations (namely, simple Kronecker and matrix identities) to quantum circuit expressions. One of the first applications of this method was to easily prove known quantum circuit identities found in any quantum computing textbook.

While the method, its canonical representation, and proofs are provided in this dissertation, its true power will be in its ability to represent quantum

circuits and operate on them. For example, a quantum circuit can now be represented by an equation which can be simplified and manipulated. This leads to a potentially rich source of “future work” in fields such as quantum logic synthesis. We also show how simple it is to extend our quantum logic algebraic method to multi-valued quantum logic.

## **Operator Eigenanalysis**

Section 6 details our eigenanalysis methods which resulted in a couple primary contributions. Overall, our attempts to derive patterns and information from the eigenanalysis was unsuccessful. However, there were a couple of mathematical tools we developed that should be explored in more depth.

For example, the technique which shows the most promise is described in section 6.4 which provides a method to adjust the eigenvalues of a quantum operator without destroying its unitary property. This method provides a mechanism to not only test the impact of modifying individual eigenvalues, but also gives us another method to optimize the operator through eigenvalue tuning.

In section 6.3, we derive a way to map the eigenvalues and eigenvectors from the spectral decomposition format into the common outer-product format using equation 6.3.12. This method provides a means to describe the function of a quantum operator in terms of the more familiar standard computational basis set. Unfortunately, this method was not applied in our eigenanalysis and although convenient, it’s debateable as to its usefulness.

## 1.4 Dissertation Organization

Chapter 2 provides background on what we felt were the pertinent areas. We give a brief introduction to Quantum Mechanics but quickly dive into Quantum Computing with a focus on basic notation, Hilbert spaces, Qubits, Quantum Circuits, and a detailed overview of Quantum Algorithms. Lastly, we cover basic concepts of Evolutionary Algorithms including Evolution Strategies (ES).

In Related Research (chapter 3), we review current topics that are directly applicable to our research. We start with a survey of known applications of evolutionary algorithms to quantum computing with a more detailed overview of a method developed by Dr. Martin Lukac [7] which uses a genetic algorithm (GA) to synthesize quantum operators. In this chapter we also describe the method to generate random unitary matrices which is at the heart of our EA. Lastly, we review quantum logic synthesis methods including the popular cosine-sine decomposition (CSD) as used by Qubiter.

In chapter 4 we detail our Quantum Logic Algebraic method. Since we're inventing a method, we give due diligence to setting up a proper framework for the method, examples, and proofs for its canonical representation.

Chapter 5 is our main section describing the development of our Evolutionary Algorithm utilizing the random unitary matrix method with an Evolution Strategy (ES). This is also where all of our evolution results and analysis occurs.

We placed the results and analysis of quantum logic synthesis into chapter 7. In this chapter we give a bit more detail into how Qubiter was used which was not relevant to the EA. The synthesis results of key studies from chapter 5 are found here.

We provide more detail by summarizing our work and major findings in

chapter 8.

Lastly, we conclude our dissertation in chapter 9 with a review of the future work applications of our research.

## Chapter 2

### Background

#### 2.1 Quantum Mechanics

While Quantum Mechanics has a broad range of topics, only those more specifically directed toward Quantum Computation are covered.

##### 2.1.1 A Quantum of History

Quantum mechanics is a description of the behavior of light and matter particularly at the atomic and sub-atomic level. Its development started around 1900 with the concept that energy is quantized (a *quanta* of energy) as a result of Max Planck's theory of blackbody radiation. A few years later in 1905, Albert Einstein explains the photo-electric effect by assuming that light is composed of light particles (later called *photons*) which move at the speed of light,  $c$ , and have energy  $h\nu$ , also contributing to the concept of energy quanta. Louis de Broglie took this concept (that is, the particle-like nature of light) and in an interesting reversal, postulated that all matter has a wave-like nature determined by the relation  $p = h\lambda$ .

At this point, there was strong evidence that a new type of physics much different from classical physics was needed to explain the atomic and sub-atomic



behavior being seen. By 1925, Max Born and Pascual Jordan developed a mathematical foundation using infinite matrices and the use of non-commutative dynamical variables. At the same time, Paul Dirac introduced his abstract mathematical system and postulated a general form for the commutator between two quantum variables using Poisson brackets.

Everything came together in 1926 with a cascade of events. Erwin Schrödinger proposed his famous Wave Equation. Werner Heisenberg developed a theory of quantum mechanics using infinite matrices to represent observables. Dirac and Schrödinger showed the wave equation, Dirac's algebraic method, and Heisenberg's infinite matrix methods were equivalent. Finally, that same year, Dirac — and independently, Born, Heisenberg, and Jordan — obtained a complete formulation of quantum mechanics that could be applied to any physical system, and was first applied to the hydrogen atom.

Also in 1926, John von Neumann — after attending a lecture by Heisenberg which involved discussions with David Hilbert who proposed Hilbert spaces — introduced the concept of applying Hilbert spaces to quantum mechanics. He did this by demonstrating that the geometry of vectors over the complex plane has the same formal properties as the states of a quantum mechanical system. The states of the quantum system (the wave functions) are represented as vectors in Hilbert Space and operations associated with position and momentum act like matrices operating on these vectors. This concept has served as the basis for describing the computational properties of quantum systems.

<i>Notation</i>	<i>Description</i>
$c^*$	Complex conjugate of $c$
$ \psi\rangle$	The <i>ket</i> or column vector of state $\psi$ .
$\langle\psi $	The <i>bra</i> or <i>dual</i> of $\psi$ . A row vector.
$\langle\psi \phi\rangle$	The <i>bra(c)ket</i> , or <i>inner-product</i> .
$ \psi\rangle \otimes  \phi\rangle$	The tensor product. Simplifies to $ \psi\rangle  \phi\rangle$ .
$ \psi\rangle \langle\phi $	The <i>outer-product</i> vector operation.
$A^*$	The <i>complex conjugate</i> of matrix $A$ .
$A^T$	The <i>transpose</i> of matrix $A$ .
$A^\dagger$	The <i>adjoint</i> (conjugate–transpose) of matrix $A$ .
$A \psi\rangle$	<b>A</b> operating on vector $ \psi\rangle$ .
$\langle\phi A \psi\rangle$	The inner-product of $ \phi\rangle$ and $A \psi\rangle$ .

Table 2.1: Dirac’s Notation

### 2.1.2 Dirac’s Notation

The common notation and the notation we use to describe quantum states was introduced by Dirac [8]. Here, Dirac introduces the so-called *bra–ket* vector notation. A *bra*, denoted as  $\langle|$ , is a row vector. The *ket*, denoted as  $| \rangle$  is a column vector. We can denote a quantum state ( $\psi$ ) as either a row vector using *bra*,  $\langle\psi|$ , or a column vector using *ket*,  $|\psi\rangle$ . The relationship between them is quiet simple. The *bra* is the conjugate–transpose (also known as the hermitian–adjoint) of *ket*. That is,

$$\langle\psi| = (|\psi\rangle^*)^T = (|\psi\rangle)^\dagger \quad (2.1.1)$$

For this reason, the *bra* is called the *dual* of the *ket* vector.

Table 2.1 gives an overview of the basic operation used with Dirac’s notation.

### 2.1.3 Hilbert Space

Hilbert spaces play a vital role in quantum computing and so they deserves some coverage as part of our background material. One can say the space quantum mechanics (and thus quantum computing) operates in is a Hilbert space. In other words, a Hilbert space is a mathematical framework that can be used to describe quantum mechanics. With that in mind, we give the definition of a Hilbert space:

**Definition 1** *A Hilbert Space,  $\mathcal{H}_n$ , is an  $n$ -dimensional Euclidean vector space over the field of complex numbers with a well defined inner-product defined by*

$$\langle x, y \rangle = \sum_{i=1}^n x_i y_i^* \quad (2.1.2)$$

*and associated norm (length) given by*

$$\|x\| = \sqrt{\langle x, x \rangle} \quad (2.1.3)$$

*A Hilbert has a complete metric with respect to the inner-product. The elements of  $\mathcal{H}_n$  are  $n$ -dimensional complex vectors and any linear combination of vectors within the Hilbert space will result in a vector which also exists in  $\mathcal{H}_n$ . We also note that  $\mathcal{H}_n$  is isomorphic with  $\mathbb{C}^n$ .*

*Hilbert spaces have the following inner-product rules:*

- 1. The inner-product of two vectors  $|\psi_a\rangle, |\psi_b\rangle \in \mathcal{H}_n$  is a complex number.*
- 2. Inner-product with itself is a real number:  $\langle \psi_a | \psi_a \rangle \in \mathbf{R}$ .*
- 3. Linearity: If  $|\psi_a\rangle, |\psi_b\rangle, |\psi_c\rangle \in \mathcal{H}_n$  and  $a, b, c \in \mathbf{C}$  then*

- $\langle \psi_a | (c |\psi_b\rangle) = c \langle \psi_a | \psi_b \rangle$ .
- $(a \langle \psi_a | + b \langle \psi_b |) |\psi_c\rangle = a \langle \psi_a | \psi_c \rangle + b \langle \psi_b | \psi_c \rangle$ .

4. *Skew Symmetry:*  $\langle \psi_a | \psi_b \rangle = \langle \psi_a | \psi_b \rangle^*$ .

5. *Satisfies the Schwartz inequality:*  $\langle \psi_a | \psi_a \rangle \langle \psi_b | \psi_b \rangle \geq |\langle \psi_a | \psi_b \rangle|^2$ .

An important aspect about Hilbert spaces is their orthonormal basis vectors which we define as:

**Definition 2** *The collection of vectors  $\{v_1, v_2, \dots, v_n\} \in \mathcal{H}_n$  is called the orthonormal basis if the inner product of any two of them is zero,  $(v_i, v_j) = 0 \forall (i, j) \in \{1, n\}$ , and the inner product of any of them with itself is one,  $(v_i, v_i) = 1 \forall i \in \{1, n\}$ . There can be many choices. However, all orthonormal basis sets of a Hilbert space have the same cardinality.*

An example of a Hilbert space is  $\mathcal{R}^3$  formed by the orthonormal basis vectors  $(1, 0, 0)$ ,  $(0, 1, 0)$ , and  $(0, 0, 1)$  and the dot-product.

### 2.1.4 Tensor and Kronecker Products

The *tensor product* is defined by the symbol  $\otimes$  and is used to combined two vector spaces into a single larger vector space. We apply this directly to quantum systems with the following definition:

**Definition 3** *If a quantum system  $\mathcal{S}$  is composed of two quantum sub-systems,  $\mathcal{S}_1$  and  $\mathcal{S}_2$ , with corresponding Hilbert spaces,  $\mathcal{H}_1$  and  $\mathcal{H}_2$ , then the space of  $\mathcal{S}$  is defined by the Hilbert space  $\mathcal{H}$  which is the tensor product of  $\mathcal{H}_1$  and  $\mathcal{H}_2$ . That is,*

$$\mathcal{H} = \mathcal{H}_1 \otimes \mathcal{H}_2$$

We make a simplification by noting that in quantum computing we are dealing with finite linear operators and we can instead use the *Kronecker product*:

**Definition 4** *The Kronecker product is defined by the symbol  $\otimes$  and is a special case of tensor product when using finite linear operators. Given the  $m \times n$  linear operator,  $A$ , and an  $o \times p$  linear operator,  $B$ , then the Kronecker product, denoted  $A \otimes B$  is the  $mo \times np$  operator,  $C$ , where*

$$C = A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix}$$

The Kronecker Product is *bilinear* and *associative* and not *commutative*. Below are some key identities of the Kronecker product:

1.  $A \otimes (B + C) = A \otimes B + A \otimes C$
2.  $(A + B) \otimes C = A \otimes C + B \otimes C$
3.  $(kA) \otimes B = A \otimes (kB) = k(A \otimes B)$
4.  $(A \otimes B) \otimes C = A \otimes (B \otimes C)$
5.  $(A \otimes B)(C \otimes D) = AC \otimes BD$
6.  $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$

Another interesting aspect is how the eigenvalues are combined when performing the Kronecker product:

**Definition 5** *Given square matrix,  $A$ , of size  $n$  with eigenvalues  $\{\lambda_i\}$  and square matrix,  $B$ , of size  $m$  with eigenvalues  $\{\mu_j\}$ , then the eigenvalues of  $A \otimes B$  are given by  $\{\lambda_i \mu_j\}$  where  $i = \{1, \dots, n\}$  and  $j = \{1, \dots, m\}$ .*

Kronecker products are a fundamental operation in quantum computing. They can be used explicitly with the  $\otimes$  symbol and they can also be used implicitly as with Dirac's notation where  $|\psi\phi\rangle$  is actually implied to be  $|\psi\rangle\otimes|\phi\rangle$ .

### 2.1.5 Quantum States

A *quantum state* is a complete description of a physical system and is represented by an  $n$ -dimensional vector in a Hilbert space,  $\mathcal{H}_n$ . For now, we'll limit our quantum state to have a unit length. This specialized Hilbert space is sometimes called a *unitary space*. In this space, a general unit quantum state vector has the form:

$$|\psi\rangle = c_0 |0\rangle + c_1 |1\rangle + \dots + c_{n-1} |n-1\rangle = \sum_{i=0}^{n-1} c_i |i\rangle \quad (2.1.4)$$

Since the *length* or *norm* of a ket (or corresponding bra) vector must be unity in our unitary space, the quantum state has the following condition:

$$\sum_{i=0}^{n-1} |c_i|^2 = 1 \quad (2.1.5)$$

Where  $c_i$  are the complex amplitudes and it can also be shown that

$$c_i = \langle i|\psi\rangle$$

We recall the *braket* relationship between unit vectors:

$$\langle i|j\rangle = \begin{cases} 0 & \text{if } j \neq i \\ 1 & \text{if } j = i \end{cases}$$

We can visualize quantum states in this space as unit length rays so that

only their direction is of importance. Given two states,  $|\psi_a\rangle$  and  $|\psi_b\rangle$ , the generalized angle between those states is given by the inner-product,  $\langle\psi_a|\psi_b\rangle$  and can represent the *overlap* between those two states. Thus,  $\|\langle\psi_a|\psi_b\rangle\|$  is a measure of the *relative orthogonality* between those two states.

A quantum state can be in a *superposition* state, that is,  $|\psi_c\rangle = a|\psi_a\rangle + b|\psi_b\rangle$ . However, for this new state to exist in our unitary space, it must have a unity norm. Therefore, the normalization constraint requires that  $|a|^2 + |b|^2 = 1$ .

An interesting aspect of quantum state is *relative phase*. For instance, the state  $|\psi\rangle$  and the state  $e^{i\gamma}|\psi\rangle$  (where  $|e^{i\gamma}| = 1$ ) describe the same physical state but differ by a *relative phase* represented by  $\gamma$ .

All quantum states in our unitary space are composed of linear combinations of orthonormal basis vectors. For example, the set of unit vectors  $\{|0\rangle, |1\rangle, \dots, |n-1\rangle\}$  forms a *normal unitary basis* in the  $4^{th}$ -dimensional Hilbert space,  $\mathcal{H}_4$ :

$$|00\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad |01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \quad |10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \quad |11\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

The state vector,  $|\psi\rangle \in \mathcal{H}_n$  can be expressed as a linear combination of the basis state,  $|0\rangle, |1\rangle, \dots, |n-1\rangle$  as  $|\psi\rangle = \sum_{i=0}^{n-1} \alpha_i |i\rangle$  with  $\alpha_i$  representing the complex amplitudes.

### 2.1.6 Quantum Operators

The role of unitary operators in quantum computing is absolutely fundamental and will be explored in more detail in section 2.2 and throughout this dis-

sertation. Here we present the basic concept of quantum operators and their properties and then focus on the importance of Hermitian and Unitary operators.

First, we look at the definition of a linear operator operating in a Hilbert space:

**Definition 6** *An  $n$ -dimensional linear operator,  $A$ , on a Hilbert space  $\mathcal{H}_n$  is a linear mapping  $A : H \rightarrow H$ . And can have the following properties:*

1.  $A = \text{Hermitian}$  if  $A = A^\dagger$
2.  $A = \text{Unitary}$  if  $AA^\dagger = A^\dagger A = I$
3.  $A = \text{Normal}$  if  $[A, A^\dagger] = AA^\dagger - A^\dagger A = 0$

A *Hermitian* operator,  $\mathbf{H}$ , maps states vectors to state vectors in  $\mathcal{H}_n$ . That is,  $|\psi'\rangle = \mathbf{H}|\psi\rangle$  where  $\mathbf{H}$  is described by the matrix,  $H$ , whose elements are defined as  $H_{ij} = \langle i|O|j\rangle$  where  $|i\rangle$  and  $|j\rangle$  are unit vectors that satisfy the Kronecker delta relation:  $\langle i|j\rangle = \delta_{ij}$ .

Hermitian operators also have the following properties:

1. Real eigenvalues.
2. The determinant equals the product of its eigenvalues and thus is real.
3. The *trace* (sum of diagonal elements),  $\text{Tr}(U)$ , equals the sum of its eigenvalues and is also real.

A *Unitary* operator also has the property that its determinant is 1.

By definition, a unitary–hermitian operator is normal. Unitary preserves the inner product, preserving distance.



Through quantum operators, we evolve the quantum state to a new state. That is,  $|\psi'\rangle = U|\psi\rangle$ .

To understand why we are interested in unitary operators in quantum mechanics, we look at the time-independent Schrödinger equation:

$$i\hbar \frac{\partial x(t)}{\partial t} = Hx(t) \quad (2.1.6)$$

Where  $x$  is the quantum state and  $H$  is the Hamiltonian. The well known solution being:

$$x(t) = e^{-iHt/\hbar}x(0) \rightarrow |x'\rangle = U|x\rangle \quad (2.1.7)$$

Because  $H$  is Hermitian,  $U = e^{-iH/\hbar}$  is a unitary operator. This sets the stage for us to investigate specifically unitary operators.

### 2.1.7 Quantum Observables and Measurements

In quantum mechanics, in order to extract quantum information from a quantum system, we need to *observe* or *measure* the system. An *observable* is a property of a physical system that can be measured. For example, position, velocity, and momentum.

An observable is associated with a Hermitian operator. The measured value of a *pure state* of an observable is an eigenvalue of its operator. That is, given a unitary operator  $U$ , the state  $|\psi\rangle$  is considered a pure state if it is also an eigenvector (or eigenket) of  $U$ . This is different from a *mixed-state* which is a linear superposition of pure states.

Quantum projection *measurements* are tests with real-valued labels such as velocity or position. For measurement we use the *projection operator* (or simply,

projector). A projection operator is the *outer product* of any state vector with itself:

$$\mathbf{P}_{\psi_a} = |\psi_a\rangle \langle \psi_a| \quad (2.1.8)$$

With the property

$$(\mathbf{P}_{\psi_a})^2 = \mathbf{P}_{\psi_a}$$

We intuitively understand why this property exists. When we take a measurement we essentially project the complex state function,  $|\psi_a\rangle$ , onto the real plane. We sometimes refer to this as *collapsing* the quantum wave function. Successive projections would have no effect since we've already removed the complex components. In measuring a quantum state, this is analogous to the measurement collapsing the state, thus removing the complex (invisible) components of the quantum wave, and projecting on to the real plane. Any further measurements would always result in the same measurement since the complex components have already been removed.

A complete set of orthogonal projectors  $\{P_0, P_1, \dots, P_{n-1}\}$  for a given  $n$ -dimensional Hilbert space,  $\mathcal{H}_n$ , follows the condition:

$$\sum_{i=0}^{i=n-1} P_i = 1$$

This tells us that the sum of probabilities after measuring all states is 1. Since our projectors form an orthogonal basis set for our linear operator, they can be used in a spectral representation (covered in more detail in section 2.1.8). More formally, in an  $n$ -dimensional Hilbert space,  $\mathcal{H}_n$ , every normal operator  $\mathbf{N}$  has  $n$  eigenvectors,  $\{|n_0\rangle, |n_1\rangle, \dots\}$  and  $n$  corresponding eigenvalues,  $\{\lambda_0, \lambda_1, \dots\}$ . If  $\mathbf{P}_i$  is the projector corresponding to these eigenvectors, that is,  $\mathbf{P}_i = |n_i\rangle \langle n_i|$ , then

the operator  $\mathbf{N}$  has the *spectral decomposition*  $\mathbf{N} = \sum_i \lambda_i \mathbf{P}_i$ . The probability of measuring eigenvalue  $\lambda_i$  is given by

$$\text{Prob}(\lambda_i) = ||P_i |\psi\rangle||^2 = \langle\psi| P_i |\psi\rangle$$

Immediately after measurement, the quantum state collapses to  $|\psi'\rangle$ , which is an *eigenstate* of the operator. This new state is defined as:

$$|\psi'\rangle = \frac{P_i |\psi\rangle}{\sqrt{\langle\psi| P_i |\psi\rangle}}$$

For completeness, the total probability for all outcomes must be unity:

$$\sum_{i=0}^{n-1} \text{Prob}(\lambda_i) = 1$$

## The Density Operator

The *density matrix* contains all the information regarding the results of measurements of an ensemble of  $N$  independent versions of a quantum system and gives the expected value of any observable of the system. The density matrix does not uniquely determine the states of individual particles. The density matrix is Hermitian and its eigenvalues are non-negative. Given all the possible states of a system, we define the density operator,  $\rho$ , as:

$$\rho = \frac{1}{N} \sum_{i=0}^{N-1} \langle\psi_i|\psi_i\rangle \quad (2.1.9)$$

The density operator gives us an alternative method to measure the outcome of an observable represented by an operator. For instance, let's say we have an

operator,  $\mathbf{M}$ , which has eigenstates denoted by the set  $\{|e_m\rangle\}$ . We define the *ensemble average* of  $N$  quantum systems each in one of  $N$  possible states,  $|\psi_i\rangle$  where  $0 \leq i < N$  as:

$$\langle M \rangle = \frac{1}{N} \sum_{i=0}^{N-1} \langle \psi_i | M | \psi_i \rangle$$

Then we can define  $\langle M \rangle$  in terms of the density matrix as

$$\langle M \rangle = \sum_m \langle e_m | \rho M | e_m \rangle = \text{Tr}(\rho M)$$

Our projector is defined as  $\mathbf{P}_m = |e_m\rangle \langle e_m|$ . This allows us to calculate the probability of measuring  $M$  in the *pure* state  $|e_m\rangle$  as

$$\text{Prob}(|e_m\rangle) = \frac{1}{N} \sum_{i=0}^{N-1} |\langle \psi_i | e_m \rangle|^2 = \text{Tr}(\rho P_m)$$

The density operator is Hermitian if  $\{|e_m\rangle\}$  form an orthonormal basis. It also follows that the *expectation* — that is, the average value we would measure — is unity. That is,

$$\text{Tr}(\rho) = 1$$

### 2.1.8 Spectral Decomposition

Spectral decomposition provides a way of representing a matrix as an expression based on its eigenvalues and eigenvectors. This method has many names: *spectral representation*, *spectral decomposition*, *eigenvalue decomposition*, *diagonal representation*, *orthonormal decomposition* (for unitary matrices), and even *eigendecomposition*. We give this particular subject focus because we use it in two key places: In section 6.3 to map eigenvectors to the standard computa-

tional basis, and also in section 6.4 to provide a method of adjusting eigenvalues of a quantum operator while maintaining its unitary property.

Spectral decomposition falls under the general category of spectral theory in linear algebra and is a special case of SVD (singular value decomposition). It is well suited for self-adjoint matrices (e.g., Hermitian) and, more generally, normal operators in the Hilbert space. This also includes unitary operators.

We'll consider the cases of a Unitary operator in a Hilbert space since this will apply to the types of operators we will deal with later in our review of quantum computing.

**Definition 7** *Given the operator  $\mathbf{A}$  which is self-adjoint (that is,  $A = A^\dagger$ ).  $\mathbf{A}$  has a spectral decomposition defined as:*

$$A = \sum_{i=1}^N \lambda_i |i\rangle \langle i| \quad (2.1.10)$$

*Where the  $|i\rangle$  vectors form a set of orthonormal eigenvectors with  $\lambda_i$  being the corresponding set of eigenvalues.*

We derive this decomposition by solving the characteristic equation

$$A |\psi\rangle = \lambda I |\psi\rangle$$

Which is re-written as

$$(A - \lambda I) |\psi\rangle = 0$$

and will have a non-trivial solution iff

$$\det(A - \lambda I) = 0$$

Hermitian operators have eigenvalues that are real numbers and the eigenvectors corresponding to different eigenvalues will be mutually orthogonal.

We can also use the *projector* convention to form an alternative decomposition form. We define  $P_i = |i\rangle\langle i|$ , where  $P_i$  is the *projection* operator into the subspace of  $\mathcal{H}_n$  spanned by the  $n$  eigenvectors corresponding to  $\lambda_i$ . Therefore,

$$A = \sum_{i=1}^N \lambda_i P_i$$

## 2.2 Quantum Computing

Quantum computation has spread into many fields, initially starting with computing[9] and information[10], and quickly spreading to a wide range of fields such as cryptography[11], artificial intelligence[12], game theory[13], economics[14, 15], and control systems[16, 17].

The computational properties of quantum mechanics were originally investigated by Benioff[18], while the concept that quantum mechanics could be more computationally powerful than a classical Turing machine is attributed to Feynman[19, 20]. Since then researchers have developed methods of describing quantum computers and developed quantum algorithms which can solve problems more efficiently than on a classical Turing machine.

Deutsch defined the quantum Turing machine[21] and the concept of quantum circuits (or gates)[22]. Along with Jozsa, they answered the long-standing question that a quantum computer could solve a sample problem more efficiently compared to a classical Turing machine[23]. More examples were found by Berthiaume and Brassard[24], Simon[25], and Bernstein and Vazirani[26]. This led to the discovery by Shor[27] of a polynomial-time quantum algorithm for two

key problems (discrete log and factoring) for which no polynomial-time classical algorithm is known. Another significant discovery is Grover's algorithm[28] for database searching.

### 2.2.1 Qubits

Classical computer systems represent a single bit of information deterministically: the value is either a logic 0 or a logic 1. Quantum computer systems represent a single bit of information as a *qubit*, which is a unit vector in a complex Hilbert space  $\mathbf{C}^2$ . The ideas are commonly expressed using *bra/ket* notation introduced by Dirac[8] and was summarized in section 2.1.2.

Any practical quantum computer manipulates a register of  $n$  qubits. If each qubit has an orthonormal basis  $\{|0\rangle, |1\rangle\}$ , then an  $n$  qubit system has a basis expressed by the *tensor product*:  $(C^2)^{\otimes n} = C^2 \otimes C^2 \otimes \dots \otimes C^2$ . This gives  $2^n$  total basis vectors. In general,  $|a\rangle$  denotes the tensor product  $\bigotimes_{i=0}^n |a_i\rangle = |a_n\rangle \otimes |a_{n-1}\rangle \otimes \dots \otimes |a_1\rangle \otimes |a_0\rangle$  which means a quantum register has the value  $a = 2^0 a_0 + 2^1 a_1 + \dots + 2^n a_n$ .

A qubit need not exist in only one basis state. Indeed, a qubit can exist as a *linear superposition* of basis state  $c_0 |0\rangle + c_1 |1\rangle$ , where  $c_0$  and  $c_1$  are complex numbers satisfying  $|c_0|^2 + |c_1|^2 = 1$ . More generally, the  $n$  qubit register can be prepared in a superposition of all possible classical states:

$$|x\rangle = \sum_{i=0}^{2^n-1} c_i |i\rangle \quad (2.2.1)$$

where the normalization condition  $\sum_i |c_i|^2 = 1$  must hold. The complex number  $c_i$  is called the *amplitude* associated with the state  $|i\rangle$ .

The most conventional representation of a base state  $|i\rangle$  is as a column matrix

with the  $i^{th}$  entry 1 and all other entries 0. A state  $|\psi\rangle$  is therefore represented as a column matrix of the complex amplitudes. That is,

$$|\psi\rangle = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \end{pmatrix}$$

Quantum systems evolve from state to state according to Schrödinger's equation [29]. Suppose we start in state  $|\psi\rangle = \sum c_i |i\rangle$ . A linear operator  $U$  produces a new state  $|\phi\rangle = U|\psi\rangle$ . Both states are linear combinations of the same base states, so  $|\phi\rangle = \sum c'_i |i\rangle$ . This means evolution occurs by modification of the state amplitudes. Note that the normalization condition required of states is satisfied iff  $U$  is unitary—i.e.,  $U^\dagger U = I$ .

It is important to emphasize the role superposition plays in quantum computing. Consider a state  $|\psi\rangle = \sum c_i |i\rangle$ . One can exploit the superposition using the property of *quantum interference*. Interference allows the exponential number of computations performed in parallel to either cancel or enhance each other. Feynman [29] beautifully describes how light waves can constructively or destructively interfere to produce this effect. The goal of any quantum algorithm is to have a similar phenomena occur—i.e., interference increases the amplitudes of computational results we desire and decreases the amplitudes of the remaining results. It is a unitary operator that would alter these amplitudes.



## Bloch Sphere Representation

The *Bloch Sphere* allows us to visualize the state of a single qubit in a three-dimensional space as shown in figure 2.1. A qubit can exist on any point on the *surface* of the Bloch Sphere. To better understand this visual representation, we first construct a general expression for the state of a qubit using  $\phi$ ,  $\psi$ , and  $\gamma$  as our three angle parameters to form

$$|\psi\rangle = e^{i\gamma} \left( \cos \frac{\theta}{2} |0\rangle + e^{i\phi} \sin \frac{\theta}{2} |1\rangle \right)$$

The *global phase factor* impact of  $\gamma$  represented by the  $e^{i\gamma}$  term is generally ignored in our calculations since it is not observable.

The state of a qubit is the unit radius vector extending outwards from the origin of an  $x, y, z$  3D space. The direction of the vector is defined by the two angles,  $\theta$  and  $\phi$ . The angle  $\theta$  is the rotation of the vector from the  $z$  axis and the angle  $\phi$  is the rotation from the  $x$  axis on the  $xy$  plane. The Bloch Sphere is the three-dimensional shaped formed by varying  $\theta$  and  $\phi$  over all possible values.

A qubit can be in a continuum of states as represented by the surface of the Bloch Sphere. We arbitrarily assign two basis states – also known as the computational basis states – to the qubit,  $|0\rangle$  and  $|1\rangle$ . On the Bloch Sphere representation, the  $|0\rangle$  state is the top-most point on the  $z$ -axis and  $|1\rangle$  is the bottom-most point on the  $z$ -axis.

As an example, the point on the  $x$ -axis access where  $\phi = 0^\circ$  and  $\theta = 90^\circ$

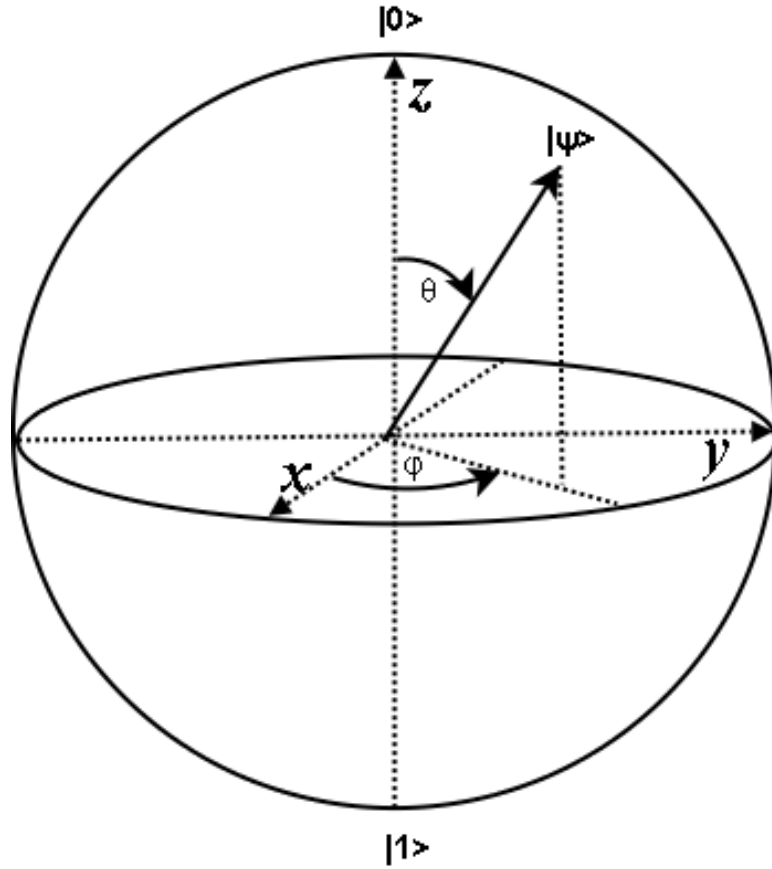


Figure 2.1: Bloch Sphere representation of a Qubit

gives us a qubit in common superposition state

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

Apart from showing how a qubit can be in a *superposition* state of  $|0\rangle$  and  $|1\rangle$ , the Bloch Sphere also allows us to visualize how single qubit operations will affect the current state of the qubit. Later, we'll discuss single qubit operators such as the  $X$ ,  $Y$ , and  $Z$  rotation gates. Fundamentally, all qubit operators cause the current state of the vector on the Bloch Sphere to rotate to another

point on the sphere.

## **Qubit Measurements**

The state of a qubit register is determined by a measurement. In quantum systems, this measurement process projects the system state onto one of the basis states. Referring to Eq. 2.2.1, the measurement returns a value of  $|i\rangle$  with probability  $|c_i|^2$ . Any subsequent measurement returns the state  $|i\rangle$  with probability 1, which means the measurement process irreversibly alters the state of the system. Measurement also gives another perspective on entanglement: two qubits are entangled *if and only if* the measurement of one affects the state of the other.

### **2.2.2 Quantum Gates**

#### **Single Qubit Quantum Gates**

Single qubit gates are defined by  $2 \times 2$  square matrices and represent the simplest quantum logical operations. The most obvious examples are the identity ( $I$ ) gate and the inverter gate ( $X$ ) since these are analogous to their classic logic counterparts, the buffer (or wire) and the inverter (or NOT) gate, respectively.

In general, a single qubit exists in a superposition of the logical basis states as given in the equation below

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

Where  $|\alpha|^2 + |\beta|^2 = 1$ .

To understand single qubit operators in more depth, we'll reference our Bloch

Sphere discussion and then look at the types of rotation operations (which are also single qubit operators) and then form a general qubit operator.

In the Bloch Sphere there are three axes (x,y, and z) and thus three rotation operators. Without going into the details, we list those three rotations below

$$R_x(\theta) \equiv e^{-i\theta X/2} = \cos \frac{\theta}{2} I - i \sin \frac{\theta}{2} X = \begin{pmatrix} \cos \frac{\theta}{2} & -i \sin \frac{\theta}{2} \\ -i \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{pmatrix} \quad (2.2.2)$$

$$R_y(\theta) \equiv e^{-i\theta Y/2} = \cos \frac{\theta}{2} I - i \sin \frac{\theta}{2} Y = \begin{pmatrix} \cos \frac{\theta}{2} & -\sin \frac{\theta}{2} \\ \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{pmatrix} \quad (2.2.3)$$

$$R_z(\theta) \equiv e^{-i\theta Z/2} = \cos \frac{\theta}{2} I - i \sin \frac{\theta}{2} Z = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix} \quad (2.2.4)$$

We define the general single qubit operator as the product of the rotation operations by first rotating over  $x$ , then  $y$ , and lastly  $z$ . To achieve this, we do the matrix multiplication in reverse. Lastly, we add the global phase term ( $e^{i\gamma}$ ) to produce equation 2.2.5.

$$U = e^{-i\gamma} R_z(\theta_z) R_y(\theta_y) R_x(\theta_x) \quad (2.2.5)$$

In fact, any unitary  $2 \times 2$  matrix with unity determinant can also be expressed as

$$U = \begin{pmatrix} e^{i\theta_z/2} & 0 \\ 0 & e^{-i\theta_z/2} \end{pmatrix} \begin{pmatrix} \cos \theta_y/2 & \sin \theta_y/2 \\ -\sin \theta_y/2 & \cos \theta_y/2 \end{pmatrix} \begin{pmatrix} e^{i\theta_x/2} & 0 \\ 0 & e^{-i\theta_x/2} \end{pmatrix} \quad (2.2.6)$$

Now we can review some of the common single qubit operators and we can see how these are derived from the rotation operators in equation 2.2.4.

### The Identity Gate, I

This is sometimes called the Pauli I gate. The function of the gate is trivial as the output state is the same as the input state. The matrix is the identity

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

And denoted by the following quantum circuit

$$\text{---}\boxed{I}\text{---}$$

### The Inverter, X Gate

This is sometimes called the Pauli X gate. The function of the gate is to invert the logical state of the qubit much like classical logic inverter. The difference is that the quantum inverter can operate on superposition states. If the qubit is in the  $|0\rangle$  state, then the result will be  $|1\rangle$ . And vice-versa, if the qubit was in the  $|1\rangle$  state, then the result will be  $|0\rangle$ . It is defined by the matrix

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

And denoted by the following quantum circuit

$$\text{---}\boxed{X}\text{---}$$

## The Y Gate

The Pauli Y gate performs the following mapping on the logical states

$$Y |0\rangle = i |1\rangle$$

$$Y |1\rangle = -i |0\rangle$$

It is defined by the matrix

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

And denoted by the following quantum circuit

$$\text{---}\boxed{Y}\text{---}$$

## The Z Gate

The Pauli Z gate changes the relative phase factor by  $-1$ , effectively negating a qubit's sign, for the  $|1\rangle$  component of the state. It performs the following mapping on the logical states

$$Z |0\rangle = |0\rangle$$

$$Z |1\rangle = -|1\rangle$$

It is defined by the matrix

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

And denoted by the following quantum circuit

$$\text{---}\boxed{Z}\text{---}$$

### The Phase Gate, S

The Phase (or  $S$ ) gate performs the following mapping on the logical states

$$S|0\rangle = |1\rangle$$

$$S|1\rangle = i|0\rangle$$

It is defined by the matrix

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

And denoted by the following quantum circuit

$$\text{---}\boxed{S}\text{---}$$

### The T Gate

This is sometimes called the  $\pi/8$  for the reason that up to a certain global phase, the  $T$  gate behaves exactly as another gate which has  $e^{i\pi/8}$  appearing in its diagonals. The  $T$  gate is defined by the matrix

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$$

And denoted by the following quantum circuit

$$\text{---}\boxed{T}\text{---}$$

## The V Gate

This is sometimes called the  $\sqrt{\text{NOT}}$  because it is literally the square-root of the  $X$  gate matrix. It showcases the multiplicative nature of quantum logic since two  $V$  operators will produce the result of a logic  $X$  gate.

The  $V$  gate is defined by the matrix

$$V = \frac{1}{2} \begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix}$$

And denoted by the following quantum circuit

$$\text{---} \boxed{V} \text{---}$$

## The Hadamard and Pseudo-Hadamard Gates

The Hadamard gate is extremely useful because it maps the basis states  $|0\rangle$  and  $|1\rangle$  into a superposition of the two states with equal weight. That is,

$$|0\rangle \rightarrow (|0\rangle + |1\rangle)/\sqrt{2}$$

and

$$|1\rangle \rightarrow (|0\rangle - |1\rangle)/\sqrt{2}$$

It is defined by the matrix

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$



And denoted by the following quantum circuit

$$-\boxed{H}-$$

We will commonly see typical quantum computing algorithms put their initial state into an equal superposition using Hadamard gates to extract the quantum computing parallelism that superposition provides.

The Pseudo-Hadamard gate[30] is practically equivalent to the standard Hadamard in that it can convert a basis state to an equal superposition state. It performs the following operations,

$$|0\rangle \rightarrow (|1\rangle + |0\rangle)/\sqrt{2}$$

and

$$|1\rangle \rightarrow (|1\rangle - |0\rangle)/\sqrt{2}$$

The difference from the Hadamard is subtle and shown by the negation for the operation on the  $|1\rangle$  state. It is defined by the following matrix which is also subtly different from the Hadamard,

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}$$

Some implementations utilizing NMR technologies found benefit by using the Pseudo-Hadamard gate over a regular Hadamard [31].

## Two Qubit Quantum Gates

Two qubit quantum gates have two inputs and can operate on one or both of the qubits. The state of a two-qubit can be expressed as a linear combination of all the computational basis states

$$|\psi\rangle = c_0 |00\rangle + c_1 |01\rangle + c_2 |10\rangle + c_3 |11\rangle$$

Where  $|c_0|^2 + |c_1|^2 + |c_2|^2 + |c_3|^2 = 1$ .

Two qubit operators exist in a four dimensional Hilbert space (denoted  $\mathcal{H}_4$ ). Therefore, their function can be described by a  $4 \times 4$  square matrix and their state vectors can be described by 4 element vectors.

The most popular two-qubit gate is the *controlled-NOT* gate or simply written as CNOT. As you would suspect, it is an  $X$  gate operating on one qubit but controlled (or activated) by the other. Figure 2.2 shows the quantum circuit for a CNOT operating on qubit 0 and being controlled by qubit 1.

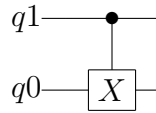


Figure 2.2: CNOT gate drawn using an  $X$  gate

However, it is more commonly drawn as shown in figure 2.3 using a XOR-like symbol noting its similarity to the logical XOR gate.

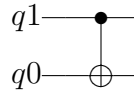


Figure 2.3: Common drawing of a CNOT using XOR symbol

The operations of the CNOT gate on the logic basis states are

$$|00\rangle \rightarrow |00\rangle$$

$$|01\rangle \rightarrow |01\rangle$$

$$|10\rangle \rightarrow |11\rangle$$

$$|11\rangle \rightarrow |10\rangle$$

The unitary matrix describing the CNOT gate is

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Another way of constructing this matrix is by through the outer-product method of summing the outer-products of each input basis state with the desired output state as shown below

$$CNOT = |00\rangle \langle 00| + |01\rangle \langle 01| + |10\rangle \langle 11| + |11\rangle \langle 10|$$

We can generalize and form a two qubit gate by adding a control to any single qubit gate. Also, it is possible the control qubit is on qubit 0 (shown in figure 2.4).

Lastly, figure 2.5 shows the circuit for a general two qubit gate where either or both qubits could be operated on.

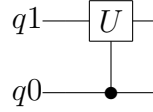


Figure 2.4: General gate operating on qubit 1 and controlled by qubit 0

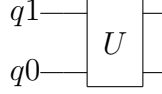


Figure 2.5: General two qubit gate

### Three Qubit Quantum Gates

Three qubit gates have three inputs and three outputs, are reversible, and are described by  $8 \times 8$  square unitary matrices. Common examples of three qubit gates that we'll cover in more detail are the Toffoli and Fredkin gates.

Three qubit gates exist in an 8 dimensional Hilbert space ( $\mathcal{H}_8$ ) with states that can be represented by an 8 element vector. The state can be written as the linear superposition of its 8 computational basis states which we re-write in  $\Sigma$ -form for brevity

$$|\psi\rangle = \sum_{i=0}^7 c_i |i\rangle \quad (2.2.7)$$

Where  $\sum_{i=0}^7 |c_i|^2 = 1$ .

### The Toffoli Gate

The Toffoli gate is also known as the *controlled-CNOT* gate because it is literally a CNOT gate with a secondary control input. The function is if both control inputs are asserted, the target qubit is inverted. The function of the quantum Toffoli gate is the same as its classical counterpart. Figure 2.6 shows a circuit diagram of the quantum Toffoli gate with  $q_0$  being the target qubit and being

controlled by qubits  $q_2$  and  $q_1$ .

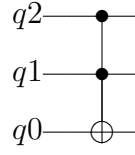


Figure 2.6: Toffoli Gate

From a logical perspective the Toffoli gate performs the function

$$q'_0 = q_0 \oplus (q_2 \text{AND} q_1)$$

and has the following truth table:

$$|000\rangle \rightarrow |000\rangle$$

$$|001\rangle \rightarrow |001\rangle$$

$$|010\rangle \rightarrow |010\rangle$$

$$|011\rangle \rightarrow |011\rangle$$

$$|100\rangle \rightarrow |100\rangle$$

$$|101\rangle \rightarrow |101\rangle$$

$$|110\rangle \rightarrow |111\rangle$$

$$|111\rangle \rightarrow |110\rangle$$

The Toffoli gate can be used to implement a NAND gate by forcing the initial state of qubit 2 to  $|1\rangle$ . Or, it can implement a FANOUT function by forcing the initial state of qubit 2 to  $|0\rangle$ .

The unitary matrix of the Toffoli gate is give by the outer-product method

$$\begin{aligned} \text{Toffoli} = & |000\rangle \langle 000| + |001\rangle \langle 001| + |010\rangle \langle 010| + \\ & |011\rangle \langle 011| + |100\rangle \langle 100| + |101\rangle \langle 101| + \\ & |110\rangle \langle 111| + |111\rangle \langle 110| \end{aligned}$$

Which results in the following  $8 \times 8$  unitary matrix

$$\text{Toffoli} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

We'll learn later in Chapter 4 that we can also describe the unitary matrix of the Toffoli gate as

$$\text{Toffoli} = D_0 \otimes I \otimes I + D_1 \otimes D_0 \otimes I + D_1 \otimes D_1 \otimes X$$

$$\text{Where } D_0 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \text{ and } D_1 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

## The Fredkin Gate

Unlike the Toffoli gate, the Fredkin gate has one control input and two target qubits (those which are operated on depending on the value of the control qubit). The quantum Fredkin gate is reversible, meaning that you are able to determine the inputs based on the outputs. The quantum circuit of a Fredkin gate is shown in figure 2.7 where the control is  $q_0$  and the target qubits are  $q_1$  and  $q_2$ . The Fredkin gate is essentially a controlled-SWAP gate. When the control qubit is asserted, the target qubits are swapped. The boolean function of the Fredkin is as follows

$$q'_0 = q_0$$

$$q'_1 = q_1\bar{q}_0 + q_2q_0$$

$$q'_2 = q_2\bar{q}_0 + q_1q_0$$

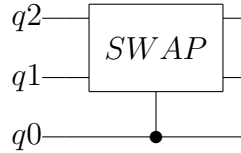


Figure 2.7: Fredkin Gate

The truth table of table of the Fredkin gate using computation basis states

is as follows

$$\begin{aligned}
|000\rangle &\rightarrow |000\rangle \\
|001\rangle &\rightarrow |001\rangle \\
|010\rangle &\rightarrow |010\rangle \\
|011\rangle &\rightarrow |101\rangle \\
|100\rangle &\rightarrow |100\rangle \\
|101\rangle &\rightarrow |011\rangle \\
|110\rangle &\rightarrow |110\rangle \\
|111\rangle &\rightarrow |111\rangle
\end{aligned}$$

Similar to Toffoli and many three-input quantum gates, the Fredkin gate can be configured to behave as other operations. For example, if  $q_2$  is set to  $|0\rangle$ , the result of  $q_2$  becomes an AND operation of  $q_0$  and  $q_1$  (and consequently, the result of  $q_1$  becomes the AND operation of  $q_1$  and  $\bar{q}_0$ ). If we set  $q_1$  to  $|0\rangle$ , the result of  $q_2$  becomes a NOT gate, producing the inverted state of  $q_0$ . Because the Fredkin is able to produce the AND and a NOT functions, it is considered a *universal logic gate*.

The unitary matrix of the Fredkin gate is give by the outer-product method

$$\begin{aligned}
\text{Fredkin} = & |000\rangle \langle 000| + |001\rangle \langle 001| + |010\rangle \langle 010| + \\
& |011\rangle \langle 101| + |100\rangle \langle 100| + |101\rangle \langle 011| + \\
& |110\rangle \langle 110| + |111\rangle \langle 111|
\end{aligned}$$



This results in the following  $8 \times 8$  unitary matrix

$$\text{Fredkin} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

As we did for the Toffoli, the algebraic method described in chapter 4 describes the unitary matrix of the Fredkin gate as

$$\text{Fredkin} = I \otimes I \otimes D_0 + \text{SWAP} \otimes D_1$$

## Universal Quantum Gates

In classical logic, we can construct a universal set of gates (e.g., the NAND and NOR gates) in which all possible boolean functions can be decomposed into just those gates. By this definition, the Toffoli and Fredkin gates are universal gates in a classical sense. However, for quantum logic, a discrete set cannot be used to implement any arbitrary unitary operator. The simplest example is demonstrated by a single qubit who can exist in an infinite number of states on the surface of the Bloch Sphere. The exception are the rotation gates ( $R_x(\theta_x)$ ,  $R_y(\theta_y)$ , and  $R_z(\theta_z)$ ). For our argument, we omit gates which have variables and can label as *dynamic* and instead focus on *static* gates.

Instead, we define a set of quantum gates to be *universal for quantum computation* if any unitary operation can be approximated to an arbitrarily selected level of accuracy using only those gates. Gruska provides us a detailed definition in [32]:

**Definition 8** *A set of quantum gates is **universal** if any unitary transformation  $U$  on any qubit register can be performed, with arbitrary precision  $\varepsilon > 0$ , by a quantum circuit  $C_{U,\varepsilon}$ , consisting solely of the gates from that set. (In other words, the unitary matrix defined by  $C_{U,\varepsilon}$  is  $\varepsilon$ -close to  $U$ .) A quantum gate is **universal** if by itself it forms a universal set when supported by constant input  $|0\rangle$  and  $|1\rangle$ .*

A formal proof is provided by [9] describing how the Hadamard, phase, CNOT, and  $\pi/8$  gates compose a universal quantum set. Thus, any unitary operation can be approximated to an arbitrary accuracy by a quantum circuit using only those gates. Barenco et al in [33] showed that a set gates consisting of all one-bit quantum and CNOT gates are sufficient to be considered a universal quantum set within an arbitrary accuracy.

## No Cloning Theorem

The “No Cloning Theorem” is one of the first fundamental limitations that we see when comparing quantum logic to classical logic and thus deserves attention. In short, it states that we cannot make an identical copy of the quantum state on one qubit to one or more other qubits. In classical logic, *fanout* is a very standard concept. In quantum logic, it is not allowed. This is a particularly important limitation of quantum operators. Because they are linear and unitary operations, unknown quantum states (that is, quantum information that has not

been measured) *cannot* be copied or cloned. This has been extensively analyzed in [9], [34], and [35]. We'll now cover a brief proof by first proposing a unitary gate,  $C$ , which we claim can clone a quantum state. It has two inputs, one being the state,  $|\psi\rangle$ , we wish to clone, and the other an ancillary qubit initially in state  $|0\rangle$  but will become the clone of  $|\psi\rangle$  after the cloning operation. Therefore,

$$C |\psi\rangle |0\rangle = |\psi\rangle |\psi\rangle \quad (2.2.8)$$

We now consider another state,  $|\xi\rangle$ , which is in the superposition state between  $|\psi\rangle$  and another state,  $|\phi\rangle$ ,

$$|\xi\rangle = \frac{1}{\sqrt{2}}(|\psi\rangle + |\phi\rangle)$$

We construct such a superimposed state so that we can test the linearity property of our cloning gate,  $C$ . Therefore, if the cloning operator functions correctly, it should do the following

$$C |\xi\rangle |0\rangle = |\xi\rangle |\xi\rangle$$

Expanding the  $|\xi\rangle |\xi\rangle$  term we get

$$|\xi\rangle |\xi\rangle = \frac{1}{2}(|\psi\rangle |\psi\rangle + |\psi\rangle |\phi\rangle + |\phi\rangle |\psi\rangle + |\phi\rangle |\phi\rangle)$$

However, working out our first equation we get

$$\begin{aligned} C |\xi\rangle |0\rangle &= \frac{1}{\sqrt{2}}[C |\psi\rangle |0\rangle + C |\phi\rangle |0\rangle] \\ &= \frac{1}{\sqrt{2}}(|\psi\rangle |\psi\rangle + |\phi\rangle |\phi\rangle) \end{aligned}$$

Which contradicts our expected result. Therefore, we conclude that no unitary linear operation can copy unknown quantum states. And to be specific, [36] sums it this way: *In summary, non-orthogonal pure quantum state  $\{|\psi_i\rangle\}$  cannot be cloned. This means that no physical system able to carry out the transformation  $|\psi_i\rangle \mapsto |\psi_i\rangle |\psi_i\rangle$  may exist.*

### 2.2.3 Quantum Circuits

Up to this point we have covered single and multi-qubit quantum gates and have seen how they share similarities to their classical counterparts (where they exist). Quantum circuits — similar to their classical Boolean counterparts — are composed of quantum gates connected to form a quantum gate level netlist. We call this quantum netlist a quantum circuit although the term quantum array is also used and means the same thing.

Originally proposed by Deutsch [22], they offer a way of visualizing the function of a quantum computer. We’ve already seen some simple examples of quantum circuits but in this section will go into more detail about how they are formed and read.

Ultimately, the quantum circuit performs and describes the unitary transfer function (or operator) of the quantum computer. In most cases, we label this transfer function as  $U$ . An example three qubit quantum circuit is shown in figure 2.8.

Figure 2.8 contains 6 quantum gates: Three Hadamards, one CNOT, one Pauli-Y, and one Pauli-Z. The quantum circuit is really best expressed as a qubit *time-line*. The lines represent the state of the qubit over time as read from *left-to-right*. We adopt the convention where the least-significant qubit

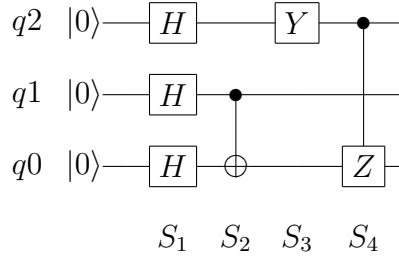


Figure 2.8: Example Three Qubit Quantum Circuit

is qubit-0 (typically written as  $q_0$ ) and placed as the bottom line of the circuit. The most-significant qubit (in this example, qubit-2 or  $q_2$ ) is the top-most line.

Although not typically done, we've placed qubit states at the left-most positions of the time-line in figure 2.8 to denote their initial qubit state being  $|0\rangle$ . Also, quantum circuits can be grouped into *slices* as noted by  $S_1$  through  $S_4$  labels. We go into more format definitions in chapter 4.

The initial state  $|\psi_0\rangle$  is the left-most position on the quantum circuit and in our example is the state

$$|\psi_0\rangle = |q_2\rangle |q_1\rangle |q_0\rangle = |q_2 q_1 q_0\rangle = |000\rangle$$

As we move through the first slice ( $S_1$ ), each qubit state is operated on by a Hadamard gate. At this point, we hit our first example of how to represent the operation of *parallel* gates. As it turns out, the unitary operator of parallel gates can be derived by taking the *Kronecker product* of those gates starting from *top to bottom*. In this case

$$S_1 = H \otimes H \otimes H = H^{\otimes 3}$$

Therefore, we can call  $|\psi_1\rangle$  the state of the qubit register just after being

operated on by  $S_1$ . That is

$$|\psi_1\rangle = S_1 |\psi_0\rangle = H^{\otimes 3} |000\rangle$$

Continuing right-wards with our current state being  $|\psi_1\rangle$ , we hit slice  $S_2$  which performs no operation on  $q_2$ . However, we have a CNOT gate operating on  $q_0$  and controlled by  $q_1$ . We can use our “parallel gates” method of deriving the operator for slice  $S_2$  by noting that it is really two gates in parallel: the identity gate and the CNOT gate. We denote the lack of operation on  $q_2$  by using a  $2 \times 2$  identity gate. From now on, when we encounter a quantum wire, we model it as the identity gate. The CNOT gate is, by definition, a two-qubit operator. Therefore, in parallel they produce

$$S_2 = I \otimes CNOT$$

Thus, the state after  $S_2$  becomes

$$\begin{aligned} |\psi_2\rangle &= S_2 |\psi_1\rangle = S_2 S_1 |\psi_0\rangle \\ &= (I \otimes CNOT) \times (H^{\otimes 3}) |000\rangle \end{aligned}$$

Next, we pass through slice  $S_3$  which is now easy for us to determine its operator as being the Kronecker product of the Pauli-Y gate and two identity gates

$$S_3 = Y \otimes I \otimes I$$

Which brings us to our new state  $|\psi_3\rangle$  after  $S_3$  to be

$$\begin{aligned}
|\psi_3\rangle &= S_3 |\psi_2\rangle \\
&= S_3 S_2 S_1 |\psi_0\rangle \\
&= (Y \otimes I \otimes I)(I \otimes CNOT)(H^{\otimes 3}) |000\rangle
\end{aligned}$$

Lastly, we come to slice  $S_4$  which presents a bit of an oddity. It cannot be described by a simple Kronecker product because the parallelism is broken by having the control from  $q_2$  extending over  $q_1$  to control the Pauli-Z gate operating on  $q_0$ . Chapter 4 is precisely needed to handle these situations with ease. For now, we provide the quantum algebraic equation for  $S_4$  as

$$S_4 = D_0 \otimes I \otimes I + D_1 \otimes I \otimes Z$$

And thus we arrive at our final state,  $|\psi_4\rangle$ , which is represented as the right-most (or end) position on the quantum circuit and is defined as

$$\begin{aligned}
|\psi_4\rangle &= |\psi_{\text{final}}\rangle = S_4 |\psi_3\rangle \\
&= S_4 S_3 S_2 S_1 |\psi_0\rangle \\
&= (D_0 \otimes I \otimes I + D_1 \otimes I \otimes Z)(Y \otimes I \otimes I) \\
&\quad \times (I \otimes CNOT)(H^{\otimes 3}) |000\rangle \\
&= U |000\rangle
\end{aligned}$$

We also note that the unitary operator  $U$  of the quantum circuit is also

called the *transfer function*. In our example,

$$U = S_4 S_3 S_2 S_1$$

In general, to form the transfer function, the slices of the quantum circuit are matrix multiplied in reverse order (i.e., starting from the final right-most slice and continuing to the initial left-most slice).

### Quantum Circuit Identities

Now that we've gone through the exercise of walking through the reading and derivations associated with a quantum circuit, next we want to look at quantum circuit identities.

**Definition 9** A quantum circuit *identity* is defined as a set,  $\Sigma$ , of two or more quantum circuits  $\{C_0, C_1, \dots\}$  who all share the same unitary transfer function,  $U$ .

The simplest analogy to classical logic would be DeMorgan's Theorem ( $AB \Leftrightarrow \overline{A + B}$ ) where one boolean circuit can be replaced with another while still keeping the same functionality. However, just as more complex classical logic identities exist, so do quantum circuit identities. We cover a few of these identities here.

There are more trivial single qubit identities some of which are quite obvious. For example,  $XX$  is simply two quantum NOT gate in series and it makes sense that — just like its classical counterpart — the product of the two inverter matrices produces an identity gate. Figure 2.9 summarizes some common single qubit identities.



Left Circuit	$\Leftrightarrow$	Right Circuit
$\text{---} \boxed{X} \text{---} \boxed{X} \text{---}$	$\Leftrightarrow$	$\text{---}$
$\text{---} \boxed{H} \text{---} \boxed{X} \text{---} \boxed{H} \text{---}$	$\Leftrightarrow$	$\text{---} \boxed{Z} \text{---}$
$\text{---} \boxed{H} \text{---} \boxed{Y} \text{---} \boxed{H} \text{---}$	$\Leftrightarrow$	$\text{---} \boxed{-Y} \text{---}$
$\text{---} \boxed{H} \text{---} \boxed{Z} \text{---} \boxed{H} \text{---}$	$\Leftrightarrow$	$\text{---} \boxed{X} \text{---}$
$\text{---} \boxed{X} \text{---} \boxed{Y} \text{---} \boxed{X} \text{---}$	$\Leftrightarrow$	$\text{---} \boxed{-Y} \text{---}$

Figure 2.9: Common Single Qubit Circuit Identities

The first complex identity is the *Controlled-Z Identity* as shown in figure 2.10. We'll go through a method of showing how the two circuits are equivalent and leave it as an exercise for the reader to prove the the remaining identities.

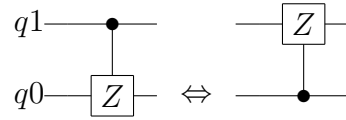


Figure 2.10: Controlled-Z Identity

As you recall, the matrix for the  $Z$  gate is defined as

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Which is formed from the outer-product method

$$Z = |0\rangle \langle 0| - |1\rangle \langle 1|$$

The controlled version of the  $Z$  gate as shown on the left side of figure 2.10 can be constructed through the outer-product method:

$$\begin{aligned}\text{CZ}_0 &= |0\rangle\langle 0| I + |1\rangle\langle 1| Z \\ &= |0\rangle\langle 0| (|0\rangle\langle 0| + |1\rangle\langle 1|) + |1\rangle\langle 1| (|0\rangle\langle 0| - |1\rangle\langle 1|)\end{aligned}$$

For the right-side circuit, we construct the outer-product form to be:

$$\begin{aligned}\text{CZ}_1 &= Z |1\rangle\langle 1| + I |0\rangle\langle 0| \\ &= (|0\rangle\langle 0| - |1\rangle\langle 1|) |1\rangle\langle 1| + (|0\rangle\langle 0| + |1\rangle\langle 1|) |0\rangle\langle 0|\end{aligned}$$

We note that the two equations for  $\text{CZ}_0$  and  $\text{CZ}_1$  can be expanded and collected by obeying the rules of Kronecker products.

Expanding  $\text{CZ}_0$ ,

$$\begin{aligned}\text{CZ}_0 &= |0\rangle\langle 0| |0\rangle\langle 0| + |0\rangle\langle 0| |1\rangle\langle 1| + \\ &\quad |1\rangle\langle 1| |0\rangle\langle 0| - |1\rangle\langle 1| |1\rangle\langle 1|\end{aligned}$$

And  $\text{CZ}_1$ ,

$$\begin{aligned}\text{CZ}_1 &= |0\rangle\langle 0| |1\rangle\langle 1| - |1\rangle\langle 1| |1\rangle\langle 1| + \\ &\quad |0\rangle\langle 0| |0\rangle\langle 0| + |1\rangle\langle 1| |0\rangle\langle 0|\end{aligned}$$

After expanding, we see both have the same terms and thus  $\text{CZ}_0 = \text{CZ}_1$ .

Figure 2.11 summarizes some common two-qubit identities and is by no means complete.

#### 2.2.4 Quantum Algorithms

We briefly overview the primary quantum algorithms discovered which have served as the foundation for continued research in this field of study. We start with description of quantum parallelism and how this characteristic along with entanglement and quantum superposition serve as the underlying basis that are exploited to allow a quantum computer to be more computationally powerful than a classical computer for some problems. We then do a survey of some of the most important quantum algorithms discovered starting with Deutsch's algorithm and its generalization to the Deutsch-Jozsa algorithm, Shor's fast factoring algorithm, and then Grover's search algorithm.

##### Quantum Parallelism

The most exploited property of quantum circuits is that of *quantum parallelism*. To explain how quantum parallelism works we first examine a given function,  $f$ , which has an input  $x$  which is an  $n$ -bit binary string. If we wanted know all the potential values of  $f$  (and without knowing any properties of  $f$  for which to exploit), then we would have two options:

1. Iterate through a single instance of  $f$  for all  $2^n$  input combinations thus taking  $2^n$  time-steps.
2. Have  $2^n$  instances which can work on all possible inputs combinations in a single time-step.

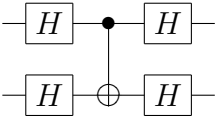

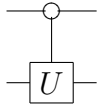
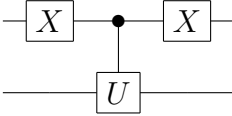
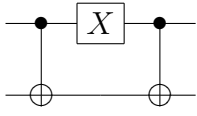
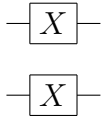
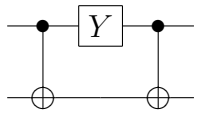
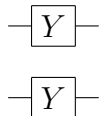
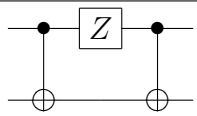
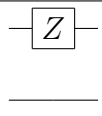
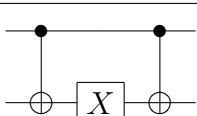
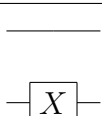
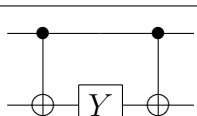
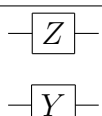
Left Circuit	$\Leftrightarrow$	Right Circuit
	$\Leftrightarrow$	
	$\Leftrightarrow$	
	$\Leftrightarrow$	
	$\Leftrightarrow$	
	$\Leftrightarrow$	
	$\Leftrightarrow$	
	$\Leftrightarrow$	

Figure 2.11: Common Two-Qubit Circuit Identities

This is the limitation that classical logic circuits impose. However, that is not the case for quantum circuits. In fact, the input to a quantum circuit can be a superposition of the logic state  $|0\rangle$  and  $|1\rangle$  and its result would also be a superposition of the function states.

To illustrate, assume we have a quantum circuit described by the unitary operator,  $U_f$  which is a reversible gate which performs the transformation

$$|xy\rangle \mapsto |x\rangle |y \oplus f(x)\rangle \quad (2.2.9)$$

The quantum operator is shown below:

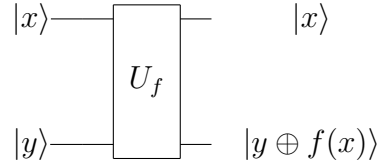


Figure 2.12: Simple two-input reversible quantum operator,  $U_f$

Now we let  $|y\rangle = |0\rangle$  so that the output is  $|x\rangle |f(x)\rangle$ . The function  $f$  has two possible outcomes,  $f(0)$  and  $f(1)$ . Now, we want to put the input  $x$  into a superposition state – something we can't do with classical logic. We do this by setting  $x$  to  $|0\rangle$  and running it through a Hadamard gate. This will give the following:

$$|x\rangle = H |0\rangle = (|0\rangle + |1\rangle)/\sqrt{2} \quad (2.2.10)$$

The result of the operator will be

$$U_f(H |0\rangle) |0\rangle \quad (2.2.11)$$

The function output component will then be

$$f(H|0\rangle) = f((|0\rangle + |1\rangle)/\sqrt{2}) \quad (2.2.12)$$

We can expand the above equation to get

$$(|f(0)\rangle + |f(1)\rangle)/\sqrt{2} \quad (2.2.13)$$

Our output state for the function is now a superposition of the two possible outputs,  $f(0)$  and  $f(1)$ . And the result was done completely in parallel and with a single time-step. This property of quantum parallelism is used in conjunction with superposition (as we did above) and entanglement to implement quantum algorithms. Later, we'll see how quantum interference is used to select a desired solution out of the superposition state.

## Deutsch's Algorithm

Deutsch's algorithm (or Deutsch's problem) was first proposed in 1985 in [21] and had the very significant impact of showing how *interference* and quantum *parallelism* can be exploited.

Referring back to figure 2.12, the problem proposed by David Deutsch was to develop a quantum algorithm that would be able to distinguish if the function  $f(0) = f(1)$  or if  $f(0) \neq f(1)$ . In classical logic, we need two computations for  $f(0)$  and  $f(1)$  so we either need two iterations through a single circuit or two instantiations of  $f$  that calculate in parallel. However, Deutsch was able to construct a quantum circuit that put placed the inputs into a known superposition and would have an output state that represented the logic *xor* of

the functions:  $f(0) \oplus f(1)$ . If  $f(0) = f(1)$ , then this state would be  $|0\rangle$ , always. And conversely, it would be  $|1\rangle$  if  $f(0) \neq f(1)$ . The circuit Deutsch developed is shown in figure 2.13.

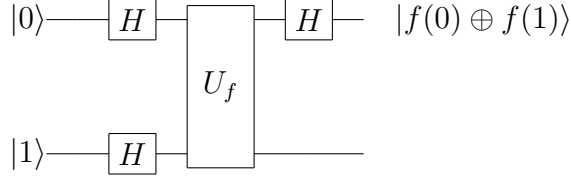


Figure 2.13: Quantum Circuit for Deutsch's Algorithm

We refer to  $x$  as the *control* and  $y$  as the *target*. To feed these inputs, we set  $x$  and  $y$  to a known superimposed state using Hadamards:

$$|x\rangle = H|0\rangle = (|0\rangle + |1\rangle)/\sqrt{2} \quad (2.2.14)$$

$$|y\rangle = H|1\rangle = (|0\rangle - |1\rangle)/\sqrt{2} \quad (2.2.15)$$

The input to  $U_f$  thus becomes  $(H|0\rangle) \otimes (H|1\rangle)$  which is equal to the ket vector we'll call  $\psi_1$ :

$$\psi_1 = \frac{1}{2} \begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \end{pmatrix} \quad (2.2.16)$$

The output after operator  $U_f$  we'll call  $\psi_2$  and is equal to  $U_f|\psi_1\rangle$  which expands into the following two possibilities:

$$|\psi_2\rangle = \begin{cases} \pm \left( \frac{|0\rangle+|1\rangle}{\sqrt{2}} \right) \left( \frac{|0\rangle-|1\rangle}{\sqrt{2}} \right) & \text{if } f(0) = f(1) \\ \pm \left( \frac{|0\rangle-|1\rangle}{\sqrt{2}} \right) \left( \frac{|0\rangle-|1\rangle}{\sqrt{2}} \right) & \text{if } f(0) \neq f(1) \end{cases} \quad (2.2.17)$$

The final output of the circuit which we'll call  $\psi_f$  takes advantage of the fact that  $f(0) \oplus f(1)$  equals 1 if they are not equal and 0 if they are equal. Thus, the final expression for the output state is

$$|\psi_f\rangle = \pm |f(0) \oplus f(1)\rangle \left( \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \quad (2.2.18)$$

Therefore, a *single* measurement of the top qubit will tell us whether the function encoded in  $U_f$  has  $f(0) = f(1)$  or otherwise. All of this utilized only one iteration of  $f$  (encoded in  $U_f$ ) using only one set of inputs which were put into a superposition state.

### The Deutsch–Jozsa Algorithm

Later, Deutsch's problem was extended to a generalized  $n$ -bit solution in 1992 by Deutsch and Jozsa [23]. The extension is such that we want to determine if the function  $f$  which now has input  $x$  which is a binary string of  $n$  bits, is either *balanced* or *constant*. The term *balanced* refers to  $f$  being equal to 1 for half of its inputs and 0 for the other half of its inputs. The term *constant* intuitively refers to  $f$  being either 1 for all of its inputs or, alternatively, 0 for all its inputs. To test this classically would require at worst case  $2^n/2 + 1$  attempts. That is, we may have a stream of  $2^n/2$  constant values (that is, half of the inputs resulted in the same constant function output) but until we test the next input, we won't know if it is balanced or constant.

The circuit for the generalized Deutsch–Jozsa problem is shown in figure 2.14. The input pattern is extended from Deutsch's original algorithm such



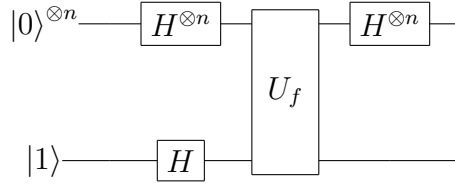


Figure 2.14: Quantum Circuit for the Deutsch-Jozsa Algorithm

that our input state,  $|\psi_0\rangle$ , is now expressed as the superposition state

$$|\psi_0\rangle = |0\rangle^{\otimes n} |1\rangle \quad (2.2.19)$$

The value of the input to operator  $U_f$  thus becomes  $(H^{\otimes(n+1)})|\psi_0\rangle$ , which we call  $|\psi_1\rangle$  and conveniently re-write in the following summation form

$$|\psi_1\rangle = \sum_x \frac{|x\rangle}{\sqrt{2^n}} \left( \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \quad (2.2.20)$$

Where  $x = \{0, 1\}^n$ .

We refer to the top  $n$  qubits as the *query register* and the bottom qubit as the *answer register* (we are being general even though it is a single qubit in this case).

Our input,  $x$ , which is stored in the query register is now an even superposition of all possible input values. The answer register (input to  $y$ ) is an even superposition state. After going through the operator  $U_f$ , the output state,  $|\psi_2\rangle$  takes the form

$$|\psi_2\rangle = \sum_x \frac{(-1)^{f(x)} |x\rangle}{\sqrt{2^n}} \left( \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \quad (2.2.21)$$

After the operation  $U_f$ , the result of the function is stored in the amplitude of the qubits in query register. We now run these through a final Hadamard

register to interfere the qubits in the query register. The result — which we'll call  $|\psi_f\rangle$  — of the output state of the system is given by the following equation

$$|\psi_f\rangle = \sum_z \sum_x \frac{(-1)^{x \cdot z + f(x)} |z\rangle}{\sqrt{2^n}} \left( \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \quad (2.2.22)$$

Where  $z = \{0, 1\}^n$  and  $x \cdot z$  is the bitwise inner-product modulo 2.

In the case where  $f$  is constant, the amplitude for the last qubit will be either  $+1$  or  $-1$ , always. Because  $|\psi_f\rangle$  is of unit length, this must mean that the amplitudes on the query bits must be zero. That is, during a measurement, if all the query bits are 0, then function  $f$  is constant. Conversely, if  $f$  was balanced, then the query bits would have a mix of amplitudes such that their summation was zero. Therefore, a measurement on any qubit in query register would result in a positive or negative value. In this case, just sampling one qubit and in their query register and getting a non-zero value means the function  $f$  is balanced.

## Quantum Fourier Transform

For completeness we review the discrete Quantum Fourier Transform (QFT) as it serves as the basis for key quantum algorithms such as Shor's factoring algorithm as discussed in section 2.2.4. The implementation of the QFT is based on the classical Fast Fourier Transform (FFT). The QFT is a linear operator which performs the following transformation

$$|j\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{i2\pi jk/N} |k\rangle \quad (2.2.23)$$

Where  $|j\rangle$  and  $|k\rangle$  are basis states.

The QFT algorithm maps a quantum state  $|\psi\rangle$  into another state  $|\phi\rangle$  with the following mappings

$$|\psi\rangle = \sum_{j=0}^{N-1} \psi_j |j\rangle \quad (2.2.24)$$

$$|\phi\rangle = \sum_{k=0}^{N-1} \phi_k |k\rangle \quad (2.2.25)$$

Where the amplitudes  $\phi_k$  are the Discrete Fourier Transforms of  $\psi$ . That is,

$$\phi_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \psi_j e^{i2\pi jk/N} \quad (2.2.26)$$

In the case for quantum circuits,  $N$  is of the form:  $N = 2^n$  which allows us to express the integers  $j$  and  $k$  using binary representation

$$j \mapsto j_0 2^{n-1} + j_1 2^{n-2} + \dots + j_{n-1} 2^0 \quad (2.2.27)$$

And

$$k \mapsto k_0 2^{n-1} + k_1 2^{n-2} + \dots + k_{n-1} 2^0 \quad (2.2.28)$$

And through further simplification, we can re-write equation 2.2.23 using our binary representations of  $j$  and  $k$  as well as the property that  $e^{i2\pi jk} = 1$  when  $k = 0$ ,

$$|j_0 j_1 \dots j_{n-1}\rangle \mapsto \frac{1}{2^{n/2}} \bigotimes_{m=0}^{n-1} (|0\rangle + e^{i2\pi j 2^{-m}} |1\rangle) \quad (2.2.29)$$

Which can be further simplified to our final equation for the QFT,

$$|j\rangle \mapsto \frac{1}{2^{n/2}} \left[ \frac{1}{e^{i2\pi(j/2^0)}} \right] \otimes \left[ \frac{1}{e^{i2\pi(j/2^1)}} \right] \otimes \dots \otimes \left[ \frac{1}{e^{i2\pi(j/2^{n-1})}} \right] \quad (2.2.30)$$

To develop a quantum circuit for the QFT we start by introducing the  $R$

gate which has the transfer function

$$R_k = \begin{pmatrix} 1 & 0 \\ 0 & e^{i2\pi/2^k} \end{pmatrix} \quad (2.2.31)$$

For our quantum circuit, the input state is  $|j\rangle$  and the qubits represent the co-factors for the binary expression of  $j$  from equation 2.2.27. For example, the top-most qubit would represent  $j_0$  and the bottom-most qubit would represent  $j_{n-1}$ . With the use of the  $R$  gate and the Hadamard gate we illustrate a general quantum circuit for the QFT in figure 2.15.

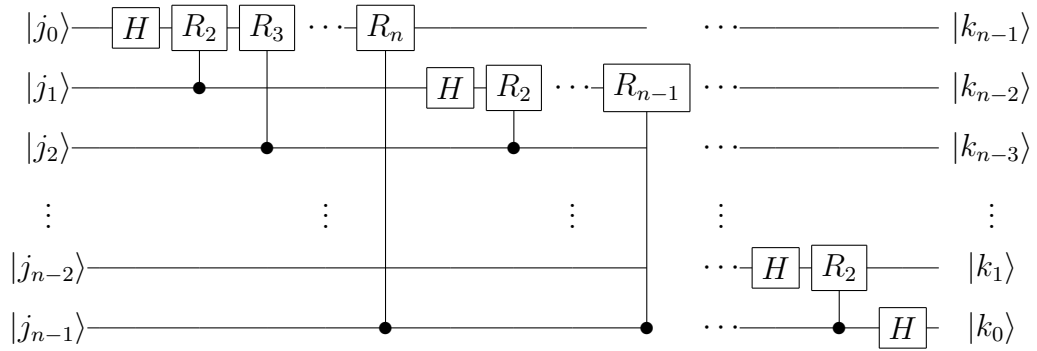


Figure 2.15: Quantum Fourier Transform Circuit

Although not obvious, we explicitly reverse the bits for the  $|k\rangle$  vector to complete the QFT transform.

### Shor's Fast Factoring Algorithm

What is quite possibly the best known application of a quantum computing algorithm is Peter Shor's Fast Factoring Algorithm. First published in 1994 [27] it describes a polynomial time quantum computing algorithm for the factorization of  $n$ -bit integer numbers. There are two reasons for its popular-

ity, the first being that is Shor's algorithm could factor a number  $N$  with a work factor of  $O((\log(N)^3))$  compared to the best known classical algorithm of  $O(e^{c(\log(N))^{1/3}(\log(\log(N)))^{2/3}})$  using the *field sieve* approach. The second reason being the its potential application to cracking the public-key encryption scheme known as *RSA* which relies on the premise that factoring large integers is highly infeasible. In 2001, a team at IBM [37] demonstrated Shor's algorithm with a 7 qubit quantum computer based on NMR technology to factor 15 into 3 and 5 although there is some doubt as to whether their implementation of was truly a quantum computer due to the inability to observe qubit entanglement. More recent experiments using photonic qubits have implemented Shor's algorithm and shown entanglement.

The problem of factorization is to simply defined as: Given a number  $N$ , we can write it as the product of prime numbers,

$$N = p \times q_1 \times q_2 \times \dots \times q_n \tag{2.2.32}$$

The integer  $p$  is called the *proper factor* and satisfies the conditions that it is neither  $N$  nor  $p$  and evenly divides  $N$ .

Shor's factoring algorithm is also significant because it utilizes two primary concepts: phase estimation and QFT. We described QFT in section 2.2.4. In quantum phase estimation, we approximate the eigenvalue associated with an eigenvector of a unitary operator. Many quantum algorithms can be reduced to using phase estimation which utilizes QFT. Shor's integer factorization problem can be shown to reduce to a period finding problem which uses QFT to calculate the period of a function. However, in the construction of Shor's algorithm, we must first represent the problem using phase estimation before using the QFT

algorithm.

Classically, the integer factoring algorithm is performed using the following pseudocode:

- (1) Pick Random integer,  $q$ , such that  $1 < q < N-1$
- (2) Compute  $g = \text{GCD}(q, N)$
- (3) If  $g > 1$ , then return  $g$  (Done), Else (4)
- (4) Determine order,  $r$ , of  $q \text{ MOD } N$  {using period-finding routine}
- (5) If  $r$  is not even, goto (1), else (6)
- (6) Let  $r = 2m$  and determine order  $r$  of  $q \text{ MOD } N$
- (7) If  $1 < p = \text{GCD}(r-1, N) < N$  return  $p$  (Done) else (8)
- (8) If  $1 < p = \text{GCD}(r+1, N) < N$  return  $p$  (Done) else (9)
- (9) Goto (3) {Failed to find proper factor}

Because we are dependent on the value of  $N$ , the quantum circuit implementing the algorithm must take  $N$  into account when being constructed. A qubit register is needed to encode the random variable we called  $q$  in the pseudocode above. This register needs to be sized large enough to hold all the possible values. Given  $N$ , we allot  $n$  qubits such that  $N^2 \leq 2^n < 2N^2$  which allows for the qubits to hold superposition values from 0 to  $2^n - 1$ .

We construct two  $n$ -bit registers, one for input and one for output. When we describe the state of our system, we use the notation:  $|\psi_i\rangle = |a\rangle |b\rangle$  where  $a$  is our *input* register and  $b$  is our output register, each  $n$ -bits.

We now go through the steps of Shor's algorithm:

*Step 1:* Set the initial state of the system for both registers. We'll call this

initial state  $|\psi_0\rangle$  which is defined as

$$|\psi_0\rangle = |0\rangle |0\rangle \quad (2.2.33)$$

*Step 2:* Randomize the input register.

$$|\psi_0\rangle \rightarrow |\psi_1\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle |0\rangle \quad (2.2.34)$$

*Step 3:* Construct unitary function,  $f(k) = q^k \bmod N$ .

$$|\psi_1\rangle \rightarrow |\psi_2\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle |f(x)\rangle \quad (2.2.35)$$

This step uses quantum parallelism to compute  $f(k)$  and store the result in output register.

*Step 4:* Apply the QFT to the input register. This has the effect of distributing the amplitude of any given  $|x\rangle$  state equally among all  $2^n$   $|y\rangle$  states.

$$U_{\text{QFT}} |x\rangle = \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} w^{xy} |y\rangle \quad (2.2.36)$$

Where  $w = e^{i2\pi/2^n}$ . Thus we have the new state

$$|\psi_2\rangle \rightarrow |\psi_3\rangle = \frac{1}{2^n} \sum_{x=0}^{2^n-1} \sum_{y=0}^{2^n-1} w^{xy} |y\rangle |f(x)\rangle \quad (2.2.37)$$

The large speed-up of Shor's algorithm comes from this step where the QFT is used for amplitude amplification (and interference).

Let  $r$  be the period of  $f$ ,  $x_0$  be the smallest value of  $x$  which give the same  $f(x)$  and such that  $x_0 < r$ , and let  $b = \lfloor (2^n - x_0 - 1)/r \rfloor$ . Then we can re-write

the coefficients of our final state as

$$\sum_{f(x)=f(x_0)} w^{xy} = \sum_b w^{(x_0+rb)y} = w^{x_0y} \sum_b w^{rby} \quad (2.2.38)$$

Through interference, amplitudes are constructive when unit vectors defined by  $w^{ryb}$  point in the same direction and destructive when pointing in opposite direction.

*Step 5:* Perform measurement. The measurement is performed by taking the mod-squared of our final state. That is,

$$||\psi_3\rangle|^2 = \frac{1}{2^{2n}} \left| \sum_b w^{(x_0+rb)y} \right|^2 \quad (2.2.39)$$

As the unit vector  $w^{ry}$  approaches the positive real axis, the ratio  $yr/2^n$  approaches being an integer.

*Step 6:* Period Extraction. This is essentially the Continued Fraction Expansion to approximate  $y/2^n$  and derive an approximate ratio  $c/r'$  that satisfies the conditions:

1.  $r' < N$
2.  $|y/2^n - c/r'| < \frac{1}{2}2^n$

Each measurement of the final state  $|\psi_3\rangle$  produces one of the multiples for  $c = y2^n/r$  where each  $y$  is chosen with probability  $1/r$ . After each measurement, we know  $c$  using the relation

$$\frac{c}{2^n} = \frac{y}{r} \quad (2.2.40)$$

Since  $y$  is chosen randomly, the measurement and computation to yield



$\text{GCD}(y, r) = 1$  should only need to be repeated  $O(\log(\log(r)))$  times to be efficiently determined.

## Quantum Oracles

A *Quantum Oracle* is a device that is able recognize solutions to search problem. In general, if we have an  $N$  size search problem with  $M$  possible solutions, we require  $\sqrt{N/M}$  oracle consultations to find the solution. Given an input state which encodes a potential solution, an oracle will signal if it recognizes the solution by flipping an answer bit. We use the word “recognize” because the oracle is not considered to be all-knowing. That is, it doesn’t know all the answers but rather is able to recognize certain criteria of a solution. Although it is possible to construct all-knowing oracles as was done for Deutsch’s problem in section 2.2.4, this would impose a ridiculous requirement of knowing all the solutions to the problem you were trying to solve. Nielsen and Chuang [9] give an appropriate illustration using the prime factoring problem. In the illustration we are given a large number,  $m$ , which is the product of two prime numbers,  $p$  and  $q$ , which we want to determine. Classically, this is a search problem where we would test every number from 2 to  $m^{1/2}$  until we found the first exact dividing number which will be the smaller of the two primes. For the quantum oracle, rather than constructing an oracle which encodes all the solutions, we build an oracle which is able to tell us if the division of  $m$  by some input number,  $x$ , is exact. By using this oracle in a quantum search algorithm, we can speed up the trial division search. Instead of making up to  $m^{1/2}$  trials divisions, we are able to get the solution with roughly  $\sqrt{(m^{1/2})} = m^{1/4}$  trials. Therefore, even though the oracle did not know the prime factors of  $m$ , it was still useful

in developing a quantum search algorithm which could find a solution.

The general structure of a quantum oracle is shown in figure 2.16.

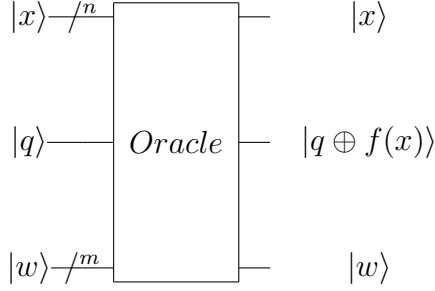


Figure 2.16: Structure of a Quantum Oracle

The oracle input is made up of three basic components:

- The Solution Register,  $|x\rangle$ .
- The Answer (or Oracle) Bit,  $|q\rangle$ .
- Ancilla (or Working) bits,  $|w\rangle$ .

The solution is encoded in the *solution* register, that is,  $|x\rangle$ , which could be many qubits. This is where the encoded tour for a TSP or nodes for a vertex covering problem would be encoded. Obviously we see how Hadamards could be used to provided the initial solution as a superposition of all possible states. The oracle can then be used iteratively — if necessary — to find solutions.

The *answer bit* is what the oracle will actually modify and output if the criteria for being a valid solution is correct. The function performed by the oracle to modify this qubit is defined as

$$|x\rangle |q\rangle |w\rangle \mapsto |x\rangle |q \oplus f(x)\rangle |w\rangle \quad (2.2.41)$$

Where,

$$f(x) = \begin{cases} 1, & \text{if } x \text{ is a solution} \\ 0, & \text{if } x \text{ is not a solution} \end{cases} \quad (2.2.42)$$

The operation  $q \oplus f(x)$  causes the oracle to effectively bit-flip the qubit if it is a valid solution. To be more specific, the term bit-flip shouldn't be confused with an inversion. Rather, the bit-flip process “negates” the qubit by inverting its “phase” and not its value. As an example, for the *independent set problem* (ISP), the oracle would flip the answer qubit if there was an independent set of greater than or equal to  $k$  nodes.

The *ancilla* or working bits are there to provide the qubit space for the quantum algorithm to do its work. For example, the working bit register could encode the valid solution criteria. As a very simple example, we could ask the oracle to determine if  $|x\rangle = |w\rangle$ . In a more sophisticated example, the working bit register could encode the target tour length for a TSP problem or minimum number of nodes for an ISP problem.

Being able to formulate a problem as a quantum oracle which can validate a correct solution provides the potential to create quantum algorithms. Some of these include graph coloring, satisfiability, set covering, Euler and Hamiltonian path finding, and search problems.

### **Grover's Algorithm**

Grover's algorithm [28] implements an exhaustive search algorithm for quantum computers which has tremendous (as in “quadratic”) speed-up over its classical counterparts. It is meant for “needle-in-a-haystack” like problems where we are trying to find  $t$  needles amongst  $N$  strands of hay and can do so in approximately

$O(\sqrt{N/t})$  iterations as compared to a classical algorithm which would take an average  $O(N/(2t))$  iterations. Grover's algorithm uses a combination of Hadamard gates, a quantum oracle (as reviewed in section 2.2.4, and other gates in an iterative manner to search a large space for solutions.

Grover's algorithm is an iterative process. We refer to the fundamental operator being iterated as the *Grover Iterate*. We also note that the number of iterations is dependent on both the problem size,  $N$ , and the number of solutions,  $t$ . We will discuss this in more detail.

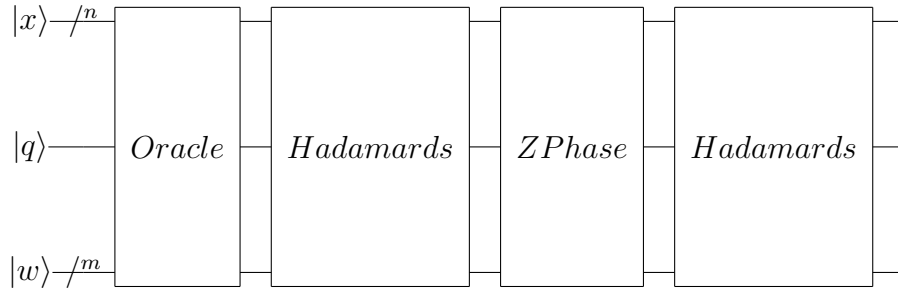


Figure 2.17: The Grover Iterate: HZHO

The typical components of a Grover Iterate are shown in figure 2.17. The current state is applied to the oracle, then a set of Hadamards, a zero-state shift operation, and another set of Hadamards. We sometimes refer to this operation as “HZHO” which are the operations in matrix multiplication order.

As we stated before, the oracle will bit-flip solution(s):

$$|x\rangle |q\rangle \mapsto |x\rangle |q \oplus f(x)\rangle \quad (2.2.43)$$

Which can be re-written as:

$$|x\rangle |q\rangle \mapsto (-1)^{f(x)} |x\rangle |q\rangle \quad (2.2.44)$$

NOTE: We've omitted  $|w\rangle$  for simplicity.

All solutions which meet the criteria of the oracle will be flipped. This is done with the *Zero-State Phase Shift Circuit* which we call  $Z$  in the Grover Iterate. The function of this gate is to flip the oracle qubit (that is, the answer qubit) when all qubits are zero. Figure 2.18 shows the general structure for this gate as a  $Z$  gate operating on the answer bit,  $|q\rangle$ , flipping its phase when the oracle solution qubits,  $|x\rangle$ , are  $|0\rangle$ .

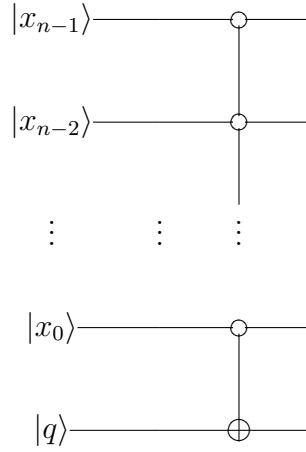


Figure 2.18: Zero State Phase Shift Circuit

Using our algebraic method described in chapter 4, we can describe the functionality of the Zero-Phase shift circuit as:

$$Z = I^{\otimes n} \otimes I - D_0^{\otimes n} \otimes I + D_0^{\otimes n} \otimes X \quad (2.2.45)$$

Grover's algorithm follows the following basic steps:

1. Initialize  $|x\rangle$ ,  $|q\rangle$ , and  $|w\rangle$ .
2. Apply Grover Iterate for desired number of iterations.
3. Measure  $|x\rangle$  to obtain the solution.

We want the solution,  $|x\rangle$  to initially be an equal superposition of all states and of course the standard method is to initially set  $|x\rangle = |00\dots 0\rangle$  and then operate on them with Hadamards. Thus, for  $x$ , we have the following state before entering the first Grover iterate,

$$|x\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle \quad (2.2.46)$$

The answer qubit (sometimes referred to as a reference qubit) is typically set to the superposition state,

$$|q\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \quad (2.2.47)$$

We call the Grover Iterate as shown in figure 2.17,  $G$ . In that case, the general topology for Grover's algorithm is shown in figure 2.19.

The total number of Grover Iterates is determined by knowing the total number of states,  $N$ , and total number of valid solutions,  $t$ . Knowing that, the number of Grover iterations is — as previously stated —  $O(\sqrt{N/(2t)})$ . More approximately, if we had one solution ( $t = 1$ ), then we need  $\lceil \pi\sqrt{N}/4 \rceil$  iterations. In the event we have  $t$  valid solutions out of  $N$  candidates, then the number of iterations is approximately  $\lceil \pi\sqrt{(N/t)}/4 \rceil$ .

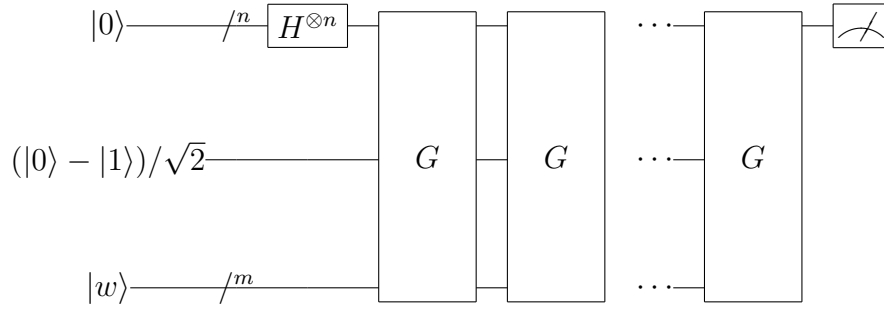


Figure 2.19: Grover's Algorithm

A great analysis of how Grover's algorithm behaves from a geometrical perspective is given by Boyer et al in [38]. Effectively, we assume we start with some initial solution  $|\psi_0\rangle$  on a 2D plane which has some complex amplitude and distant from the desired target amplitude of 1.0 as shown in figure 2.20. Initially, it has a slight angle,  $\theta$ , which is determined by the fact it will be one of  $N/t$  solutions that exists with an equal probability amongst all  $N$  states since we *randomized* our input solution using Hadamards.

With each Grover iteration, we attenuate the non-solutions and effectively rotate the desired solution closer to the  $|u\rangle$  axis. In the case where there are  $t$  solutions we can determine the rotation step size (and initial angle),  $\theta$ , as given by,

$$\theta = 2 \arcsin(\sqrt{t/N}) \quad (2.2.48)$$

Thus, we want to determine the correct number of iterations,  $m$ , such that we will land as close to the ideal solution as possible so that a measurement will yield the correct state with high confidence. We do this by assuming  $t \ll N$

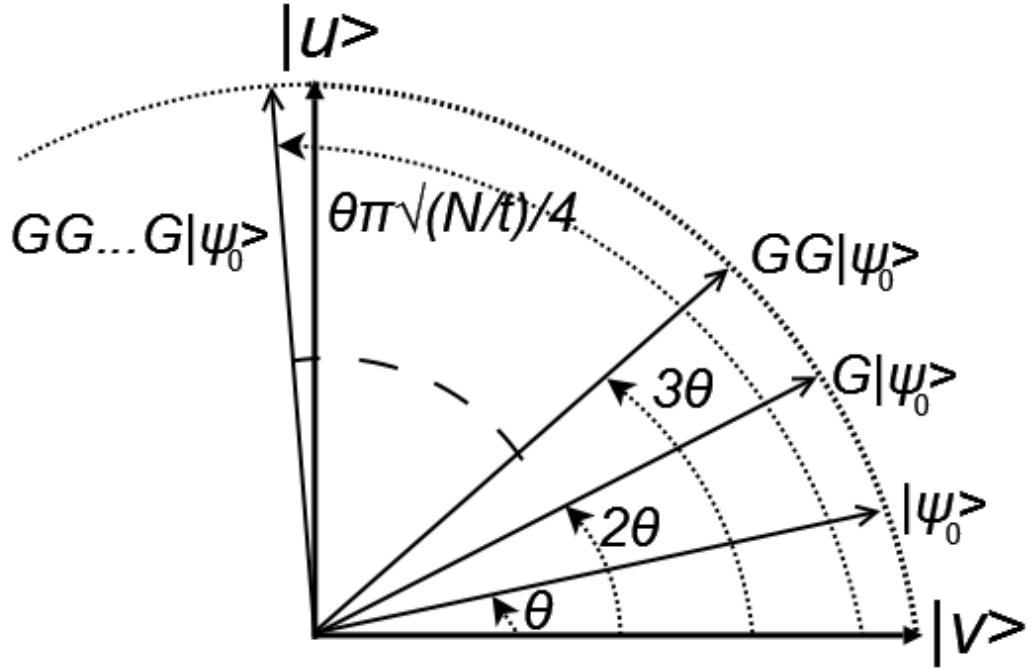


Figure 2.20: Grover Iterations rotate an initial solution to the desired solution and through approximation we derive,

$$m \approx \pi \sqrt{(N/t)}/4 \quad (2.2.49)$$

We note that further Grover iterations do not continue to hone-in on the target solution. Referring back to figure 2.20, more iterations would actually continue to rotate the vector past the target solution and reduce the probability of measuring the correct solution.

### 2.3 Evolutionary Algorithms

All evolutionary algorithms (EAs) share the same basic organization: iterations of competitive selection and random variation. Although there are several varieties of EAs, they are all biologically inspired and generally follow the format



depicted in Figure 2.21.

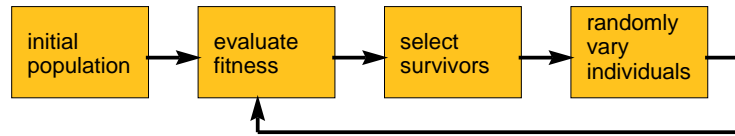


Figure 2.21: The canonical EA

EAs manipulate a population of individuals where each distinct individual encodes a unique set of problem parameters needed to form a solution. The initial population is randomly generated. During each generation (iteration), the current population is evaluated and each individual is assigned a numerical fitness value. High fitness means the associated individual represents a good solution to the given problem. The selection process chooses the higher fit individuals for reproduction. These survivors undergo stochastic reproduction operations to create new individuals. The loop shown in Figure 2.21 continues until either a fixed number of generations are processed or an acceptable solution has been found.

The three common EA paradigms used for optimization are the genetic algorithm (GA), the evolution strategy (ES), and evolutionary programming (EP). Each paradigm was independently developed. Although they all follow the evolving population model, there are some differences. For example, the GA chooses parents with a probability proportional to its fitness with respect to other individuals in the current population. This means the fitter parents are more frequently chosen for reproduction. Conversely, ES and EP allow every parent to reproduce regardless of its relative fitness. ES ranks all parents and offspring according to fitness and deterministically chooses the best individuals to be parents in the next generation; EP conducts a tournament among all

individuals and the tournament outcome determines who survives. Most importantly, GAs use components from two parents to produce offspring—a process called recombination—as the primary reproduction operator, whereas EP only uses mutation. The ES can use multi-parent recombination, but it relies heavily on mutation for reproduction.

### 2.3.1 Elements of an Evolutionary Algorithm

A general *Evolutionary Algorithm* can be broken down into a common set of stages as indicated in figure 2.21 which we discuss in detail. The primary components of an EA are listed below. In section 2.3.2 we'll see how these components are defined more specifically for an Evolution Strategy.

- Individuals and Representation
- Fitness Function
- Population
- Parent Selection
- Recombination
- Mutation
- Survivor Selection

#### Individuals and Representation

The goal of an EA is to find the solution to a problem through large exploration of the solution space. An *individual* represents a possible solution to the

problem. How the individual can be converted to form the solution is called its *representation*.

We use the term *phenotype* to refer to the solution as presented by the individual. This could be, for example, a set of integer values. The representation determines how that set of integers is mapped to an actual solution to the problem being solved. As an example, a weighted summation of the integer values could form the representation of the individual.

An individual has a particular encoding for its phenotype. This could literally be anything from binary numbers, strings, floating point values, or more complicated structures. The particular encoding of an individual is referred to as its *genotype*. Continuing with our example, if the phenotype were sets of integers, the genotype (or encoding) could be bit-strings which encode the integer values.

It is noted that some problems lend themselves in such a way that the phenotype and genotype could be equivalent. For example, the genotype could be a set of integers and it just so happens that the phenotype is, as well. Thus, the mapping is trivial. Furthermore, the phenotype itself could be used directly in the representation.

## **Fitness Function**

The *fitness function* (sometimes referred to as the *evaluation function*) is a critical aspects of an EA. This is what determines the fitness of an individual — in other words — how good of a solution is represented by an individual. In an EA, individuals are compared against each other using the fitness function and it is the fundamental method that drives the decision making process of any

EA. Ultimately, the individual with the best fitness is the best solution found by the EA.

Fitness functions typically derive a fitness value based on the encodings of the individual. For example, an individual could encode the tour for a traveling salesman problem. In that case, the fitness function could determine the length of the tour using a distance map. The fitness would be calculated in such a way that shorter tours have a higher fitness. For example, if the tour length is calculated by a function  $L$  which operates on an arbitrary individual,  $I$ , then a possible fitness function for this problem could be:

$$\text{fitness}(I) = \frac{1}{1 + L(I)} \quad (2.3.1)$$

It is completely up to the designer to determine a suitable fitness function for an EA. Sometimes it is sufficient that it accurately determines whether one individual is better than another. However, other times the fitness function can play a vital role to avoiding getting solutions stuck in a highly non-optimal local minima. A fitness function need not be a simple function. In fact, it could be statistically generated, a database look-up, or a sophisticated function with many corner-cases.

It is also common in optimization problems to refer to an *objective function*. There is usually a simple mapping from a fitness function to an objective function and are often used interchangeably.

## Population and Generations

The *population* is the set of individuals which encode possible solutions to the problem our EA is trying to solve. A population is also defined for a certain

*generation*. A generation is a time measure and specifically refers to the unit of time when the EA is considering a specific population of individuals. In the first generation we have the initial population. In the second generation, we will most likely have a different population made up of the children from the previous population and possibly some of the parents from the previous population that are carried over. Likewise for the third generation and so on and so forth.

### **Parent Selection**

At the end of a generation, parents are selected from the current population and combined (or mated) to form the children for the next generation. The *parent selection* process is also extremely important for an EA. Fundamentally, healthy (that is, highly “fit”) parents are chose to mate with each other in hopes they will produce highly fit children. However, care must be taken to make sure a measure of diversity is maintained for the typical issue of avoiding poor local minima. For this reason, lower-quality parents are chosen with some probability for the mating process to encourage diversity.

### **Recombination**

Once parents have been selected, they are combined together to form one or more children. In mimicking biology, the genes of the children are chosen from the parents. The process of how the genes from the parents are combined to form the genes of the children is known as *recombination* or sometimes *crossover*. The exact method of how this happens can vary. However, a set of recombination operators are applied on any number of parents to produce any number

of offspring. Common examples are *intermediate* and *discrete* recombination which are useful for integer representations. In discrete recombination, alleles for specific genes are chosen randomly between the parents for each child. An *allele* is essential a valid value for a particular gene. In intermediate recombination, the alleles for specific genes are weighted sums from the parents. After recombination, there is usually some measure of “error checking” to make sure legal children are being produced.

## **Mutation**

Once children have been produced through recombination, the next step is *mutation*. Just as in biology, mutation is the effect of randomly altering genes in the hope that the perturbation will produce a superior individual. A set of mutation operators are defined for a particular EA to work on a particular individual. For example, if genes were represented as integers, then a mutation operator could produce a random increment or decrement which could be applied to one or more (but not necessarily all) of the genes. Just as with recombination, we make sure the mutated offspring are still legal solutions. Sometimes this means placing bounds (for example, the maximum and minimum value of the integer).

Although mutation is one of the primary methods of exploring the solution space, it could be skipped and left solely to recombination. Likewise, recombination could be skipped and mutation is the sole method of deriving children (in such cases, there is only a single parent of which the child is a clone of and then mutated).

## Survivor Selection

Once the parents have been selected, mated, and the children have been formed, we then come to the process of *survivor selection*. That is, determining which individuals will be selected to form the population for the next generation. Survivor selection can be similar to parent selection and survivors are chosen in large part by their fitness. However, some EA paradigms implement survivor selection as *replacement*, where the old parents are replaced by the fittest offspring. Other EAs allow the older parents to participate in the selection process and thus potentially live into the next generation. This method is especially useful if one of the older generation parents represents a very good solution and we want their genetics to influence future generations. However, this also can lead to extensive influence through genetic drift and can drive the population towards the local minima represented by the most fit parents — a property that could be good or bad. Whereas, by not keeping the parents around, we allow more diverse children which might have the effect of jumping out of local minima and perhaps find better solutions. A method known as *elitism* allows the most fit individual (or individuals) to be part of the selection process.

For most EAs, there is a set number of parents and a set number of children and an overall set population size. It is quite common for a smaller set of parents create a larger set of offspring and that the set of individuals to be selected from be larger than the survivor population. The primary driving force is to encourage diversity while still selecting highly fit individuals.

## Initialization

In an EA, the first population (also called the *initial population*) is typically randomly generated although care should be taken to make sure that the population is diverse and actually contains legal solutions whether they are good or bad. We start the first generation with our initial population.

It can be potentially useful to ensure through other (possibly deterministic) methods the initial population is actually composed of valid and maybe even relatively fit solutions. However, this also may have the adverse effect of driving your individuals towards poor local minima, potentially avoiding better solutions. Sometimes the extra computation to produce a more fit initial population costs more overall than to have started with a simple initial population.

## Termination Condition

The *termination condition* of an EA is the criteria needed to be met to stop the EA. The criteria are generally broken down to two cases: 1) an individual was found that was reached or surpassed the desired fitness level (that is, we found an acceptable solution), or 2) the EA did not find a solution after a certain number of generations or run-time. In many optimization problems we may not know how good a sufficient solution should be. For example, in the travelling salesman problem, if we were looking for the shortest tour, how do we know what a good solution is? Had we known a desired target tour distance, we could use that knowledge in the termination condition. In case where we cannot adequately derive a target fitness we simply let the EA run to a set number of generations based on run-time or possibly terminate based on a convergence characteristic (e.g., if a better solution hasn't been found in  $x$  generations).



### 2.3.2 Evolution Strategies

An *Evolution Strategy* (ES) is a type of evolutionary algorithm invented in the 1960s by Rachenberg and Schwefel [39]. The main contribution of an ES is that of *self-adaptation*. That is, parameters are introduced to aid in the evolution process which are considered part of the chromosomes of an individual and go through their own evolution process.

The primary characteristics of an ES are:

1. Individuals are represented by real-valued vectors.
2. Mutation is the primary method of producing children.
3. Strategy parameters are implemented and adapted to aid in guiding mutation.

Because of these characteristics, Evolution Strategies are typically used for continuous parameter optimization problems.

An ES represents the size of its parent population using  $\mu$  and the size of its child population as  $\lambda$ .

The basic flow of an ES follows is the same as figure 2.21. We'll now go through the primary components of an ES in more detail.

#### Representation

An ES represents individuals using a vector of real-valued vectors. The genotype of an ES is composed of a two distinct sets: *objective parameters* and *strategy parameters*:

$$\langle \vec{x}, \vec{\sigma} \rangle = \langle x_0, \dots, x_n, \sigma_0, \dots, \sigma_m \rangle$$

We refer to  $\vec{x}$  as the *object parameters* which represent the traditional representation of an EA. These are real-valued vectors. Whereas,  $\vec{\sigma}$  are the *strategy parameters* which are special to an ES are used for self-adaptation.

The number of the strategy parameters is typically one of two cases. In a *single strategy* ES, there is only one single strategy parameter (that is,  $\vec{\sigma} = \langle \sigma_0 \rangle$ ) which is used to adapt all of the object parameters. In an *N-Strategy* or *Multi-Strategy* ES, there are  $N$  strategy parameters which have a 1-to-1 mapping to the  $N$  objective parameters. That is,  $\sigma_0$  is the strategy parameter for object parameter  $x_0$ , and  $\sigma_1$  for  $x_1$ , and so on and so forth.

## Mutation

Mutation is the primary method for the creation of offspring and is based on a normal (Gaussian) distribution. This is perhaps the most complex aspect of an ES. Each object parameter is mutated using:

$$x'_i = x_i + N(0, \sigma) \quad (2.3.2)$$

Where  $\sigma$  in this case is the standard deviation and  $N(0, \sigma)$  refers to a random number generated from a *normal* distribution with zero mean and standard deviation  $\sigma$ . It just so happens that our strategy parameters represent the standard deviations. In the single strategy schema, we have one strategy parameter which is the standard deviation used for all object parameters. In the multi-strategy schema, we use the following equation:

$$x'_i = x_i + N(0, \sigma_i) \quad (2.3.3)$$

For mutations, we consider only the case of *uncorrelated mutations* such that there is one or  $N$  strategy parameters which are *independent* of each other. In the multi-strategy case, we have a standard deviation for each axis (that is, the axis of each object parameter) but the strategy parameters (or even the object parameters) do not influence each other directly.

The strategy parameter can be thought of as a *step size* used to increment the object parameter. The strategy parameter is mutated along with the object parameters. In particular, we first mutate the strategy parameters and then mutate the object parameters based on the new strategy parameters.

In the single strategy case, the strategy parameter is mutated by multiplying itself with a term  $e^\xi$  where  $\xi$  is a random variable drawn from a normal distribution with zero mean and standard deviate  $\tau$ .

$$\sigma' = \sigma e^{\tau N(0,1)} \quad (2.3.4)$$

$$x'_i = x_i + \sigma' N_i(0, 1) \quad (2.3.5)$$

Note the use of the identity  $N(0, \sigma) = \sigma N(0, 1)$  above. The  $N_i$  is used to denote that we draw a new random number from the normal distribution for each object parameter.

The standard deviation  $\tau$  is also referred to as the *proportionality constant* and is inversely proportional to the square root of  $n$  where  $n$  is the size of the problem. Typically the size of the problem is the number of object parameters.

$$\tau \propto \frac{1}{\sqrt{n}} \quad (2.3.6)$$

In the multi-strategy case, we have a  $\sigma$  for each  $x$ . This has a significant

effect on how we mutate our strategy parameters. We still apply the same proportionality constant,  $\tau$ , but we introduce another standard deviation which is applied per strategy parameter. Our multi-strategy mutation equations become:

$$\sigma'_i = \sigma_i e^{\tau N(0,1) + \tau' N_i(0,1)} \quad (2.3.7)$$

$$x'_i = x_i + \sigma' N_i(0,1) \quad (2.3.8)$$

Notice the extra standard deviation term for  $\sigma'$  we denote  $\tau'$  which is defined as:

$$\tau' \propto \frac{1}{\sqrt{2\sqrt{n}}} \quad (2.3.9)$$

The primary advantage of the multi-strategy schema over the single strategy is that it provides a separate self-adapting step-size parameter to each object parameter. This allows the range of object parameters to be more varied, that is, if we had a variety of object parameters and some wanted large step-size while others wanted small step-sizes (either because of their definitions or because a particular object parameter is at a near optimal value), then separate strategy parameters could adapt to fit those the step-size increments.

As mutation progresses and we see individuals becoming more fit, the strategy parameters will begin to relax to a state close to zero. This smaller step-size causes object parameters to change less and the individual to converge. It then becomes important to bound our strategy parameters so that they do not produce too big a step-size or produce a new  $\sigma$  that is so close to zero that it would take numerous multiplication of random values greater than unity to bring it to a step-size that actually represents a sufficient delta for an object parameter.

Obviously if  $\sigma = 0$  then no mutation can occur. Therefore, we typically impose a bound:

$$\epsilon_o < \sigma' < \Sigma \quad (2.3.10)$$

Where  $\epsilon_o$  represents the lower-bound value of  $\sigma'$  and  $\Sigma$  represents the upper-bound. If during mutation we break the boundary condition we typically set it equal to the particular boundary point. For example, if  $\sigma' < \epsilon_o$  then set  $\sigma' = \epsilon_o$ .

Individuals which have converged to their most fit state will have strategy parameters at their minimum allowed boundary (that is, strategy parameters near  $\epsilon_o$ ).

As part of the initialization of an individual, initial  $\sigma$  values are chosen at random. Obviously we want the  $\sigma$ 's to be within their bounds. However, care is also taken to make sure the  $\sigma$ 's are not too small (which could result in step-sizes so small that converge takes too long) or too large (such that object parameters vary so large that we enter we never converge). Although we want highly fit individuals to converge to a low  $\sigma$ , we also want to avoid premature convergence. For this reason, when the minimum  $\sigma$  value ( $\epsilon_o$ ) is reached, it may be beneficial to reset the  $\sigma$  to a higher value rather than setting it to  $\epsilon_o$ . We will sometimes refer to this as the *reset*  $\sigma$ . This takes cues from simulated annealing where the temperature may be increased to encourage the state to jump out of a potential local minima.

## Recombination

Recombination in an ES uses two or more parents combined together to form one child using either discrete or intermediate recombination. Another method called *global recombination* allows using any random parent for each gene and

thus a large number of parents could be used to create a child. Although we gave broad coverage of the terms *discrete* and *intermediate* recombination, for an ES they are more firmly defined as real-valued operators such that:

$$z_i = \begin{cases} (x_i + y_i)/2 & \text{if intermediate} \\ \text{Randomly choose } x_i \text{ or } y_i & \text{if discrete} \end{cases} \quad (2.3.11)$$

### Parent Selection

Parent selection is quite simple which is one of the attractions of an ES. Parents are selected from the parent population (denoted sometimes by  $\mu$ ) using a normal distribution. That is, the fitness of the parent does not bias the decision.

### Survivor Selection

There are two primary methods for survivor selection in an ES. These are known as  $(\mu, \lambda)$  and  $(\mu + \lambda)$ . The notation will become obvious. In the  $(\mu, \lambda)$  method, the selection is made from only the child population,  $\lambda$ . In the  $(\mu + \lambda)$  method, the selection is made from both the parent and offspring population. In both cases, the selection population is sorted by fitness and the top  $\mu$  individuals are chosen as the parents for the next generation.

The reasoning behind the two methods can be summarized this way. In  $(\mu, \lambda)$  selection, we omit the parents which granted might drop a very fit individuals but hopefully their impact to the genetics of the offspring is somewhat carried through and possible reduce the direct attraction of being pulled into a poor local minima. Obviously the concern is losing extremely fit parents and possibly never arriving at similarly fit solutions for a long time if ever. In  $(\mu + \lambda)$  selection, we maintain the healthiest parents but the obvious concern is that fit parents

that represent local minima will stay around from generation to generation, influencing children more and more to look more like the parents and also center on their local minima. In such cases, the probability of a significant mutation is reduced because of pre-mature convergence (that is, strategy parameters are shrinking) and the low probability of a large enough mutation that finds an individual in a deep enough local (or even global) minima to be considered in the top  $\mu$  most fit individuals (and stay there long enough to make an impact).

## Chapter 3

### Related Research

#### 3.1 Evolutionary Algorithms in Quantum Computing

Designing or synthesizing quantum logic has traditionally been researched using deterministic and analytic approaches. As an example, we describe the CSD method as used by the popular *Qubiter* [3] in section 3.4.4. Much of the work cited in 3.4 is largely mathematically formulated methods to solving quantum logic problems. However, these methods tend to be focused on either a small number of qubits or pigeon-holed to very particular problem criteria. These have provided very powerful devices and understanding to deal with quantum logic synthesis but general solutions are still lacking.

The application of Evolutionary Algorithms (EA's) to quantum computing is an emerging research area [40, 41, 4, 42, 43, 44, 45, 46]. The reasons for using an EA for quantum logic design are quite obvious. The configuration space for quantum logic design is huge and EA's provide the ability to search such large spaces. Although EA's provide parameters to guide searching, it is also the ability to have a minimal set of parameters to allow searching that gives EA's an advantage. An analytical method could be focused on a specific



family of problems but the EA is free to explore as long as it doesn't violate certain criteria. As we reviewed in chapter 5, having a very rudimentary fitness function common to EA's is many times all that is needed to find highly optimal solutions. Another advantage is that EA's can lead to discoveries, for example, families of gates more optimal than previous analytical methods discovered [40].

There are many types of Evolutionary Algorithms, but the bulk of the research of applying EA's to quantum logic design problems have largely been focused on two types of EA's: 1) Genetic Algorithms and 2) Genetic Programming. Examples applications of Genetic Algorithms to quantum computing can be found in [43, 44, 46, 47, 48, 49, 50, 51, 52, 53]. Examples of Genetic Programming to quantum computing can be found in [42, 45, 54, 55, 56, 57].

A new application of using an Evolution Strategy (ES) to quantum computing was first proposed by Hutsell and Greenwood in [4] and is the source for the material in chapter 5.

### **3.1.1 Application of a Genetic Algorithm for Quantum Logic Synthesis**

For this section we briefly review an example GA<sup>1</sup> application for Quantum Logic synthesis as published by our colleague, Dr. Martin Lukac, in [40]. A general overview of EA's can be found in section 2.3.

#### **Individual Representation**

In the GA, an *individual* represents a valid quantum circuit that encompasses as many qubits as needed for the overall quantum array. In particular, an

---

<sup>1</sup>For a comprehensive background in Genetic Algorithms, consult [58] as well as [59]

individual is partitioned into parallel blocks. This is largely in part to allow the representation of the overall unitary operator of the individual using only Kronecker products.

The encoding scheme (that is, the chromosomes) used for an individual is an integer number which indicates the number of qubits encoded followed by a string of characters which map to a quantum gate from a given quantum gate library. The ‘r’ character is reserved as a special tag to indicate where a block begins and ends to allow blocks that operate on a varying number of qubits. In particular, the characters between a set of ‘r’ characters define a slice of logic whose quantum unitary operator is simply the Kronecker product of those gates in their order specified. An example encoding and how it relates to a partitioned circuit is shown in figure 3.1.

Figure 3.1 shows the string encoding on the left describing a 5-qubit quantum circuit (as indicated by the integer in the first position) with three slices: S1, S2, and S3, as partitioned by the ‘r’ character tag. The “SW” gate refers to a 2-qubit SWAP operation which is used in lieu of the representation of control operations that span multiple qubit levels. The  $T$  gate refers to the Toffoli gate which operates on three qubits.

The unitary operations can be easily written as

$$S_1 = I \otimes SW \otimes I \otimes I \quad (3.1.1)$$

$$S_2 = H \otimes I \otimes T \quad (3.1.2)$$

$$S_3 = I \otimes SW \otimes I \otimes I \quad (3.1.3)$$

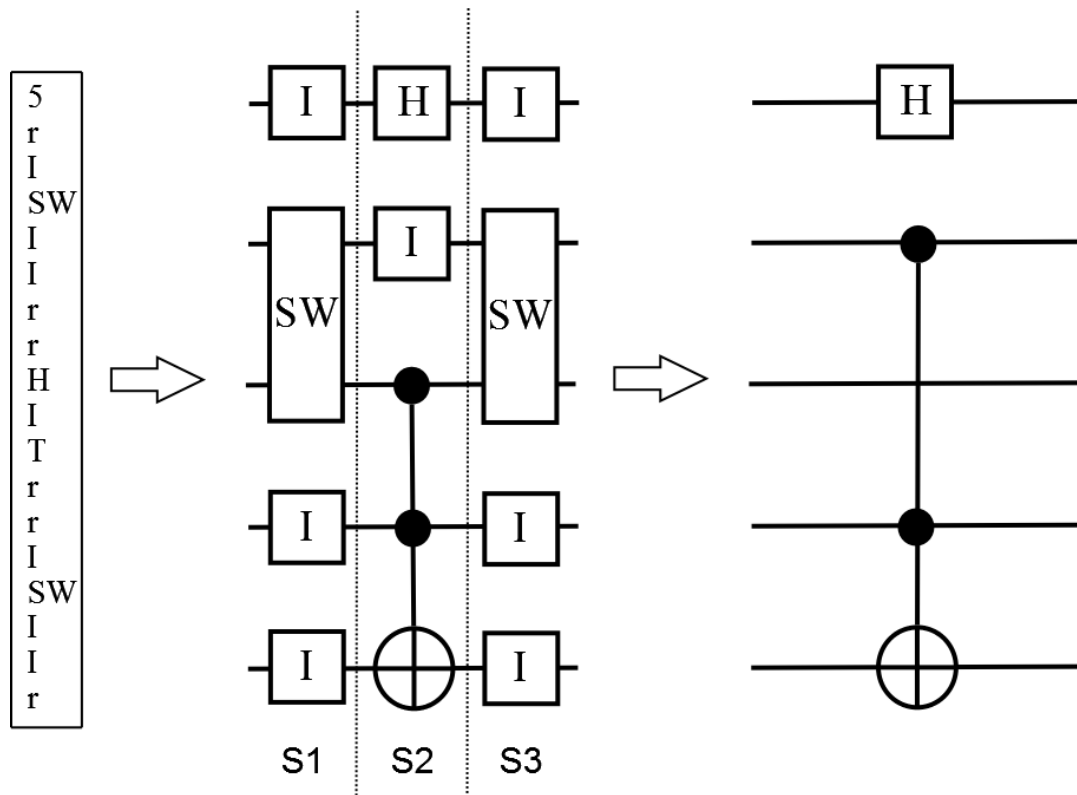


Figure 3.1: Example GA Encoding for 5 Qubit Individual

Thus, the overall operation of this individual,  $U$ , is described as

$$U = S_3 S_2 S_1 \quad (3.1.4)$$

This circuit maps to the right-most circuit in figure 3.1 which we are able to describe using our quantum algebraic notation from chapter 4 as,

$$U = H \otimes D_0 \otimes I \otimes I \otimes I + H \otimes D_1 \otimes I \otimes D_0 \otimes I + H \otimes D_1 \otimes I \otimes D_1 \otimes X \quad (3.1.5)$$

Or using the alternative expression (equation 4.5.13),

$$U = H \otimes I \otimes I \otimes I \otimes I - H \otimes D_1 \otimes I \otimes D_1 \otimes I + H \otimes D_1 \otimes I \otimes D_1 \otimes X \quad (3.1.6)$$

## Mutation and Crossover

For *crossover*, we restrict the candidate chromosomes to have the same number of wires. That is, we wouldn't swap a 'T' gate with an 'H' gate since they operate on a different number of qubits. Although possible to implement, it simplifies the crossover search operation.

*Mutation* is done through three basic operations: 1) Adding a gate, 2) Removing a gate, and 3) Replacing a gate. In the process of adding a gate, we replace one or more 'T' gates (quantum wires) with the desired (or rather, random) gate. We need to make sure there is a sufficient number of qubits available — or — we can add another parallel logic slice in the string by using 'r' tags. In the removal of a gate, we simply replace the gate character with the appropriate number of 'T' gates. The replacement operation requires more work largely due to the case where we are replacing a gate with one of larger order (more qubits).

This causes us to check if enough qubits are available to fit the new gate. If we replace a gate with one of less qubits, we need to insert ‘I’ gates. Some repair work is needed for the replacement operation.

## Fitness Function

The fitness function is used to bias the selection operation (selecting parents) and for determining when a valid solution has been found. Although many fitness functions and their interpretations were tested, the most fundamental fitness function had the form

$$F = \frac{1}{1 + \text{Error}} \quad (3.1.7)$$

Where *Error* was entirely based on the *correctness* of the circuit, that is, how closely did it match the desired operation:

$$\text{Error} = \sum_i \sum_j |O_{ij} - S_{ij}| \quad (3.1.8)$$

Where  $O_{ij}$  is the target element of the desired operator matrix and  $S_{ij}$  is the actual element from the unitary matrix encoded by the individual.

However, this fitness function did not take into account the *cost* of the quantum circuit. The *cost* of a quantum circuit depends on many factors and is eventually heavily influenced by the physical implementation of the quantum circuit (e.g., number of NMR pulses). For a simple implementation, the cost could be the number of quantum gates excluding ‘I’ and swap gates.

Eventually, a compromise fitness equation is constructed which allows a

parameterized balance between *correctness* and *cost*, namely

$$F = \alpha \left( 1 - \frac{\text{Error}}{\text{Max\_Error}} \right) + \beta \frac{1}{\text{Cost}} \quad (3.1.9)$$

Where  $\alpha$  and  $\beta$  are the balancing parameters.

## GA Steps

The GA uses the following steps:

1. Initiates a random population of  $M$  individuals.
2. Pick  $N$  random individuals based on fitness.
3. For all  $N$  selected individuals, apply two genetic operators (below) to create  $M$  new individuals.
  - *Crossover* to recombine two individuals by exchanging respective chromosomes.
  - *Mutation* to inject noise into the chromosomes of the individual with a very small probability.
4. Update the old generation of individuals with the newly created generation.
5. Evaluate the fitness of each individual in the new population.
6. Determine if any individual(s) meet the success criteria. If so, the GA is done. Otherwise, we iterate for another generation until we've hit a solution or the maximum number of generations allowed.

## GA Results

The GA was able to synthesize well-known gates in different manners. There was a clear balance between run-time, correctness, and cost. Overall, the GA was very successful in both finding near-optimal circuits and even families of circuits, and as tool to explore alternative quantum circuits. Elegant solutions were found for gates such as the Margolus and Miller gates and the Peres solution seems to be optimal.

### 3.2 Random Unitary Matrices

The method chosen to generate random unitary matrices was developed by Życzkowski and Kuś[6] which conform to the statistical properties of circular unitary ensembles (CUE) originally investigated to describe the spectral properties of quantum objects[60].

We generate the  $N \times N$  unitary operator,  $U$ , as a composition of elementary unitary operators which perform transformations in two-dimensional subspaces. This elementary matrix is denoted by  $E^{(i,j)}(\phi, \psi, \chi)$  The non-zero elements are given by the following rule:

$$\begin{aligned} E_{kk}^{(i,j)} &= 1 & k = 1, \dots, N \quad k \neq i, j \\ E_{ii}^{(i,j)} &= \cos(\phi) e^{i\psi} \\ E_{ij}^{(i,j)} &= \sin(\phi) e^{i\chi} \\ E_{ji}^{(i,j)} &= -\sin(\phi) e^{-i\chi} \\ E_{jj}^{(i,j)} &= \cos(\phi) e^{-i\psi} \end{aligned} \tag{3.2.1}$$

The angles are taken over the intervals:

$$0 \leq \phi \leq \frac{\pi}{2} \quad 0 \leq \psi < 2\pi \quad 0 \leq \chi < 2\pi \quad (3.2.2)$$

Using the above elementary operators, we construct the  $N - 1$  composite rotations:

$$\begin{aligned} E_1 &= E^{(1,2)}(\phi_{12}, \psi_{12}, \chi_{12}) \\ E_2 &= E^{(2,3)}(\phi_{23}, \psi_{23}, 0) E^{(1,3)}(\phi_{13}, \psi_{13}, \chi_{13}) \\ &\vdots \\ E_{N-1} &= \\ &E^{(N-1,N)}(\phi_{N-1,N}, \psi_{N-1,N}, 0) \times \\ &E^{(N-2,N)}(\phi_{N-2,N}, \psi_{N-2,N}, 0) \times \\ &\dots \times E^{(1,N)}(\phi_{1N}, \psi_{1N}, \chi_{1N}) \end{aligned} \quad (3.2.3)$$

The unitary operator is finally defined as:

$$U = E_1 E_2 E_3 \dots E_{N-1} \quad (3.2.4)$$

### 3.3 Eigenanalysis Methods and Separability

A quantum computer is a register of qubits whose computation can be described by a unitary operator which utilizes quantum mechanics to evolve the initial state of the register to produce a final desired state. Given this matrix, we can analyze its properties with the goal to discover *clues* that would help us either improve an existing unitary matrix or help us design from-the-ground-up better unitary matrices, and thus, better quantum computers.

An eigenanalysis can be performed where we take an in-depth look at the eigenvalue distribution and the eigenvector placement. The eigenvalues are



found by solving the *characteristic polynomial* of the unitary matrix that satisfies the equation  $U\vec{v} = \lambda\vec{v}$ , where  $\vec{v}$  are the eigenvectors and  $\lambda$  are the eigenvalues<sup>2</sup>.

Given an  $n$  qubit system, the unitary operator of the whole system is a  $2^n \times 2^n$  matrix. Thus, we could have up to  $2^n$  eigenvalues and their corresponding eigenvectors. Obviously, for sufficiently high  $n$  (say,  $n > 5$ ), numerical methods are needed to solve for the roots of the characteristic polynomial. Software such as MATLAB and LAPACK will be used to calculate the eigenvalues and eigenvectors.

Given a  $U_f$ , we know that its eigenvalues are of the form  $\lambda_k = e^{i\theta_k}$  and thus complex. We can plot the eigenvalues on the unit circle where the horizontal axis represents the real components,  $\Re\{\lambda_k\}$ , and the vertical axis represents the imaginary components,  $\Im\{\lambda_k\}$ . It is of interest to see where the eigenvalues of a particular unitary matrix that represents an algorithm land on the unit circle, and how that placement changes for similar unitary matrices (i.e., different operators evolved to solve the same instance problem with similar fitness) or as we increase the complexity of the algorithm (e.g., increasing the number of nodes in the “independent set problem”).

The *degeneracy* of a unitary matrix is also of interest and occurs when we have two or more eigenstates  $|\xi_1\rangle$  and  $|\xi_2\rangle$  of  $\xi$  correspond to the *same* eigenvalue<sup>3</sup>. A comparison across multiple unitary matrices of different characteristics might reveal useful knowledge about the system. For example, we might find that eigenvalues tend to clump in a particular region on the unit sphere or have

---

<sup>2</sup>The set of eigenvalues is sometimes called the *spectra* or the *spectrum* of the unitary operator.

<sup>3</sup>Any linear combination of  $|\xi_1\rangle$  and  $|\xi_2\rangle$  is also a degenerate eigenstate corresponding to the same eigenvalue.

some consistent pattern. It is possible that this knowledge could be used to aid in the design of quantum unitary operators although we admit it is not clear how this can be done.

Eigenvector analysis is more complicated due to their multidimensional properties (i.e., an eigenvector of an  $n$  qubit system will have  $2^n$  components). However, the analysis can be performed and visualized as in [6] by analyzing the *localization* properties and *entropy* of the eigenvectors. The localization property is found by calculating the *inverse participation ratio*,  $\mu_l$ , by

$$\mu_l = N \sum_{k=1}^N y_{lk}^2 \quad (3.3.1)$$

where  $y_{lk} = |c_{lk}|^2$  and  $c_{lk}$  is the  $k^{th}$  component of the  $l^{th}$  eigenvector. We can then approximate the number of relevant states,  $M$ , by  $M = N/\langle\mu\rangle$ .

A second similar method is to use the Shannon entropy,  $H_l$ , of an eigenvector where

$$H_l = - \sum_{k=1}^N y_{lk} \ln(y_{lk}) \quad (3.3.2)$$

and comparing their mean entropies. Other methods will be investigated.

The *separability* is directly linked to the theory of quantum entanglement and thus a key feature of quantum computational and quantum information theory. Two systems (A and B) are separable if it meets the condition

$$\rho_{AB} = \sum_j p_j |\psi_j\rangle \langle\psi_j| \otimes |\phi_j\rangle \langle\phi_j| \quad (3.3.3)$$

Methods of identifying state separability have been investigated using eigenvalues [61] and eigenvectors [62]. How state separability relates to the function

of a quantum operator is not well understood.

### 3.4 Quantum Logic Synthesis

A rapidly increasing area of research is Quantum Logic Synthesis and numerous foundation papers [33, 63, 64, 1, 65] have been established. Similar to classic logic synthesis, a quantum logic “synthesizer” attempts to decompose a given unitary matrix into a series of fundamental quantum logic gates to form what we could consider a quantum circuit *netlist*— that is, quantum gates connected via quantum wires.

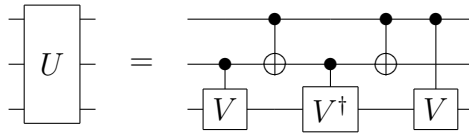


Figure 3.2: Quantum Logic Synthesis of a 3 qubit system

In Figure 3.2 above, the original unitary matrix which describes a 3 qubit system,  $U$ , is decomposed into smaller quantum gates.

#### 3.4.1 Quantum Gate Universality

A quantum computer is implemented using a series of quantum gates that are available which is dependent on its physical implementation. We refer to this array of quantum gates as the *gate library*. These gates may operate on one or many qubits. The gate library is considered a *universal set* if any  $n$ -qubit operator can be approximated within a specific degree of accuracy using only the gates from the gate library.

Deutsch et al showed in [66] that almost all 2-qubit gates are universal. However, it also continues to explain — intuitively — that the set of single

qubit gates and classical gates cannot form a universal set. The argument being that single qubit gates cannot entangle two initially un-entangled qubits into an entangled state. Also, classical gates (or quantum gates that behave like their classical counterparts) cannot produce a universal set because they map from one computational basis state to another and the superposition needed for universality is not implemented.

### 3.4.2 Computation Bounding

In 1995, Barenco *et al* showed in [33] that the set of single qubit gates and CNOT gates comprise a universal set. The method utilized  $QR$  factorization. In fact, figure 3.2 is the circuit Barenco et al proposed that any three qubit network could be broken down into given any  $2 \times 2$  matrix  $U$ , where  $V^2 = U$  and  $V^\dagger$  is the *adjoint* matrix of  $V$  such that  $V^\dagger V = I$ . This also sets the upper-bound for the number of computations at  $O(n^3 4^n)$  for  $n$  qubits. That same year Knill [67] showed an improved upper-bound of  $O(n 4^n)$  but it wasn't until 2004 that Vartanian et al made a significant leap in [65] which showed a new lowest upper-bound of  $O(4^n)$ . This was later followed by improvements by Möttönen and Vartanian in [68] to a new upper-bound of  $O(\frac{23}{48} 4^n)$  CNOT operations. This is still higher than the predicted highest lower-bound [63] of  $\lceil (4^n - 3n - 1)/4 \rceil$ .

### 3.4.3 Uniformly Controlled Gate Decomposition

A technique for decomposing uniformly controlled gates was first proposed in [65]. In short, it describes the method of decomposing a  $k$ -fold uniformly controlled  $m$ -qubit gate into a sequence of  $2^k$  gates, each with a different sequence of  $k$  controls nodes. This is illustrated in figure 3.3.

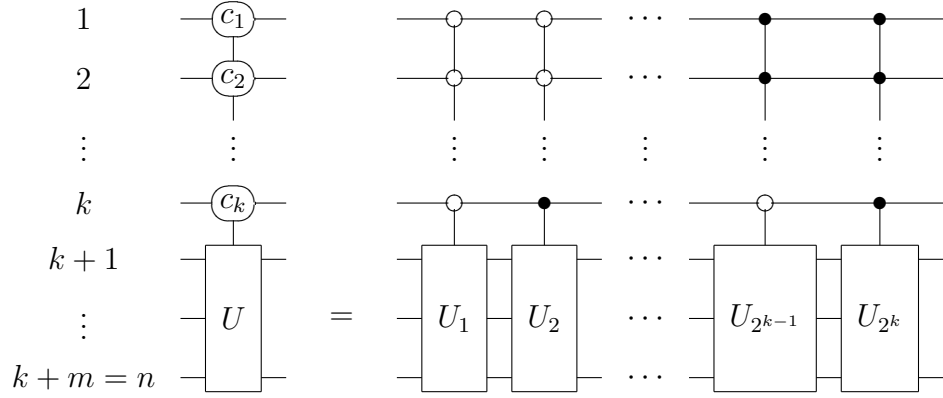


Figure 3.3: Uniformly Controlled Gate Decomposition

We refer to this type of gate by the operator  $F_T^k(U(2^m))$ . Where  $n$  is the total number of qubits,  $k$  is the number of control nodes. The target qubits are grouped into the set  $T$  of which there are  $k = n - m$  qubits. The left-side general quantum gate is controlled by an arbitrary combination of the control nodes. The right-side shows its decomposition into uniformly controlled gates, each having a distinct combination of control nodes.

#### 3.4.4 Cosine–Sine Decomposition

Cosine–Sine Decomposition (CSD) is a recursive algorithm [69] and utilizes the General Singular Value Decomposition method (GSVD). CSD operates on even-dimensional unitary matrices and thus perfectly suited for quantum operators. CSD recursively decomposes the unitary matrix into smaller unitary matrices of half dimensionality. We describe this algorithm as it applies to the decomposition of quantum operators in more detail as it is the method used by Qubiter [3] to produce the synthesis results used in chapter 7.

Given a unitary matrix,  $U$ , which is of even-dimensionality and described by a  $2^n \times 2^n$  matrix where  $n$  is an integer and represents the number of qubits,

CSD states that we can express  $U$  as:

$$U = \begin{pmatrix} L_0 & 0 \\ 0 & L_1 \end{pmatrix} D \begin{pmatrix} R_0 & 0 \\ 0 & R_1 \end{pmatrix} \quad (3.4.1)$$

As a matter of notation, we'll refer to the *left* side matrix as  $L$  and the *right* side matrix as  $R$ . We also note using our quantum logic algebra,

$$L = D_0 \otimes L_0 + D_1 \otimes L_1 \quad (3.4.2)$$

$$R = D_0 \otimes R_0 + D_1 \otimes R_1 \quad (3.4.3)$$

(Note:  $D_0$  and  $D_1$  are the projection matrices and not to be confused with the CSD central matrix,  $D$ ).

In this case, sub-matrices  $R_0$ ,  $R_1$ ,  $L_0$ , and  $L_1$  are of the same dimensionality which is half that of  $U$ . That is, they are  $\frac{2^n}{2} \times \frac{2^n}{2}$  unitary matrices. From now on,  $N$  is the dimension of the original matrix,  $U$ , where  $N = 2^n$ .

The center  $D$  matrix is described as

$$D = \begin{pmatrix} C & S \\ -S & C \end{pmatrix} \quad (3.4.4)$$

Where  $C$  and  $S$  are diagonalized matrices of half-dimensionality ( $N/2$ ) compared to  $D$  and are formed using *cosine* and *sine* functions, respectively. That is,

$$C = \text{diag}(\cos \theta_1, \cos \theta_2, \dots, \cos \theta_{N/2}) \quad (3.4.5)$$

and,

$$S = \text{diag}(\sin \theta_1, \sin \theta_2, \dots, \sin \theta_{N/2}) \quad (3.4.6)$$

We note the unitary relationship of the angles,

$$\sin^2 \theta_i + \cos^2 \theta_i = 1 \quad (3.4.7)$$

The *diag* function used above has an  $N$ -length vector argument and produces an  $N \times N$  matrix with the  $N$ -length main diagonal set to values specified in the argument vector. That is,

$$diag(v_N) = M_{ij} = \begin{cases} 0 & \text{if } i \neq j \\ v_i & \text{if } i = j \end{cases} \quad (3.4.8)$$

It can be shown that given any CSD of  $U$ , another CSD of  $U$  can be found such that the angles  $\{\theta_1, \dots, \theta_{N/2}\}$  are in non-decreasing order and are bounded to the range  $[0, 90]$  degrees.

As it turns out, matrix  $D$  can be implemented as uniformly controlled  $y$  rotation operation using  $F_1^{n-1}(R_y)$  with associated rotation angles  $\{\theta_1, \dots, \theta_{N/2}\}$ . The  $L$  and  $R$  correspond to the uniformly controlled  $(n-1)$  qubit gates which are of the type  $F_T^1(SU(2^{n-1}))$ .

In the CSD algorithm,  $U$  is partitioned into four blocks:  $U_{ij}$  where  $i, j \in \{0, 1\}$ . In which case,

$$U_{ij} = L_i D_{ij} R_j \quad (3.4.9)$$

and  $D_{ij}$  is the singular value of  $U_{ij}$ .

The recursive property is such that if we start with a unitary  $U$  of size  $2^n \times 2^n$  then after  $n$ -levels of recursion, our decomposed matrices ( $L$ ,  $D$ , and  $R$ ) are  $2 \times 2$  meaning our sub-matrices ( $L_0, L_1, R_0, R_1, C$ , and  $S$ ) are single complex elements with modulo unit value. The iteration stops when the blocks can

no longer be broken down which are typically 2-input ( $4 \times 4$ ) or single-input ( $2 \times 2$ ) matrices. These are then mapped to their closest quantum gates which are typically CNOT, rotation, or phase gate operations.

If we had a gate library which did not include variable gates (e.g., rotation gates or arbitrary phase gates), then our synthesis mapping is significantly more complicated. We would instead need to find a combination of gates whose resultant operator is similar to the block matrix derived from CSD. In these cases, there is an error on the angle parameters. To keep this error level low, there is a direct impact to gate count as more gates from the library are needed.

There is a mapping that can be done of result of CSD to quantum multiplexors. In fact, the  $D$  matrix implementation of controlled  $R_y$  rotations is a variation of the quantum multiplexor. The block diagonal matrices,  $L$  and  $R$ , are in fact quantum multiplexors, selecting between either  $L_0$  and  $L_1$  (or between  $R_0$  and  $R_1$  for the  $R$  block matrix). Figure 3.4 illustrates such a quantum circuit in two different forms. The figure on the left shows the quantum multiplexor drawn in a similar fashion as a classic mux. The  $qc$  qubit controls which unitary operators output will be chosen. The right-hand figure is the equivalent circuit but using quantum inverters to activate the desired unitary block and deactivate the other, turning it into a quantum wire, logically behaving as an identity gate.

### 3.4.5 Quantum Compilers

There is a handful of known quantum logic synthesizers<sup>4</sup>. A very well known example is Qubiter [3]. Qubiter uses the very successful approach of using *Cosine*-

---

<sup>4</sup>Quantum Logic Synthesizers can also be called Quantum Logic *Compilers*



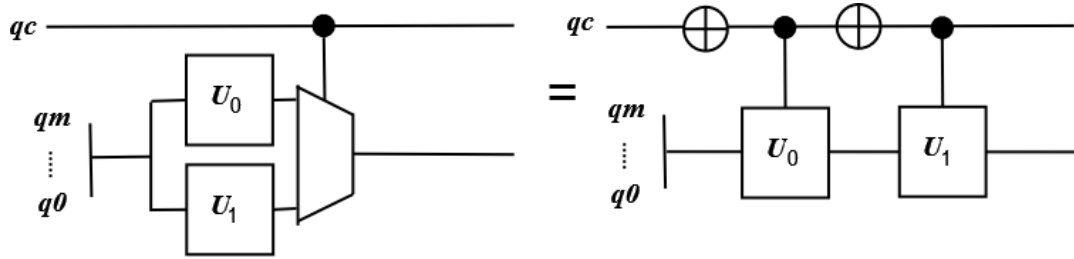


Figure 3.4: Quantum Multiplexor selecting between block operators from CSD

*Sine Decomposition* which we just described in section 3.4.4 to recursively decompose the initial unitary matrix to simpler matrices which can then be mapped to a quantum gate library. This method was also used in [2]. The source code for Qubiter is freely available at <http://www.ar-tiste.com/qubiter.html>. As Qubiter was freely available, it was used for the synthesis results of this research and the results are summarized in chapter 7.

Although other synthesizers exist [70], their code is not readily available—or—is too severely limited (e.g., S. Bullock’s *3CNOT* synthesizer can only handle two qubit systems).

Lukac et al published a quantum logic synthesis approach [40] that used a *Genetic Algorithm* to synthesis an initial given unitary matrix into a sequence of elementary gates. An overview of their method was covered in section 3.1.

## Chapter 4

### Quantum Logic Algebra

#### 4.1 Introduction

Quantum Circuits (also called *Quantum Arrays* or *Quantum Netlists*) were originally developed by Deutsch [22] and are a convenient and widely accepted method of visualizing how quantum gates operate on qubits over time to ultimately describe the unitary matrix of the quantum computer,  $U$ . This is analogous to a classical boolean gate netlist implementing the logic described by a truth table. However, unlike classical logic, deriving the algebraic expression for a quantum circuit has been—until now—difficult.

There are traditional methods for constructing the unitary matrix[9] and we'll briefly cover these so that we can later contrast them with our method.

A method known as *diagonal representation* or sometimes *orthonormal decomposition* allows representing unitary operators under certain criteria using the equation  $U = \sum_i \lambda_i |i\rangle \langle i|$ . The  $|i\rangle$  vectors form an orthonormal set of eigenvectors with  $\lambda_i$  being the corresponding set of eigenvalues. Simple examples of this method include the Controlled-NOT with  $CNOT = |00\rangle \langle 00| + |01\rangle \langle 01| + |10\rangle \langle 11| + |11\rangle \langle 10|$  and the Z-gate with  $Z = |0\rangle \langle 0| - |1\rangle \langle 1|$ . The limitation of the diagonal representation is that: 1) The operator must be diagonalizable, 2)

it requires knowing all eigenvectors which in general requires  $2^n$  vectors for  $n$  qubits meaning the expression gets very large for moderate number of qubits, and lastly, 3) it also requires deriving the eigenvalues. For permutation operators — such as the NOT and Toffoli gates — the problem is simplified since the eigenvalues are unity.

Perhaps the easiest method is to divide the quantum circuit into vertical slices and identify *parallel* quantum gates (and quantum wires, denoted by  $I$ ) in each slice and use the Kronecker product as depicted in figure 4.1 to form the equation for each slice. The slices can then be matrix multiplied together to form the final  $U$  of the quantum computer using the property that  $U = S_N \times S_{N-1} \times \cdots \times S_1$ .

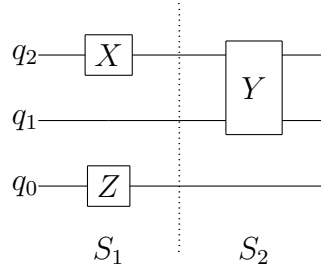


Figure 4.1: Parallel Logic Example:  $S_1 = X \otimes I \otimes Z$  and  $S_2 = Y \otimes I$ , thus  $U = S_2 \times S_1$

The limitation with this method is that it can only handle parallel operations and although it is possible to identify known circuit transforms or reductions[33, 68], the difficulty increases quickly when we add multiple controls (including negative control nodes), control nodes that span over many qubits, complex gates, etc. Some of these difficult examples are shown in figure 4.2 and rely on special math routines or error-prone hand calculations to solve.

We propose a new method that allows representing a slice of logic as a *sum of*

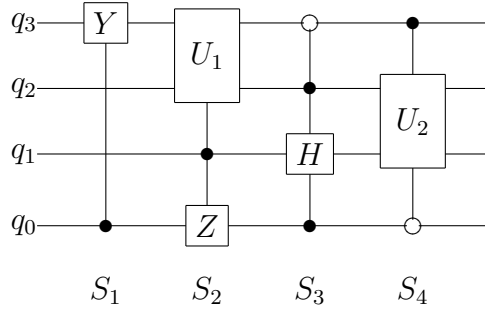


Figure 4.2: Example of Difficult Logic Slices:  $U = S_4 \times S_3 \times S_2 \times S_1$

*Kronecker products*(SOK). The slice equations can then be multiplied together to form the final algebraic expression of the quantum computer, that is,  $U$ . In essence, the quantum computer is described by a *product of sum of Kronecker products* (POSOK). The method handles complex, multi-input quantum gates and multiple positive and negative control nodes. Essentially, constructing the equation for  $U$  based on the quantum circuit becomes trivial!

At the heart of our method is how the unitary matrix is constructed in the presence of control nodes. Note the unitary matrix for a generalized controlled gate,  $G$ , shown in figure 4.3. For the left-hand figure, we can construct the

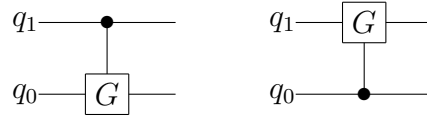


Figure 4.3: Generalized control gate

unitary matrix and note its algebraic expression:

$$U = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & g_1 & g_2 \\ 0 & 0 & g_3 & g_4 \end{pmatrix} = \begin{pmatrix} I & 0 \\ 0 & G \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \otimes I + \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes G$$

However, things become interesting if we flip the circuit as in the right-hand figure where  $G$  is operating on qubit 1 and being controlled by qubit 0. Then we have

$$U = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & g_1 & 0 & g_2 \\ 0 & 0 & 1 & 0 \\ 0 & g_3 & 0 & g_4 \end{pmatrix} = I \otimes \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + G \otimes \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

In the above examples, we derive the general equation for a two qubit controlled gate circuit in both orientations and we see common matrix terms which we later call  $D_0$  and  $D_1$ . Through multiple examples and analysis, we find a pattern and eventually arrive at a general method. Section 2 details this method.

We now go on to demonstrate this method after listing some key definitions. We follow this with a section on the canonical representation of a quantum logic slice and how it is equivalent to our method. We also show how using certain properties, we can manipulate our equations to show known circuit equivalences.

## 4.2 Key Definitions

**Definition 10** *Two matrices  $A$  and  $B$  are independent if  $A \times B = B \times A$ , that is, they are commutative.*

Note that any matrix,  $M$ , and the identity matrix,  $I$ , are naturally independent since  $M \times I = M = I \times M$ .

Let  $Q$  be a quantum circuit and let  $\hat{Q}$  be a sub-circuit of  $Q$ .

**Definition 11**  *$\hat{Q}$  is a quantum logic slice if all of the following properties hold*

1. All qubits used in  $Q$  are also used in  $\hat{Q}$ .
2. No qubit is operated on by more than one quantum gate in  $\hat{Q}$ .
3. Any qubit that is used as a control in  $\hat{Q}$  cannot be operated on by any gate in  $\hat{Q}$ .

Without loss of generality, we assume quantum gates operate on consecutive qubits. For example, a two-input quantum gate could operate on, say, qubits  $q_0$  and  $q_1$ , but not on qubits  $q_0$  and  $q_3$ .

A fundamental property of all quantum logic slices—which we exploited to develop our method—is they can be recursively decomposed into smaller independent slices.

**Theorem 4.2.1** *Any quantum logic slice,  $S$ , consisting of  $n$  gates (whether controlled or not) can be decomposed into two independent logic slices,  $S_1$  and  $S_2$  such that one contains  $n - 1$  gates and the other contains 1 gate.*

*Proof:* Pick any arbitrary quantum gate from  $S$  (say the  $k$ -th gate  $G_k$ ). Let  $A_k$  be the set of qubits operated on by  $G_k$  and, if  $G_k$  is a controlled gate, all bits used for its control as well. Move  $G_k$  to  $S_1$  and leave the  $n - 1$  remaining gates in  $S_2$ . Any qubit  $j \in A_k$  in  $S_1$  behaves like an identity  $I$ . Similarly, any qubit  $j \in A_k$  in  $S_2$  behaves like an identity  $I$ . More specifically, any qubit that is used for control or is operated on in  $S_1$  behaves like an identity in  $S_2$  and any qubit that is an identity in  $S_1$  is either operated on or is used for control in  $S_2$ . This means  $S_1$  is a block-diagonal matrix with all but the  $k$ -th block an identity matrix.  $S_2$  is a block-diagonal matrix of the same overall dimensions (including the blocks) but with only the  $k$ -th block an identity matrix. Under these circumstances  $S = S_1 \times S_2 = S_2 \times S_1$ . Hence,  $S_1$  and  $S_2$  are independent.

### 4.3 The Hutsell Method

Unless otherwise stated, Kronecker products are depicted as matrix multiplications to simplify the notation. Matrix multiplies will be explicitly written (i.e.,  $A \times B$ ).

$$A \otimes B = AB$$

We introduce two matrices,  $D_0 = |0\rangle\langle 0|$  and  $D_1 = |1\rangle\langle 1|$ ,

$$D_0 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \quad D_1 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Notice  $D_0 + D_1 = I$

The quantum circuit is first separated into quantum logic slices. A slice can contain both non-controlled and controlled gates, however, we introduce the following definition of what constitutes a legal slice for the method:

**Definition 12** *A method legal slice is a quantum logic slice where all the controlled gates share the same control nodes.*

If this is not the case, then we can split the slice using theorem 4.2.1 into multiple slices until this condition is met. Figure 4.4 shows an example of creating legal slices for our method.

After identifying our legal slices, the method is broken down to three key steps that are applied to each slice. We'll apply this method to a number of key examples.

**STEP 1. Identify Number of Terms:** Given a slice with  $N$  control nodes, the number of terms in the algebraic expression of the slice will be  $N + 1$ .

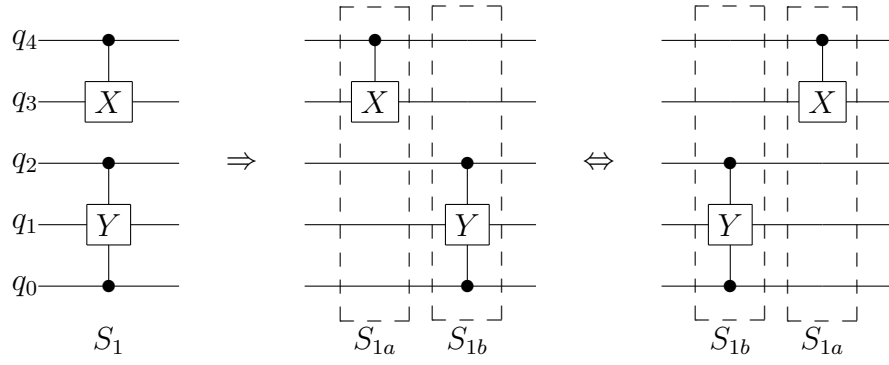


Figure 4.4: Splitting Slices:  $S_1 = S_{1b} \times S_{1a}$ . Alternatively, we could have done  $S_1 = S_{1a} \times S_{1b}$  since  $S_{1a}$  and  $S_{1b}$  are independent

If  $T_i$  represents term  $i$ , then the equation for the slice will have the form

$$S = \sum_{i=1}^{N+1} T_i$$

**STEP 2. Control and Parallel Logic Terms:** The first  $N$  terms represent the necessary manipulation to the unitary matrix of the slice to appropriately handle the controlled gates and non-controlled gates. The controlled gates will not be present in the first  $N$  terms. We leave that to step 3.

For an  $m$  qubit circuit  $q_{m-1}, q_{m-2}, \dots, q_0$ , each term,  $T_i$ , is represented as a Kronecker product of matrices ordered from left to right. The left-most position representing the most-significant qubit,  $q_{m-1}$ , and the right-most position representing the least significant qubit,  $q_0$ . Each term is constructed using the follow template,

$$T = \underbrace{P_{m-1} \otimes P_{m-2} \otimes \dots \otimes P_0}_{q_m, q_{m-1}, \dots, q_1} = P_{m-1} P_{m-2} \dots P_0$$

- **RULE 1:** If there is a  $k$ -qubit *non-controlled* gate,  $G_u$ , that operates on qubits  $l$  to  $l + k - 1$ , then replace  $P_{l+k-1} \dots P_l$  with  $G_u$ .



- **RULE 2:** If there is a  $k$ -qubit *controlled* gate,  $G_c$ , that operates on qubits  $l$  to  $l + k - 1$  then replace  $P_{l+k-1} \dots P_l$  with  $I^{\otimes k}$  (that is,  $k$   $I$  matrices tensored together).
- **RULE 3:** Assume  $\mathbf{C} = \{a, b, c, \dots\}$  is the set of  $N$  control nodes ordered from highest to lowest qubit order (that is  $a > b > c$ , etc). We then use table 4.1 to replace  $\{P_a, P_b, P_c, \dots\}$  with the specified operators. For example, in  $T_1$ , we replace  $P_a$  with  $D_0$  and the rest with  $I$ . In  $T_2$ , we promote  $P_a$  from  $D_0$  to  $D_1$ ,  $P_b$  becomes  $D_0$ , and the rest stay  $I$ , and so on.

**Exception:** If the control node is a negative control, then we do the opposite and set to  $D_1$  then promote to  $D_0$  in the next term.

**STEP 3: Define your Controlled Logic Term:** The last term is really a logical continuation of STEP 2, except the controlled gates are now represented by themselves and all control nodes are represented by  $D_1$  (or  $D_0$  for the negative control nodes). RULE 1 still applies.

Term	Substitution Order
1	$D_0, I, I, \dots, I$
2	$D_1, D_0, I, \dots, I$
3	$D_1, D_1, D_0, I, \dots, I$
4	$D_1, D_1, D_1, D_0, I, \dots, I$
$N - 1$	$D_1, D_1, \dots, D_1, I$
$N$	$D_1, D_1, \dots, D_1$

Table 4.1: Substitution table for  $\#$  symbols. Substitutions are made from left to right in each term.

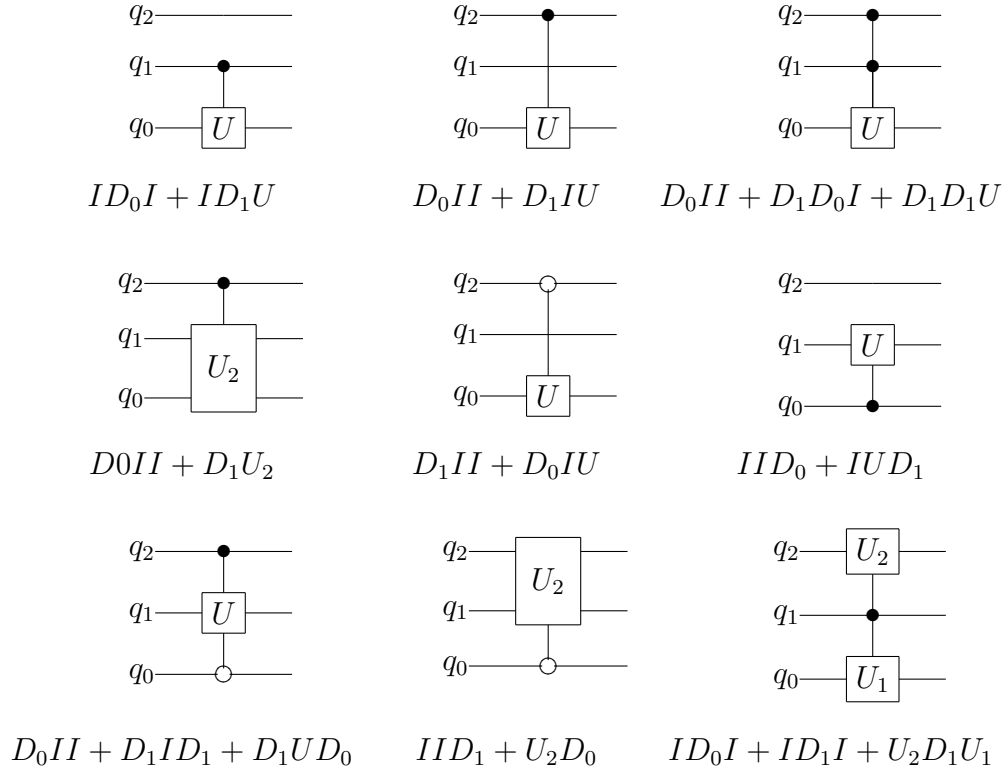


Figure 4.5: Method Examples

#### 4.4 Slice Examples

We leave it as an exercise to the reader to use the method described to derive the algebraic equations shown in figure 4.5

#### 4.5 Canonical Construction

The method described above is really a simplification derived from the more formal canonical description which we attempt to do here. We start by defining some mathematical tools, the first being the definition of a *diagonal function*

**Definition 13** The diagonal function,  $\mathbf{D}_b$  is defined as

$$\mathbf{D}_b = \mathbf{D}_{b_{N-1} \dots b_0} = \bigotimes_{i=N-1}^0 D_{b_i} \quad (4.5.1)$$

Where  $b$  is an  $N$ -wide binary number ranging from 0 to  $2^N - 1$  and  $b_i \in \{0, 1\}$ ,

$$\mathbf{D}_b = D_{b_{N-1}} \otimes D_{b_{N-2}} \otimes \dots \otimes D_{b_0} \quad (4.5.2)$$

It can be shown that  $N$  arbitrary Kronecker products of  $D_0$ 's and  $D_1$ 's result in setting a particular diagonal element of a  $2^N \times 2^N$  zero matrix to unity. In general,  $\mathbf{D}_b$  will set the  $b$ 'th diagonal element to 1. As an example, consider  $b = 101$ , then  $\mathbf{D}_{101}$  creates a  $2^3 \times 2^3$  matrix with the 6th (decimal value of  $b$  is 5) diagonal element to 1. This is done by the expression

$$\mathbf{D}_{101} = D_1 \otimes D_0 \otimes D_1$$

Each binary address corresponds to one of the  $2^N - 1$  diagonal elements. For example,  $\mathbf{D}_{00..0}$  sets the first top diagonal element,  $M_{0,0}$ , to 1 and  $\mathbf{D}_{11..1}$  sets the last bottom diagonal element,  $M_{N-1,N-1}$ , to 1. It follows that the sum of the diagonal function over all possible addresses will set all diagonal elements to unity thus creating the  $2^N$ -order identity matrix (i.e.,  $I^{\otimes N}$ ) which we'll simply write as  $I$ :

$$\sum_{b=00..0}^{11..1} \mathbf{D}_b = \sum_{b=00..0}^{11..1} D_{b_0} \otimes D_{b_1} \otimes \dots \otimes D_{b_{N-1}} = I \quad (4.5.3)$$

To simplify expressions where a particular qubit's state can be either  $|0\rangle$  or  $|1\rangle$ , we can introduce "don't cares" (denoted by  $X$ ). This allows us to write  $\mathbf{D}_x =$

$\mathbf{D}_0 + \mathbf{D}_1 = D_0 + D_1 = I$ . Therefore, whenever we see  $X$  in the diagonal function, we can substitute that particular qubit with the identity matrix. Therefore,

$$\mathbf{D}_{xx\dots x} = I \otimes I \otimes \dots \otimes I$$

A useful property of the diagonal function is that, given an arbitrary state vector,  $|\psi\rangle = [x_0, x_1, \dots, x_{2^N-1}]^T$  which can be re-written using binary subscripts as  $[x_{0\dots 00}, x_{0\dots 01}, \dots, x_{1\dots 11}]^T$ . We get

$$\mathbf{D}_b |\psi\rangle = |\psi_b\rangle = [0 \dots x_b \dots 0]^T \quad (4.5.4)$$

Also,  $\mathbf{D}_b$  is linear and obeys superposition

$$(\mathbf{D}_{b_1} + \mathbf{D}_{b_2}) |\psi\rangle = \mathbf{D}_{b_1} |\psi\rangle + \mathbf{D}_{b_2} |\psi\rangle = |\psi_{b_1}\rangle + |\psi_{b_2}\rangle \quad (4.5.5)$$

It also follows that

$$\mathbf{D}_{xx\dots x} |\psi\rangle = I |\psi\rangle = |\psi\rangle \quad (4.5.6)$$

We next look at the properties of *partial identity matrices*.

**Definition 14** A partial identity matrix is any matrix,  $M$ , that has any of its diagonal elements set to unity and all other elements are zero. That is,  $M$  is of the form:

$$M = \sum_{b=00\dots 0}^{11\dots 1} (\mathbf{D}_b, \text{ if } b \in \mathbf{B}) \quad (4.5.7)$$

Where  $\mathbf{B}$  is an arbitrary set of binary address  $\{b_1, b_2, \dots\}$  each addresses a particular diagonal element in  $M$ .

**Definition 15** The Rules of Partial Matrices are as follows; Given a partial

identity matrix,  $M$ :

1. Basic Identity:  $M \times M = M$
2. Conjugate Rule:  $M^* = M$
3. Transpose Rule:  $M^T = M$
4. Adjoint Rule:  $M^\dagger = M$
5. Preservation Rule: *Given an arbitrary matrix,  $A$ ,  $(M \times A)^\dagger = M \times A^\dagger$  (same for conjugate and transpose)*
6. Non-Unitary Clause: *Because of the Adjoint Rule,  $M$  is not unitary unless  $M = I$ .*
7. Compliment: *The complement of  $M$ , is  $K = I - M$ .*
8. Compliment Rules: *Given  $M$  and its complement,  $K$ ,  $M \times K = K \times M = 0$ , and  $M + K = I$ .  $M$  and  $K$  are independent.*

Previously we had defined the concept of a *legal* logic slice for our method and now we refine the definition for our canonical expression. Now we only consider a quantum circuit slice composed of a single gate (controlled or not controlled). Theorem 4.2.1 can be iteratively applied to decompose a matrix into refined quantum logic slices. Figure 4.6 below demonstrates this further refinement.

For an  $N$  qubit logic slice composed of a set of qubits,  $\mathbf{Q} = \{q_0, q_1, \dots, q_{2^N-1}\}$ , which contains an  $n$ -input controlled gate,  $G$ , operating on the successive set of qubits,  $\mathbf{Q}_g = \{q_i, q_{i+1}, \dots, q_{i+n-1}\}$ , *positively* controlled by a set of qubits,  $\mathbf{Q}_p$ , and *negatively* controlled by a set of qubits,  $\mathbf{Q}_n$ , such that  $\mathbf{Q}_p, \mathbf{Q}_n \subseteq \mathbf{Q}$  and

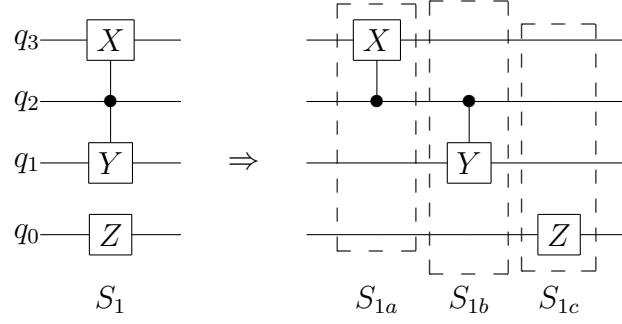


Figure 4.6: Refined Slices:  $S_1 = S_{1c} \times S_{1b} \times S_{1a}$ .  $S_{1a}$ ,  $S_{1b}$ , and  $S_{1c}$  are independent.  $S_{1a} = ID_0II + XD_1II$ ,  $S_{1b} = ID_0II + ID_1YI$ , and  $S_{1c} = IIIZ$ .  $S_1 = ID_0IZ + XD_1YZ$ .

$\mathbf{Q}_n \cap \mathbf{Q}_p = \emptyset$ , we can describe the general form of the logic slice,  $S$ , using the canonical form:

$$S = P + (P_c \times C) \quad (4.5.8)$$

$S$  becomes a  $2^N \times 2^N$  unitary matrix.  $P$  is the  $2^N \times 2^N$  matrix which maintains the state of the pass-through and control qubits when our control conditions are not all active. The  $P_c$  term is a  $2^N \times 2^N$  matrix which maintains the state of qubits being used as control nodes when they are active. This is multiplied by  $C$  which is the  $2^N \times 2^N$  matrix which performs the gate operation on the appropriate target qubits to change their state.

**Definition 16** *The control condition is the set,  $\mathbf{B}_c$ , of basis states,  $|b\rangle$ , that assert all the qubits required to activate the controlled gate,  $G$ , of a logic slice.*

For example, in a 3 qubit slice, if controlled gate,  $G$ , is controlled positively by qubit  $q_0$  and negatively by qubit  $q_2$ , then its control condition will be the set,  $\mathbf{B}_c = \{|100\rangle, |110\rangle\}$ .

Conceptually, when we're not in a control condition, the equation for  $S$  becomes  $S = P$  where  $P$  is a partial identity matrix, and for the state of the qubits,  $S$  behaves like an identity matrix allowing the state to stay the same. However, when the control conditions are all active, our  $P_c \times C$  term will come into play and operate on the qubits appropriately while still maintaining state of the pass-through and control qubits.

More formally, we can define our  $P$ ,  $P_c$ , and  $C$  terms as:

$$P = \sum_{b=00..0}^{11..1} \begin{cases} \bar{0}, & \text{if } |b\rangle \in \mathbf{B}_c \\ \mathbf{D}_b = D_{b_0} \otimes D_{b_1} \otimes \cdots \otimes D_{b_{N-1}}, & \text{otherwise} \end{cases} \quad (4.5.9)$$

*Caveat:*  $P = \bar{0}$  if  $S$  contains no control nodes.

$$P_c = \sum_{b=00..0}^{11..1} \begin{cases} \mathbf{D}_b, & \text{if } |b\rangle \in \mathbf{B}_c \\ \bar{0}, & \text{otherwise} \end{cases} \quad (4.5.10)$$

$$\begin{aligned}
C &= \bigotimes_{i=N-1}^0 \left\{ \begin{array}{l} I, \quad \text{if } q_i \text{ is not operated on by } G \\ G, \quad \text{if } q_i \text{ is operated on by } G \end{array} \right. \\
(4.5.11) \quad & \quad \quad \text{NOTE: If } G \text{ is an } n \text{ multi-qubit gate, then a single instance} \\
& \quad \quad \text{of } G \text{ is needed for qubits } i - n + 1 \text{ to } i
\end{aligned}$$

**Definition 17**  $P$  and  $P_c$  are partial identity matrices and also complementary such that  $P + P_c = I$ . Also,  $P \times P_c = P_c \times P = 0$ .

This is understood from the equation definitions for  $P$  and  $P_c$  and the rules of partial identities. In equation 4.5.9,  $P$  is defined by as the sum of the diagonal function for all qubit combinations *except* the control conditions. In contrast, equation 4.5.10 shows  $P_c$  is defined by the sum of the diagonal function for *only* the control condition. Since  $P$  and  $P_c$  use the exhaustive set of qubit combinations,  $P + P_c$  becomes the sum of diagonal functions on all qubit combinations and thus becomes like equation 4.5.3 and is equal to  $I$ .

**Lemma 1**  $P_c$  and  $C$  are independent and therefore can be written as  $P_c \times C = C \times P_c$ .

We'll omit the proof but it can be easily shown through their construction that they operate on different qubits and are thus independent.

**Lemma 2**  $P_c$  can always be reduced to a single term of Kronecker products.

From the definition of  $P_c$ , we can simplify the controlled condition states into a single expression using the “don't care” property of the diagonal function since



the same control qubits have the same state in all states in  $\mathbf{B}_c$ .  $P_c$  can then be expressed as:

$$P_c = \bigotimes_{i=N-1}^0 \begin{cases} D_1, & \text{if } q_i \text{ is a positive control node of } G \\ D_0, & \text{if } q_i \text{ is a negative control node of } G \\ I, & \text{otherwise.} \end{cases} \quad (4.5.12)$$

**Lemma 3**  *$P$  can always be reduced to  $m$  terms where  $m$  is the number of control nodes in the slice.*

According to the definition of  $P$ , if we have  $m$  control qubits, then  $P$  will be the sum of all diagonal functions except when all  $m$  qubits are in their control configuration. That is, we can assume all other qubits are “don’t cares”. However, for the  $m$  qubits, one of them must be in the non-active state, while the others are in the active state. That is, if we label the  $m$  control qubits as  $q_{c1}, q_{c2}, \dots, q_{cm}$ , then we encode their state (assuming positive controls) as shown in table 4.2. From the table we see that we’ll have  $m$  entries and therefore,  $P$  can be simplified to  $m$  terms.

$q_{c1}$	$q_{c2}$	$q_{c3}$	$\dots$	$q_{cm}$
0	$X$	$X$	$\dots$	$X$
1	0	$X$	$\dots$	$X$
1	1	0	$\dots$	$X$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
1	1	1	$\dots$	0

Table 4.2: Encoding table to simplify  $P$

**Definition 18** *Since  $P + P_c = I$ , we can replace  $P$  in equation 4.5.8 with  $I - P_c$ . This gives us*

$$S = (I - P_c) + P_c \times C \quad (4.5.13)$$

This is generally a better representation of the canonical form since  $P_c$  always simplifies to one term and  $P$  always simplifies to  $m$  terms where  $m$  is the number of control nodes in the slice. This comes from the definition of  $P$  where at two control nodes,  $P$  will have 2 terms and  $I - P_c$  is two terms. After—which, as we increase the number of control nodes, the number of terms in  $P$  grows linearly.

**Theorem 4.5.1**  $S = P + P_c \times C = (I - P_c) + P_c \times C$  can also represent a logic slice with a non-controlled gate,  $G$ .

*Proof:* The equation for such a slice is the simple Kronecker product of identity matrices and the gate,  $G$ , and is clearly equation 4.5.11 for  $C$ . Under these circumstances,  $P = 0$  (by our caveat and also by  $P = I - P_c$ ),  $P_c = I$ . Therefore,  $S = C$ .

**Theorem 4.5.2**  $S = P + P_c \times C = (I - P_c) + P_c \times C$  can generalize a single gate,  $U$ , operating on all qubits.

*Proof:* That is,  $S = U$ . Under this condition, there are no controls which makes this a trivial problem. According to our definitions of  $P$ ,  $P_c$ , and  $C$  we find that  $P = 0$  (from our caveat and also from  $P = I - P_c$ ),  $P_c = I$ , and  $C = U$ . This yields,  $S = 0 + I \times U = U$ .

**Theorem 4.5.3**  $S$  is unitary.

*Proof:* That is,  $S^\dagger \times S = I$  where  $S^\dagger$  denotes the hermitian adjoint of  $S$ . Since  $P$  and  $P_c$  are — by definition — partial identity matrices,  $P^\dagger = P$  and  $P_c^\dagger = P_c$ .  $C$  must be unitary since it is the result of Kronecker products of identities and the unitary gate,  $G$ , and therefore,  $C^\dagger \times C = I$ . We now do the math

$$S^\dagger \times S = (P + P_c \times C)^\dagger \times (P + P_c \times C).$$

It can be shown that the adjoint is preserved over addition, that is,  $(A + B)^\dagger = A^\dagger + B^\dagger$ . Therefore,

$$S^\dagger = P^\dagger + (P_c \times C)^\dagger$$

Which leads us to the expansion

$$\begin{aligned} S^\dagger \times S &= (P^\dagger + (P_c \times C)^\dagger) \times (P + P_c \times C) \\ &= P^\dagger \times P + P^\dagger \times P_c \times C + (P_c \times C)^\dagger \times P + (P_c \times C)^\dagger \times (P_c \times C) \end{aligned}$$

We use the rules of partial identities to simplify the above equation. For example,  $P^\dagger = P$ , and  $P_c^\dagger = P_c$ , therefore,  $P^\dagger \times P = P \times P = P$  and the same for  $P_c$ . Also, since  $P$  is the complement of  $P_c$ ,  $P \times P_c = P_c \times P = 0$  which directly cancels our second term. The third term must be manipulated by the preservation rule to convert  $(P_c \times C)^\dagger \times P = P_c \times C^\dagger \times P$ . Since  $P_c$  and  $C$  are independent, we can write it as  $C \times P_c \times P$  and we can clearly cancel the term. The last term requires similar techniques. We use the preservation rule to re-write it as  $P_c \times C^\dagger \times P_c \times C$  and because of the independence of  $P_c$  and  $C$  we can re-arrange it as  $P_c \times P_c \times C^\dagger \times C$  which is simply  $P_c \times I = P_c$  since  $C$  is unitary. Finally, we are left with

$$\begin{aligned} S^\dagger \times S &= P + P_c \\ &= I \end{aligned}$$

A similar method will also show that the form:  $S = I - P_c + P_c \times C$  is also unitary.

As an illustrative example, consider figure 4.7 where we start with a refined legal slice and show how we break it into the  $P$ ,  $P_c$ , and  $C$  components.

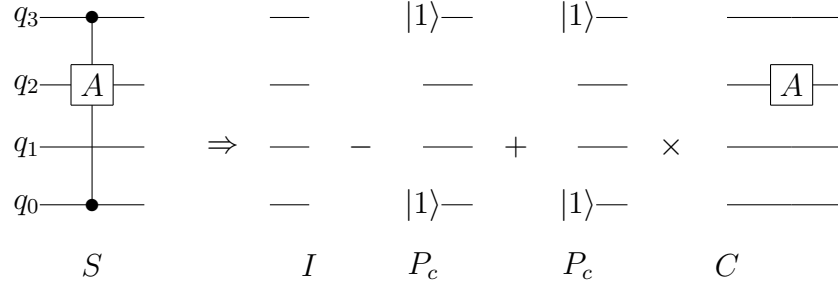


Figure 4.7: Canonical Representation Example:  $S = P + P_c \times C = (I - P_c) + P_c \times C$  (shown)

The control qubits are  $q_3$  and  $q_0$ . Both are positive controls. The pass-through term,  $P$ , according to equation 4.5.9 is the summation of all diagonal functions with  $q_3$  and  $q_0$  set to non-active state combinations (i.e.,  $q_3$  and  $q_0$  can't both be  $|1\rangle$ ):

$$P = \mathbf{D}_{0000} + \mathbf{D}_{0001} + \mathbf{D}_{0010} + \cdots + \mathbf{D}_{1110} \quad (4.5.14)$$

Using equation 4.5.10,  $P_c$  becomes the sum of diagonal functions when both  $q_0$  and  $q_2$  are  $|1\rangle$ . That is,

$$P_c = \mathbf{D}_{1001} + \mathbf{D}_{1011} + \cdots + \mathbf{D}_{1111} \quad (4.5.15)$$

Continuing with our example, we can greatly simplify the equation for  $P$  and  $P_c$  to

$$P = \mathbf{D}_{0xx0} + \mathbf{D}_{0xx1} + \mathbf{D}_{1xx0} \quad (4.5.16)$$

$$\begin{aligned} &= D_0 \otimes I \otimes I \otimes D_0 + D_0 \otimes I \otimes I \otimes D_1 + D_1 \otimes I \otimes I \otimes D_0 \\ &= D_0 \otimes I \otimes I \otimes (D_0 + D_1) + D_1 \otimes I \otimes I \otimes D_0 \\ &= D_0 \otimes I \otimes I \otimes I + D_1 \otimes I \otimes I \otimes D_0 \end{aligned} \quad (4.5.17)$$

$$P_c = \mathbf{D}_{1xx1} \quad (4.5.18)$$

$$= D_1 \otimes I \otimes I \otimes D_1 \quad (4.5.19)$$

Lastly, the  $C$  term is generated directly from equation 4.5.11 or quite easily by hand as the Kronecker product

$$C = I \otimes A \otimes I \otimes I \quad (4.5.20)$$

Rolling it all up using equation 4.5.8 and using our simplified Kronecker product notation (namely,  $A \otimes B = AB$ ), we get:

$$S = D_0 III + D_1 IID_0 + (D_1 IID_1) \times (IAII) \quad (4.5.21)$$

$$= D_0 III + D_1 IID_0 + (D_1 AID_1) \quad (4.5.22)$$

As a side-note, if we wanted to use equation 4.5.13 and went back and calculated  $P$  as  $I - P_c$ , we would have had  $P = IIII - D_1 IID_1$ . It's not directly obvious that this is equivalent to equation 4.5.14. However, because

$I = D_0 + D_1$ , we can expand  $IIII$  into  $D_0III + D_1III$  and even further to create  $D_0III + D_1IID_0 + D_1IID_1$ . Then, we see the  $D_1IID_1$  term is canceled and we're left with equation 4.5.14.

Using our method, we would arrive at the same solution above.

**Theorem 4.5.4** *The unitary matrix,  $U$ , describing the operation of a quantum computer can be described as matrix products of canonical slice equations.*

According to theorem 4.2.1, any unitary matrix can be divided into multiple slices. After sufficient decompositions, we'll be left single gate slices which can be algebraically modeled by our canonical expression, that is,  $S = P + P_c \times C$  or  $S = I - P_c + P_c \times C$ . We can then matrix multiply our slices to form  $U$ . That is, if  $U$  was decomposed into  $L$  slices,

$$U = \prod_{i=L}^1 S_i = S_L \times S_{L-1} \times \cdots \times S_1 \quad (4.5.23)$$

## 4.6 Creating the Method

To show that the method is general, we need to show that it is equivalent to the canonical form we defined in the previous section. We'll first show how the definitions of a logic slice can be made equivalent, and then show how the terms themselves derived from STEP 2 and STEP 3 implement equation 4.5.8.

We had stipulated that the method works on what we call a *method legal slice*. In our canonical representation, we had more stringent criteria for a slice. The only difference is that in the canonical slice, we only allowed a single gate. Whereas, a method legal slice can have more than one gate as long as they all shared the same control nodes. The extension is perfectly legal in the canonical case since the true requirement is that – according to our rules

for partial identity matrices –  $P_c$  and  $P$  are complements and  $P_c$  and  $C$  are independent. If we introduce another control gate which shares the same qubits for control, we can essentially treat it as a single multi-input gate, even if it doesn't operate on qubits that are next to each other. Visually this doesn't look appealing, but mathematically it works.

We'll illustrate this with an example as shown in figure 4.8. The slice  $S$  is

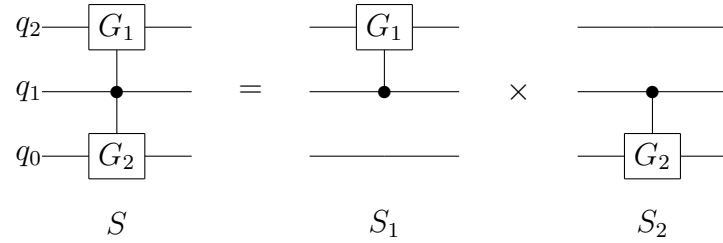


Figure 4.8: Method legal slice versus canonical slice:  $S = S_2 \times S_1 = S_1 \times S_2$

a method legal slice where using our method we'll quickly find the algebraic equation to be  $S = I \otimes D_0 \otimes I + G_1 \otimes I \otimes G_2$ . We use theorem 4.2.1 to create  $S_1$  and  $S_2$  which are legal for our canonical representation. We'll note that  $P$  for both  $S_1$  and  $S_2$  is the same with  $P = I \otimes D_0 \otimes I$ . Also,  $P_c$  is the same for  $S_1$  and  $S_2$  with  $P_c = I \otimes D_1 \otimes I$ . Lastly,  $C$  for  $S_1$  equals  $C_1 = G_1 \otimes I \otimes I$  and  $C$  for  $S_2$  equals  $C_2 = I \otimes I \otimes G_2$ . It can be easily shown that  $S_2 \times S_1 = (P + P_c \times C_2) \times (P + P_c \times C_1) = S$ . Therefore, because  $C_1$  and  $C_2$  are independent and share the same control nodes — meaning,  $P_c$  and  $P$  are the same — we can extend the canonical representation to include slices with multiple gates.

In **STEP 1** we identify the number of terms in as  $N + 1$  if  $N$  is the number of control nodes. The first  $N$  terms come directly from lemma 3 which states that  $P$  has the same number of terms as control nodes. The last  $N + 1$  term comes from lemma 2 which shows that  $P_c$  can be reduced to a single term. In which case, the product  $P_c \times C$  then becomes a single term. This is how we

derived STEP 1.

In **STEP 2** we begin defining our first  $N$  terms. After working out a few examples it becomes quite evident that we are solving for  $P$  as defined in equation 4.5.9 and using the optimized form as proposed in lemma 3. The method asks us to reference table 4.1 which is essentially the same as table 4.2 used for lemma 3. That said, if we knew  $P_c$ , we could replace STEP 2 with  $I - P_c$  according to equation 4.5.13. This would be advantageous if we have more than two control nodes in a slice.

In **STEP 3**, we are defining the last term which we will realize is equivalent to  $P_c \times C$ . According to the definition of  $P_c$  in equation 4.5.10, we have an identity term wherever there is a gate or a wire. Because of lemma 1, wherever  $P_c$  has an identity term, there is a corresponding identity term or gate term in  $C$ . This is implemented in STEP 3, rule 2. The  $D_1$  terms (or  $D_0$  for negative controls) in the  $P_c$  are implemented in STEP 3 by rule 3.

Remaining is the handling of non-controlled gates. Our canonical representation explicitly places such gates in a slice of their own. However, our method conveniently incorporates them into STEP 2 and STEP 3. The extension to the canonical method is also quite easy. If we had a slice,  $S$ , that is legal for our canonical method and we add a non-controlled gate which is independent with the slice (that is,  $G_u$  doesn't operate on qubits used for controls or controlled gates), then we can express it as  $S' = S \times (I \otimes \cdots G_u \cdots \otimes I) = (P + P_c \times C) \times (I \otimes \cdots G_u \cdots \otimes I)$ . This also means  $G_u$  is independent with  $P$ ,  $P_c$ , and  $C$ . We take advantage of this property in the method where replacing qubit terms with non-controlled gate matrices in both STEP 2 and STEP 3 is the same as the result of multiplying the slice without the non-controlled gate



with a slice with only the non-controlled gate.

#### 4.7 Examples

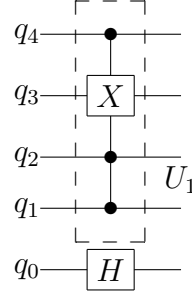


Figure 4.9: The quantum circuit for Example 1.

**Example 1:** Consider the single slice quantum circuit shown in Figure 4.9 which has 3 control nodes on qubits 4, 2, and 1, and therefore its expression will have 4 terms. We also note that this slice meets our requirement that all control nodes are connected to the same controlled-gates. In this case, all 3 control nodes connect to  $X$  on qubit 3.

We can easily apply rules 1 and 2 from STEP 2 by substituting  $X$  for  $I$  in qubit 3's position, and  $H$  for itself (since it is un-controlled) in qubit 0's position for the first 3 terms. This gives us our first glimpse of our algebraic expression below.

$$S = \underbrace{\# I \# \# H}_{T_1} + \underbrace{\# I \# \# H}_{T_2} + \underbrace{\# I \# \# H}_{T_3} + \underbrace{\# \# \# \# \#}_{T_4}$$

Applying STEP 2 is more methodical and the process becomes quite easy after a few examples. We begin with the first term and we can use table 1 to substitute the  $\#$ 's or we can work it out by hand. The process for this example

is as follows:

- **TERM 1:** The highest order control node is  $q_4$  so we replace the  $\#$  in qubit 4's position with  $D_0$ . The rest of the control nodes will be represented by  $I$ .
- **TERM 2:** We promote qubit 4 from  $D_0$  to  $D_1$ . The next highest control node is on  $q_2$  so we set it to  $D_0$ . The remaining control node on  $q_1$  remains  $I$ .
- **TERM 3:** Nothing is done to  $q_4$  since it remains  $D_1$ . The control node on  $q_2$  is promoted from  $D_0$  to  $D_1$ , and finally we can promote  $q_1$  to  $D_0$ .

After the above steps, our new equation becomes:

$$S = D_0 I I I H + D_1 I D_0 I H + D_1 I D_1 D_0 H + \# \# \# \# \#$$

Lastly, applying STEP 3 is the final and easiest step. In the last term, all gates are represented by themselves and all control nodes are fully promoted to  $D_1$ . Thus leaving us the final equation for our slice,  $S$ , and since this is a single slice circuit,  $U = S$ . Therefore,

$$U = D_0 I I I H + D_1 I D_0 I H + D_1 I D_1 D_0 H + D_1 X D_1 D_1 H$$

The  $32 \times 32$  unitary matrix,  $U$ , is constructed by performing the indicated Kronecker products followed by the matrix summations. It is also of interest to show that this equation can be simplified to a parallel combination of the controlled logic depicted as  $U_1$  in figure 4.9 and the Hadamard gate:

$$\begin{aligned}
U &= (D_0 III + D_1 I D_0 I + D_1 I D_1 D_0 + D_1 X D_1 D_1) H \\
&= U_1 H
\end{aligned}$$

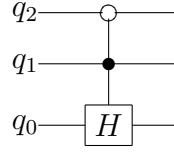


Figure 4.10: The quantum circuit for Example 2. The open circle is the negative control.

**Example 2:** Figure 4.10 shows a quantum circuit with a negative and a positive control node. The procedure described above is only slightly changed to handle negative control nodes. Referring to Table 1, the  $\#$  symbols are initially replaced by  $D_0$ , but in subsequent terms the  $D_0$  is promoted to  $D_1$ . This order is reversed for negative controls. That is, the replacement starts with  $D_1$  but in subsequent terms  $D_1$  is promoted to  $D_0$ . The replacements for positive controls are the same as before.

$$U = D_1 I I + D_0 D_0 I + D_0 D_1 H$$

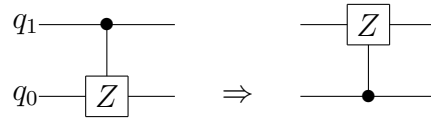


Figure 4.11: The quantum circuit for Example 3.

**Example 3:** This example applies our method to a well-known equivalent circuit pair. The LHS expression is

$$D_0I + D_1Z$$

while the RHS expression is

$$ID_0 + ZD_1$$

Using the properties  $I = D_0 + D_1$  and  $Z = D_0 - D_1$ , the RHS can be expanded to

$$(D_0 + D_1)D_0 + (D_0 - D_1)D_1$$

Further expanding yields

$$D_0D_0 + D_1D_0 + D_0D_1 - D_1D_1$$

Collecting terms 1 and 3 together and terms 2 and 4 together, we get

$$D_0(D_0 + D_1) + D_1(D_0 - D_1)$$

Which simplifies to the RHS

$$D_0I + D_1Z$$

**Example 4:** This last example shows how to handle multi-input quantum gates. Our rules (specifically #2) are general enough to cover this situation. Since there is only 1 control node the equation has only two terms. The controlled gate  $B$  is a 2 qubit gate and will thus be replaced by  $II$  in the first term and by itself in the second term. The algebraic expression for the quantum

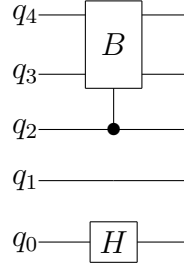


Figure 4.12: Quantum circuit for Example 4.

circuit of Figure 4.12 is

$$U = \text{II} D_0 \text{IH} + \text{B} D_1 \text{IH}$$

You'll note that the final equation is indeed a 5 qubit unitary gate.

#### 4.8 Extension to Multi-valued Logic

In multi-valued quantum logic [71], qubits are replaced by more general *qudits*. A single  $n$ -logic qudit gate is represented by an  $n \times n$  matrix. Without diving into too much detail, our method can be naturally extended to handle multi-value logic by introducing more  $D$ -terms. Whereas for binary quantum logic we had only two  $2 \times 2$   $D$ -terms ( $D_0$  and  $D_1$ ), in  $n$ -value logic we have  $n$   $D$ -terms ( $D_0, D_1, \dots, D_{n-1}$ ) of dimension  $n \times n$ . For example, if we have tertiary logic (using so-called *qutrits*), then we have three  $3 \times 3$   $D$ -terms:

$$D_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad D_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad D_2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Our method and canonical representations still apply. As an example, we can derive the expression for the generalized  $n$ -logic two-qudit controlled gate shown in figure 4.13. In this example, the gate,  $G$ , is activated when qudit  $q_1$  is asserted

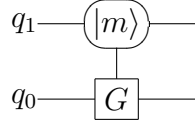


Figure 4.13: Generalized two-qudit controlled gate example

to the state  $|m\rangle$ , where  $m$  is one of the  $n$  logic states:  $0, 1, \dots$ , or  $n - 1$ . The equation for this circuit is best represented using our alternative canonical representation from equation 4.5.13:

$$U = I \otimes I - D_m \otimes I + D_m \otimes G \quad (4.8.1)$$

## Chapter 5

### Evolving Unitary Operators

Recent work has shown that evolutionary algorithms can be effectively used to help design quantum computers. For example, genetic algorithms have been used for gate-level synthesis of a quantum computer[43]. However, we take a different approach by using an *Evolution Strategy* — or ES — as described in section 2.3.2 to directly evolve unitary operators which represent solutions to problem instances.

An ES represents individuals as a vector of real values. In particular, the genotype has the structure:  $\langle x_1, x_2, \dots, x_n, \sigma_1, \sigma_2, \dots, \sigma_n \rangle$ , where the first set,  $\{x\}$ , are the *object parameters* and the second set,  $\{\sigma\}$ , are the *strategy parameters*. The object parameters are the basic parameters needed to define the individual. In our case, the  $\phi$ ,  $\psi$ , and  $\chi$  angles used in our random unitary matrix method described in section

The initial state of the quantum system is represented by an  $N$ -element

column vector, which has an initial state

$$|\psi_0\rangle = \frac{1}{\sqrt{N}} \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \quad (5.0.1)$$

This particular initialization indicates the initial state of the quantum system is a linear superposition of all possible states with equal probability. Once the unitary operator ( $U$ ) is constructed, the operation

$$U |\psi_0\rangle = |\psi\rangle \quad (5.0.2)$$

produces  $|\psi\rangle$ , which is the result of the quantum computation which the evolved  $U$  described. Each individual in the ES represents a unique operator  $U$ .

The fitness of an individual can be calculated depending on whether our target vector is a *pre-measurement* or *post-measurement* state. Pre-measurement uses the complex state vector ( $|\psi\rangle$ ) produced by equation (5.0.2) whereas post-measurement refers to the final (or classical) real-valued state the quantum system collapses to after an observation is taken. The output state is interpreted as probability distribution. The probability distribution vector is the modulus-squared of the state vector ( $|\psi|^2$ ) which can be compared to a target probability distribution vector.

Given an individual with unitary operator  $U$ , an initial state vector  $|\psi_0\rangle$  and



a target final state vector  $|\psi'\rangle$ , fitness is calculated with the function

$$\text{fitness}(U) = \sum_0^{N-1} \frac{1}{(|c_i|^2 - |c'_i|^2)^2 + \epsilon} \quad (5.0.3)$$

Where  $c_i$  is the  $i^{\text{th}}$  complex component in the resultant state vector as described in equation (5.0.2) and  $c'_i$  is the  $i^{\text{th}}$  complex element in the target state vector. The fitness equation above is an example of using probability distribution method since it involves the modulus-squared of complex elements. When designing quantum gates or other sub-circuits, the complex behavior is important so using a pre-measurement type fitness function would be more appropriate.

## 5.1 Constructing the ES

The construction of the ES is a signification portion of our research and provided much of the data and learnings. In this section we'll go into the details about the actual ES.

There are a number of parameters to consider when designing an ES to obtain near-optimal results. For example,

1.  $(\mu + \lambda)$  or  $(\mu, \lambda)$ ?
2. The size  $\mu$  and  $\lambda$ .
3. Initial and reset values of strategy parameters  $(\{\sigma\})$ .
4. Balance between probability of Mutation and Recombination.
5. Convergence Behavior.
6. Best Convergence.

We will also cover enhancement decisions in the ES — examples being the choice of fitness function, probabilistic versus complex targets, handling multiple input/output pairs, hybrid-algorithms, and tweaks to the algorithm such as rounding angles — all in the hope to get better convergence behavior. Of course, as much as we would like a one-size-fits-all solution, it is understandable that certain problem families benefited from certain ES configurations and not for others. For example, single and two qubit problems converged very quickly and had a weak dependence to many parameters. Whereas, large qubit problems (which for us, is on the order of 4 or 5 qubits) would not converge without significant tweaking of the parameters.

The high-level flow diagram is shown in figure 5.1. We will often refer to the “Best” individual as the current most fit individual for a given run and the “Very Best” individual as the most fit individual over all runs. We now take a closer glance at the mutation/recombination and fitness calculation steps as these are special to our ES implementation.

### 5.1.1 Representation of an Individual

We refer back to section 3.2 which describes how to generate random unitary matrices and specifically equation 3.2.4 which we restate here:

$$U = E_1 E_2 E_3 \dots E_{N-1} \tag{5.1.1}$$

Using this method to generate an  $N = 2$  (that is, single qubit) matrix results

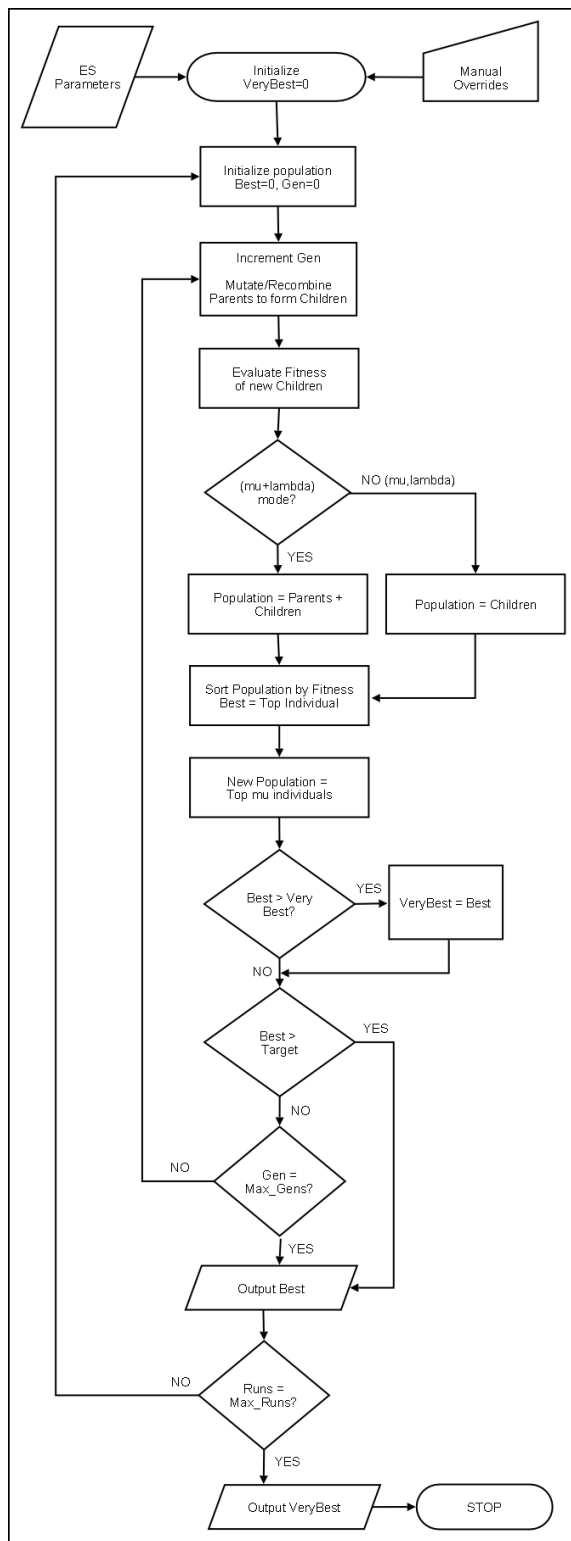


Figure 5.1: ES Flow Chart

in the following symbolic matrix:

$$\begin{pmatrix} \cos(\phi_{1,2})e^{i\psi_{1,2}} & \sin(\phi_{1,2})e^{i\chi_{1,2}} \\ -\sin(\phi_{1,2})e^{-i\chi_{1,2}} & \cos(\phi_{1,2})e^{-i\psi_{1,2}} \end{pmatrix} \quad (5.1.2)$$

After—which, larger qubit matrices start to dramatically increase in complexity. For example, we’ll show the two-qubit symbolic matrix here but we need to define each of its  $4 \times 4$  elements in a vertical list as writing out the matrix in standard form is too difficult due to the large terms for each element. Therefore, we define the unitary matrix using  $U[row][col]$  notation:

$$\begin{aligned}
U[0][0] &= \begin{aligned} &((((\cos(\phi_{1,2})e^{i\psi_{1,2}})\cos(\phi_{1,3})e^{i\psi_{1,3}} + ((\sin(\phi_{1,2})e^{i\chi_{1,2}}\sin(\phi_{2,3}) - \\ &\sin(\phi_{1,3})e^{-i\chi_{1,3}})))\cos(\phi_{1,4})e^{i\psi_{1,4}} + \\ &((((\sin(\phi_{1,2})e^{i\chi_{1,2}}\cos(\phi_{2,3})e^{i\psi_{2,3}})))\sin(\phi_{2,4}) + \\ &((((\cos(\phi_{1,2})e^{i\psi_{1,2}})\sin(\phi_{1,3})e^{i\chi_{1,3}} + \\ &((\sin(\phi_{1,2})e^{i\chi_{1,2}}\sin(\phi_{2,3}))\cos(\phi_{1,3})e^{-i\psi_{1,3}})\sin(\phi_{3,4}))\cos(\phi_{2,4})e^{-i\psi_{2,4}} - \\ &\sin(\phi_{1,4})e^{-i\chi_{1,4}}) \end{aligned} \\
U[0][1] &= \begin{aligned} &((((\sin(\phi_{1,2})e^{i\chi_{1,2}}\cos(\phi_{2,3})e^{i\psi_{2,3}})))\cos(\phi_{2,4})e^{i\psi_{2,4}} + \\ &((((\cos(\phi_{1,2})e^{i\psi_{1,2}})\sin(\phi_{1,3})e^{i\chi_{1,3}} + \\ &((\sin(\phi_{1,2})e^{i\chi_{1,2}}\sin(\phi_{2,3}))\cos(\phi_{1,3})e^{-i\psi_{1,3}})\sin(\phi_{3,4}) - \sin(\phi_{2,4}))) \end{aligned} \\
U[0][2] &= \begin{aligned} &((((\cos(\phi_{1,2})e^{i\psi_{1,2}})\sin(\phi_{1,3})e^{i\chi_{1,3}} + \\ &((\sin(\phi_{1,2})e^{i\chi_{1,2}}\sin(\phi_{2,3}))\cos(\phi_{1,3})e^{-i\psi_{1,3}})\cos(\phi_{3,4})e^{i\psi_{3,4}})) \end{aligned} \\
U[0][3] &= \begin{aligned} &((((\cos(\phi_{1,2})e^{i\psi_{1,2}})\cos(\phi_{1,3})e^{i\psi_{1,3}} + ((\sin(\phi_{1,2})e^{i\chi_{1,2}}\sin(\phi_{2,3}) - \\ &\sin(\phi_{1,3})e^{-i\chi_{1,3}})))\sin(\phi_{1,4})e^{i\chi_{1,4}} + \\ &((((\sin(\phi_{1,2})e^{i\chi_{1,2}}\cos(\phi_{2,3})e^{i\psi_{2,3}})))\sin(\phi_{2,4}) + \\ &((((\cos(\phi_{1,2})e^{i\psi_{1,2}})\sin(\phi_{1,3})e^{i\chi_{1,3}} + \\ &((\sin(\phi_{1,2})e^{i\chi_{1,2}}\sin(\phi_{2,3}))\cos(\phi_{1,3})e^{-i\psi_{1,3}})\sin(\phi_{3,4})) \\ &\cos(\phi_{2,4})e^{-i\psi_{2,4}})\cos(\phi_{1,4})e^{-i\psi_{1,4}}) \end{aligned} \\
U[1][0] &= \begin{aligned} &(((((-\sin(\phi_{1,2})e^{-i\chi_{1,2}})\cos(\phi_{1,3})e^{i\psi_{1,3}} + \\ &((\cos(\phi_{1,2})e^{-i\psi_{1,2}}\sin(\phi_{2,3}) - \\ &\sin(\phi_{1,3})e^{-i\chi_{1,3}})))\cos(\phi_{1,4})e^{i\psi_{1,4}} + \\ &((((\cos(\phi_{1,2})e^{-i\psi_{1,2}}\cos(\phi_{2,3})e^{i\psi_{2,3}})))\sin(\phi_{2,4}) + \\ &(((((-\sin(\phi_{1,2})e^{-i\chi_{1,2}})\sin(\phi_{1,3})e^{i\chi_{1,3}} + \\ &((\cos(\phi_{1,2})e^{-i\psi_{1,2}}\sin(\phi_{2,3}))\cos(\phi_{1,3})e^{-i\psi_{1,3}})\sin(\phi_{3,4}))\cos(\phi_{2,4})e^{-i\psi_{2,4}} - \\ &\sin(\phi_{1,4})e^{-i\chi_{1,4}}) \end{aligned} \\
U[1][1] &= \begin{aligned} &((((\cos(\phi_{1,2})e^{-i\psi_{1,2}}\cos(\phi_{2,3})e^{i\psi_{2,3}})))\cos(\phi_{2,4})e^{i\psi_{2,4}} + \\ &(((((-\sin(\phi_{1,2})e^{-i\chi_{1,2}})\sin(\phi_{1,3})e^{i\chi_{1,3}} + \\ &((\cos(\phi_{1,2})e^{-i\psi_{1,2}}\sin(\phi_{2,3}))\cos(\phi_{1,3})e^{-i\psi_{1,3}})\sin(\phi_{3,4}) - \sin(\phi_{2,4}))) \end{aligned} \\
U[1][2] &= \begin{aligned} &(((((-\sin(\phi_{1,2})e^{-i\chi_{1,2}})\sin(\phi_{1,3})e^{i\chi_{1,3}} + \\ &((\cos(\phi_{1,2})e^{-i\psi_{1,2}}\sin(\phi_{2,3}))\cos(\phi_{1,3})e^{-i\psi_{1,3}})\cos(\phi_{3,4})e^{i\psi_{3,4}})) \end{aligned} \\
U[1][3] &= \begin{aligned} &(((((-\sin(\phi_{1,2})e^{-i\chi_{1,2}})\cos(\phi_{1,3})e^{i\psi_{1,3}} + ((\cos(\phi_{1,2})e^{-i\psi_{1,2}}\sin(\phi_{2,3}) - \\ &\sin(\phi_{1,3})e^{-i\chi_{1,3}})))\sin(\phi_{1,4})e^{i\chi_{1,4}} + \\ &((((\cos(\phi_{1,2})e^{-i\psi_{1,2}}\cos(\phi_{2,3})e^{i\psi_{2,3}})))\sin(\phi_{2,4}) + \\ &(((((-\sin(\phi_{1,2})e^{-i\chi_{1,2}})\sin(\phi_{1,3})e^{i\chi_{1,3}} + \\ &((\cos(\phi_{1,2})e^{-i\psi_{1,2}}\sin(\phi_{2,3}))\cos(\phi_{1,3})e^{-i\psi_{1,3}}) \\ &\sin(\phi_{3,4}))\cos(\phi_{2,4})e^{-i\psi_{2,4}})\cos(\phi_{1,4})e^{-i\psi_{1,4}}) \end{aligned}
\end{aligned}$$

$$\begin{aligned}
U[2][0] &= \begin{aligned} &((((\cos(\phi_{2,3})e^{-i\psi_{2,3}} - \\ &\sin(\phi_{1,3})e^{-i\chi_{1,3}}))) \cos(\phi_{1,4})e^{i\psi_{1,4}} + \\ &(((((-\sin(\phi_{2,3})))) \sin(\phi_{2,4}) + \\ &((((\cos(\phi_{2,3})e^{-i\psi_{2,3}}) \cos(\phi_{1,3})e^{-i\psi_{1,3}}) \sin(\phi_{3,4})) \cos(\phi_{2,4})e^{-i\psi_{2,4}} - \\ &\sin(\phi_{1,4})e^{-i\chi_{1,4}}) \end{aligned} \\
U[2][1] &= \begin{aligned} &(((((-\sin(\phi_{2,3})))) \cos(\phi_{2,4})e^{i\psi_{2,4}} + \\ &((((\cos(\phi_{2,3})e^{-i\psi_{2,3}}) \cos(\phi_{1,3})e^{-i\psi_{1,3}}) \sin(\phi_{3,4}) - \\ &\sin(\phi_{2,4})) \end{aligned} \\
U[2][2] &= (((((\cos(\phi_{2,3})e^{-i\psi_{2,3}}) \cos(\phi_{1,3})e^{-i\psi_{1,3}}) \cos(\phi_{3,4})e^{i\psi_{3,4}}))) \\
U[2][3] &= \begin{aligned} &((((\cos(\phi_{2,3})e^{-i\psi_{2,3}} - \\ &\sin(\phi_{1,3})e^{-i\chi_{1,3}}))) \sin(\phi_{1,4})e^{i\chi_{1,4}} + \\ &(((((-\sin(\phi_{2,3})))) \sin(\phi_{2,4}) + \\ &((((\cos(\phi_{2,3})e^{-i\psi_{2,3}}) \cos(\phi_{1,3})e^{-i\psi_{1,3}}) \sin(\phi_{3,4})) \\ &\cos(\phi_{2,4})e^{-i\psi_{2,4}}) \cos(\phi_{1,4})e^{-i\psi_{1,4}}) \end{aligned} \\
U[3][0] &= \begin{aligned} &(((\cos(\phi_{3,4})e^{-i\psi_{3,4}}) \cos(\phi_{2,4})e^{-i\psi_{2,4}} - \\ &\sin(\phi_{1,4})e^{-i\chi_{1,4}}) \end{aligned} \\
U[3][1] &= \begin{aligned} &(((\cos(\phi_{3,4})e^{-i\psi_{3,4}} - \\ &\sin(\phi_{2,4})) \end{aligned} \\
U[3][2] &= ((((-\sin(\phi_{3,4})))) \\
U[3][3] &= (((\cos(\phi_{3,4})e^{-i\psi_{3,4}}) \cos(\phi_{2,4})e^{-i\psi_{2,4}}) \cos(\phi_{1,4})e^{-i\psi_{1,4}})
\end{aligned}$$

Although many terms zero-out during the matrix multiplication process, we see that the symbolic equations become quite cumbersome even for a moderately sized matrix. It becomes quite obvious that the calculation must be done iteratively and that pre-simplification (that is, having a pre-determined symbolic representation of the matrix) is out of the question.

Now we refer back to section 2.3.2 which describes Evolution Strategies which is the method we've used to evolve quantum operators. In particular, we're using the multi-strategy parameter method where we have a strategy parameter for each object parameter. Recall the generic representation for the multi-strategy parameter case is as defined in equation 5.1.3:

$$\langle \vec{x}, \vec{\sigma} \rangle = \langle x_0, \dots, x_n, \sigma_0, \dots, \sigma_n \rangle \quad (5.1.3)$$

Our angles used in the random unitary matrix method will be our object parameters for our ES representation. For an  $N \times N$  matrix, we'll need  $(N - 1)N/2$   $\phi$  angles,  $(N - 1)N/2$   $\psi$  angles, and  $N - 1$   $\chi$  angles giving us a total of  $N^2 - 1$  angles. Therefore, we will have  $N^2 - 1$  object parameters and  $N^2 - 1$  strategy parameters. This means the representation for our individual has  $2N^2 - 2$  parameters total where  $N = 2^{\text{qubits}}$ . This is definitely a lot and higher qubits will create a very large configuration space for our ES to search.

To demonstrate this, we look at the general angle requirements for an  $N \times N$  matrix:

$$\{\psi\} = \{\psi_{1,2}, \dots, \psi_{1,N}, \psi_{2,3}, \psi_{3,4}, \dots, \psi_{N-1,N}\} = (N - 1)N/2 \text{ angles}$$

$$\{\phi\} = \{\phi_{1,2}, \dots, \phi_{1,N}, \phi_{2,3}, \phi_{3,4}, \dots, \phi_{N-1,N}\} = (N - 1)N/2 \text{ angles}$$

$$\{\chi\} = \{\chi_{1,2}, \chi_{1,3}, \chi_{1,4}, \dots, \chi_{1,N}\} = N - 1 \text{ angles}$$

As an example, let's investigate the single qubit representation. The list of

angles is fairly small since  $N = 2$  for one qubit. Therefore, we need one of each angle as shown in equation 5.1.2 and are listed below:

$$\psi = \psi_{1,2}$$

$$\phi = \phi_{1,2}$$

$$\chi = \chi_{1,2}$$

To verify, when  $N = 2$ , we indeed have a total of three ( $N^2 - 1 = (2)^2 - 1 = 3$ ) angles.

Therefore, the final representation of a single qubit individual is done using the following structure:

$$\langle \vec{x}, \vec{\sigma} \rangle = \langle \psi_{1,2}, \phi_{1,2}, \chi_{1,2}, \sigma_{\psi_{1,2}}, \sigma_{\phi_{1,2}}, \sigma_{\chi_{1,2}} \rangle \quad (5.1.4)$$

As another example, the 2 qubit case would have  $2^4 - 1 = 15$  angle parameters along with an additional 15 strategy parameters. The list of angles would be:

$$\psi = \psi_{1,2}, \psi_{1,3}, \psi_{1,4}, \psi_{2,3}, \psi_{2,4}, \psi_{3,4}$$

$$\phi = \phi_{1,2}, \phi_{1,3}, \phi_{1,4}, \phi_{2,3}, \phi_{2,4}, \phi_{3,4}$$

$$\chi = \chi_{1,2}, \chi_{1,3}, \chi_{1,4}$$

As can be seen for larger quantum operators, the number of angles scale with the size of the matrix. For example, an 8 qubit quantum operator would be specified with a  $256 \times 256$  matrix and have 65535 angle parameters along with another 65535 strategy parameters yielding a representation vector with over 131 thousand floating-point elements.



## 5.2 Optimizing the ES Parameters

We now go through the exercise of adjusting the parameters of our ES so that we can have the best possible convergence behavior across many different problem sizes. We break this down into a few different sections. The study break is listed below:

- STUDY 1:  $\mu$  and  $\lambda$  for  $(\mu + \lambda)$  and  $(\mu, \lambda)$  schemes.
- STUDY 2: Mutation and Recombination.
- STUDY 3: Probabilistic versus Complex Targets.
- STUDY 4: Initial and Reset  $\sigma$ .
- STUDY 5: Fitness Calculation Methods.

For these studies, the ES was configured to evolve a unitary operator whose input vector,  $|\psi_0\rangle$  is the equal superposition state as shown in equation 5.0.1. The target output state is “state-0”, or  $|00\dots 0\rangle$ . Another way of showing this is that we are evolving  $U$  to meet the following equation:

$$\begin{pmatrix} u_{00} & u_{01} & u_{02} & \cdots \\ u_{10} & u_{11} & u_{12} & \cdots \\ u_{20} & u_{22} & u_{22} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \frac{1}{\sqrt{N}} \begin{pmatrix} 1 \\ 1 \\ 1 \\ \vdots \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \end{pmatrix} \quad (5.2.1)$$

We determine a target which can either be a fitness value as determined by a fitness function such as 5.0.3, or, we look at how closely the target state  $|00\dots 0\rangle$  is met by looking at the probability distribution. For example, if we set the target at 70%, then we say the ES has achieved a solution if in the

probability distribution, the first element,  $c_0$ , has a mod-squared value (that is,  $|c_0|^2$ ) of  $\geq 70\%$ .

### 5.2.1 Parent and Offspring Size Selection

In this study we look at how the size of the parent and child population impacts the performance of the ES. The number of parents is defined by  $\mu$  and the number of offspring is defined by  $\lambda$ . We also look at the two primary ES schemes of using  $(\mu + \lambda)$  where survivor selection includes individuals from both the offspring and parent pools, as well as  $(\mu, \lambda)$  where only the offspring are used for survivor selection.

In general, we want the population to be as small as possible. This is because there is a computation overhead for each individual which translates into run-time. On the other hand, we want fast convergence in terms of number of generations required to hit the target fitness.

For this study we only concentrate on the larger qubit studies which encompass the 4 and 5 qubit simulations. This is primarily because these are indicative of the more complex convergence problems and the lower qubit problems are somewhat insensitive to these parameters.

The previous graphs in figures 5.2, 5.3, and 5.4, show how  $\mu$  and  $\lambda$  selection impacts specific configurations. In the case, these configurations used  $(\mu + \lambda)$  mode and a probabilistic target vector.

Our inclination is to favor the cases where  $\mu = 10$ . In both the 4 and 5 qubit cases, it achieves the best convergence. Logically, as we increase the number of children, the number of generations to target is reduced. However, this comes at a computation time cost. The  $\mu = 10$  curves appear to taper off around

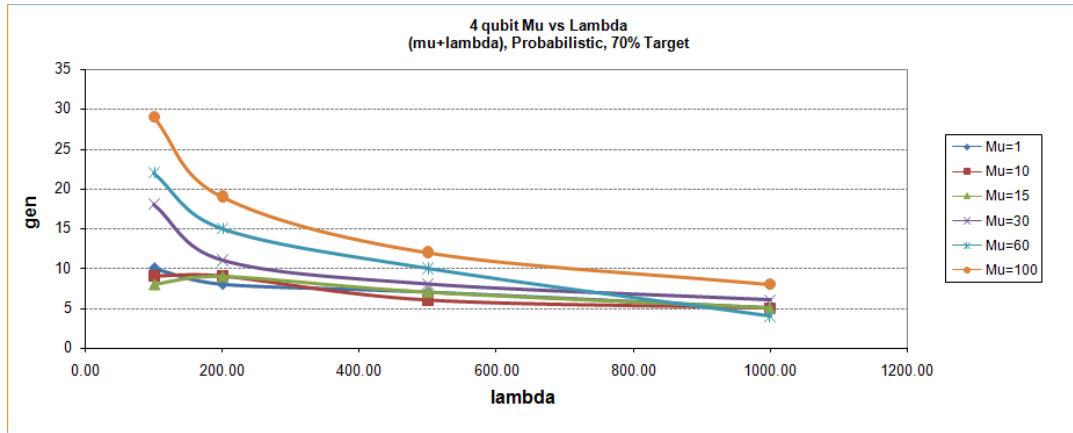


Figure 5.2:  $\mu$  vs.  $\lambda$ , 4 Qubits,  $(\mu + \lambda)$ , Probabilistic, 70% Target

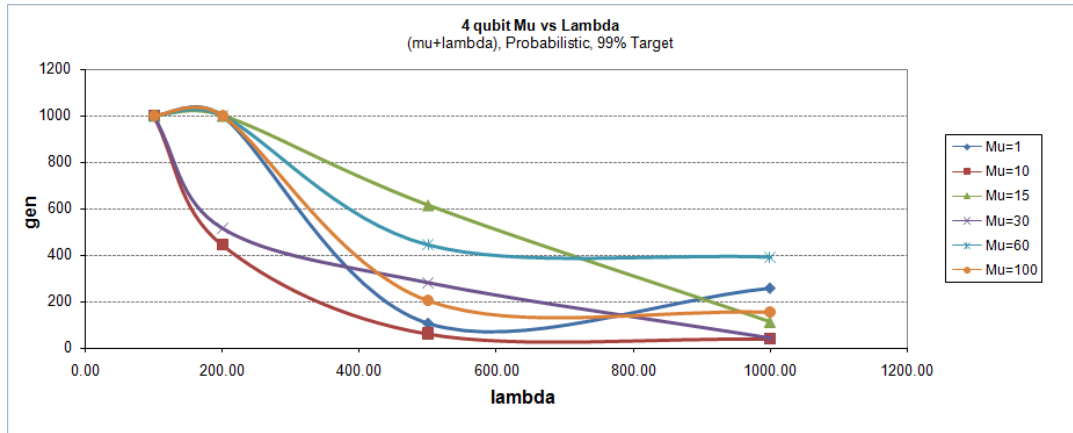


Figure 5.3:  $\mu$  vs.  $\lambda$ , 4 Qubits,  $(\mu + \lambda)$ , Probabilistic, 99% Target

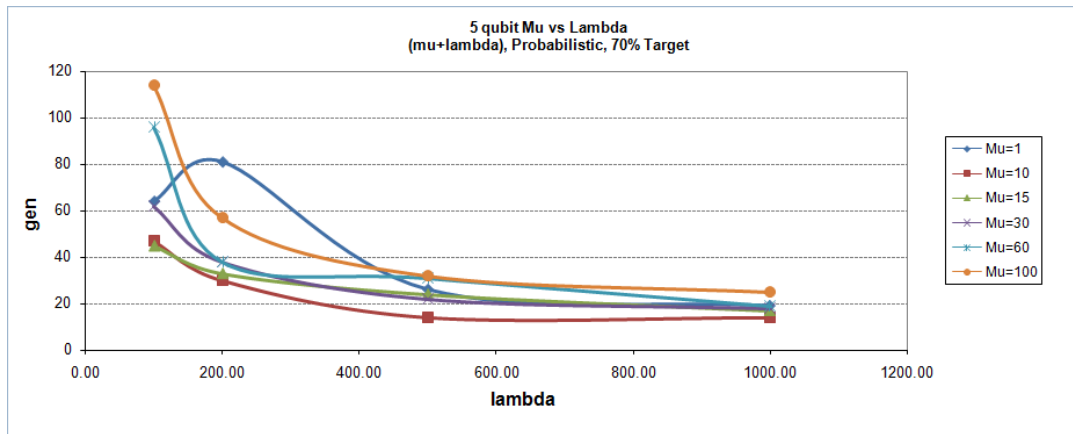


Figure 5.4:  $\mu$  vs.  $\lambda$ , 5 Qubits,  $(\mu + \lambda)$ , Probabilistic, 70% Target

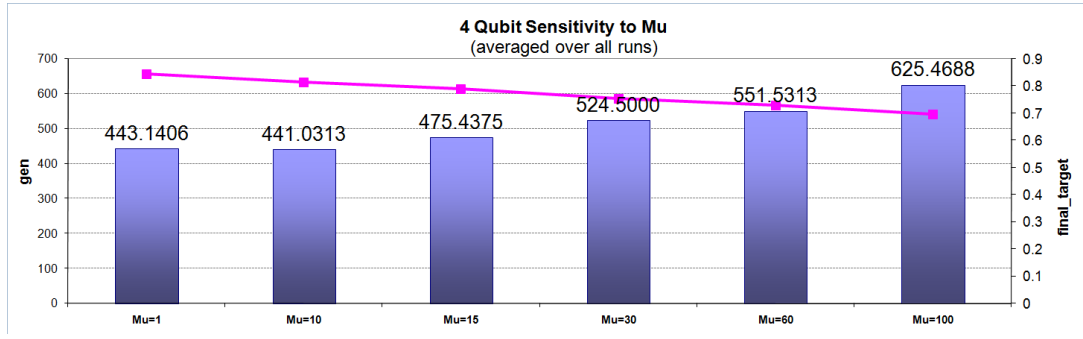


Figure 5.5:  $\mu$  vs.  $\lambda$ , 4 Qubits, Average over all runs

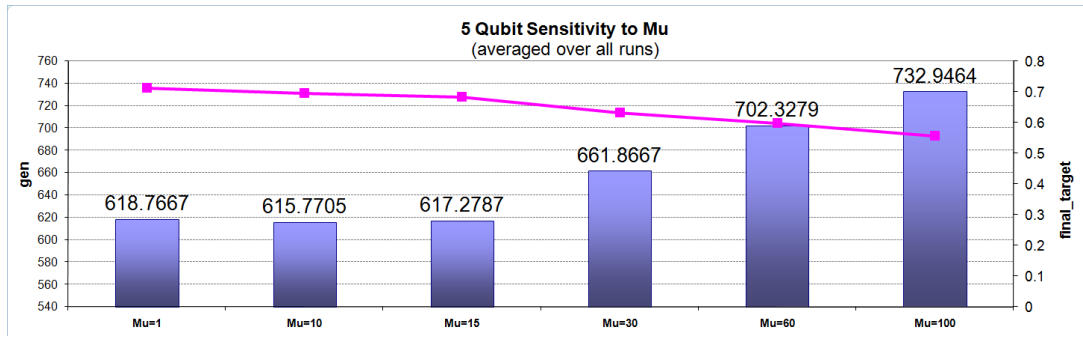


Figure 5.6:  $\mu$  vs.  $\lambda$ , 5 Qubits, Average over all runs

$\lambda = 500$ .

To further verify this assumption, we look at another couple of graphs in figures 5.5 and 5.6. These graphs look at the cases we simulated whether it is  $(\mu + \lambda)$  or  $(\mu, \lambda)$ , or probabilistic vs. complex targets, or 70% vs. 99% fitness targets, etc. In fact, this is averaged for all cases of  $\lambda$ . This was done on both the 4 and 5 qubit cases to see if we can generalize our statement that a  $\mu = 10$  and  $\lambda = 500$  configuration seems like a good configuration across the board.

Both graphs show that the average number of generations (that is, number of generations until the target fitness was reached) seems most optimal at the  $\mu = 10$  point.

We now examine if we should use  $(\mu + \lambda)$  or  $(\mu, \lambda)$ . As described in section

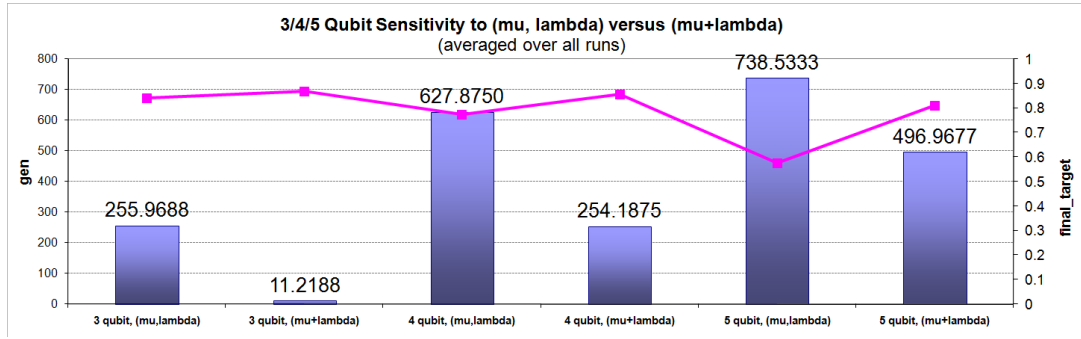


Figure 5.7: Comparison of  $(\mu, \lambda)$  to  $(\mu + \lambda)$  for 3, 4, and 5 qubits

2.3.2, there are advantages and disadvantages to each. The graph in figure 5.7 shows the average over all runs for 3, 4, and 5 qubits when either using  $(\mu + \lambda)$  or  $(\mu, \lambda)$ . There is a clear advantage of using  $(\mu + \lambda)$ .

At the end of this study we now have our optimum configuration going forward of using a  $(10 + 500)$  ES. The next study will focus on the comparison between using a complex versus a probabilistic target vector.

### 5.2.2 Mutation and Recombination

Although Evolution Strategies are typically driven primarily by mutation, recombination is also a fundamental feature that can be used. In this study we examine the pros and cons of using recombination.

We selected a  $(15 + 100)$  ES with a probabilistic target vector of  $|00 \dots 0\rangle$  for this study and implemented recombination in the ES. Recombination has two modes: *Intermediate* and *Discrete*. As described in section 2.3.2, intermediate recombination is when the allele is the resultant *average* of multiple parents. In discrete recombination, the allele value is chosen as a copy from only one of many potential parents.

Two different fitness targets are attempted: 70% and 90%. We only look at

two to five qubits. We are interested if there is a good mixture of mutation and recombination and so we varied the probability of mutation from 25% to 100%. The probability of recombination — that is, the probability that an allele will be recombined — was varied from 1% to 50%. Typically, mutation and recombination are mutually exclusive. An allele is either mutated or recombined but not both. Therefore, when the probability of mutation is 100%, no recombination occurs. Also for this study, we set the number of parents to two individuals.

Preliminary data indicated that recombination did not improve the results, and in fact, appeared to cause diverge behavior. It wasn't until a detailed investigation actually revealed pockets of highly optimal configurations where adding recombination actually resulted in more fit solutions. In this section we present a few of the many graphs showing our journey.

At this point we reference figure 5.8 which shows the sweep over all simulations and highlights the number of generations, the probability of mutation, and overlaps the recombination mode. A recombination mode of 0 is the discrete mode and a value of 1 is an intermediate mode. We note the circled areas around 50% and 75% mutation where we show better convergence than at 100% converge (which is at the far right of the graph). These are sweet-spots and deserve more investigation. There are a couple at the 50% mark, one in the discrete mode and the other in the intermediate mode. The 70% sweet-spot is in the discrete mode. The best solution used 50% mutation in the discrete mode.

We carried this similar examination to 4 qubits in figure 5.9. Although there are many configurations as good or better than 100% mutation, it appears to be insignificant with no outstanding result. However, in figure 5.10 we set the

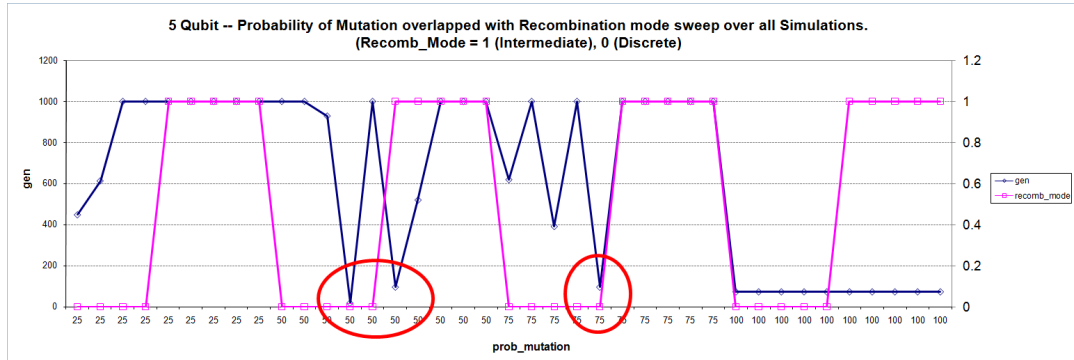


Figure 5.8: 5 Qubit Mutation/Recombination Sweep at 70% Target

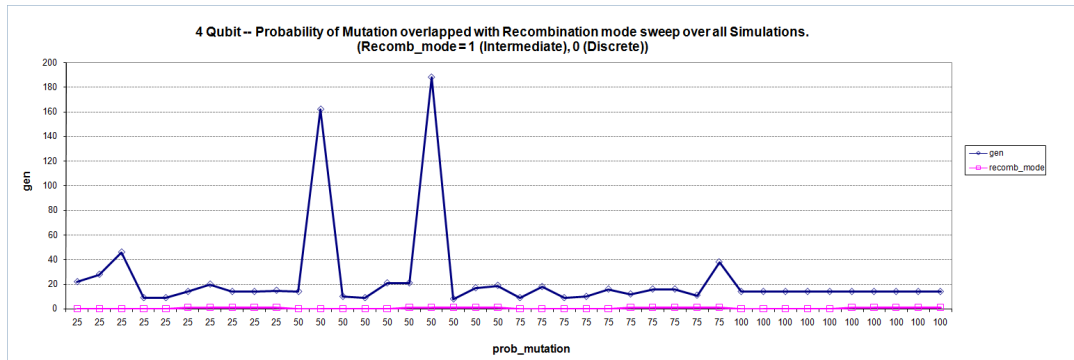


Figure 5.9: 4 Qubit Mutation/Recombination Sweep at 70% Target

target to 90% to push the ES further. Here we see many distinct pockets of highly optimal configurations such as at 25% and 75% mutation with discrete mode.

To dig deeper, we look at specific mutation/recombination configurations which we label in a “% Mutation / % Recombination” format and look at 3, 4, and 5 qubit results shown in figures 5.11 through 5.13. The final bar to the right is the 100% mutation case.

The clear configuration is in the 5 qubit case in figure 5.11 where we have a 50/50 (that is, 50% mutation and 50% recombination configuration. Here, we converge in 18 generations! Compared to the 73 generations with 100% mutation, this is phenomenal.

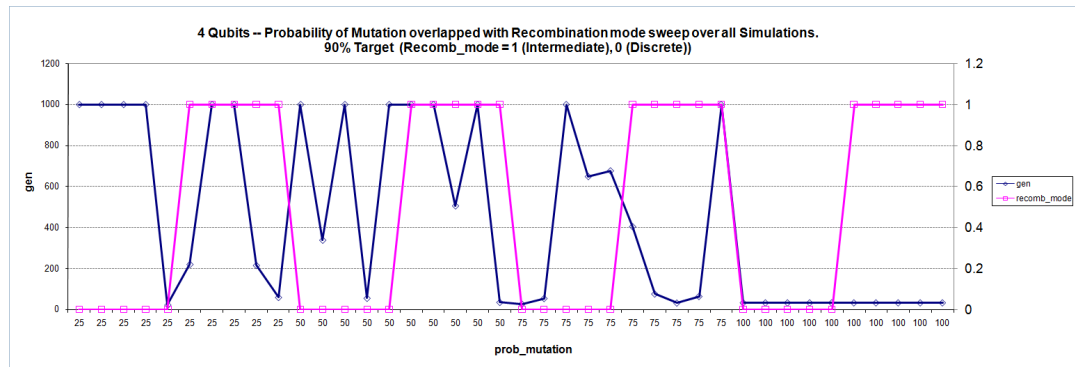


Figure 5.10: 4 Qubit Mutation/Recombination Sweep at 90% Target

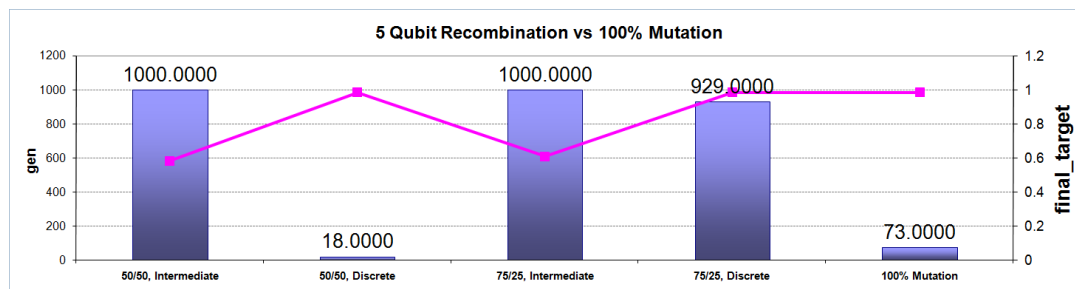


Figure 5.11: 5 qubit recombination versus 100% mutation

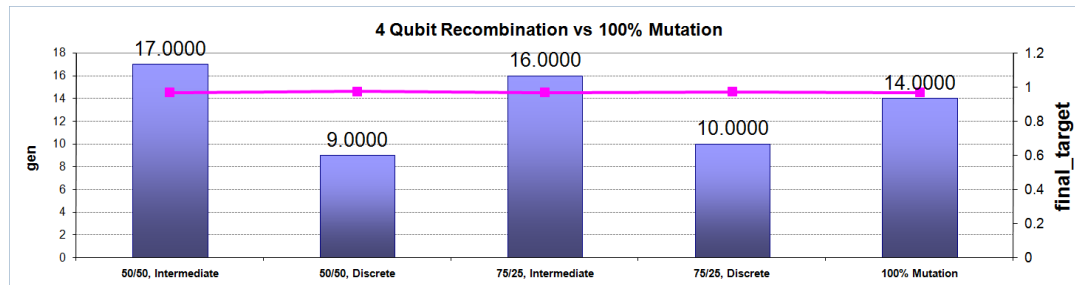


Figure 5.12: 4 qubit recombination versus 100% mutation

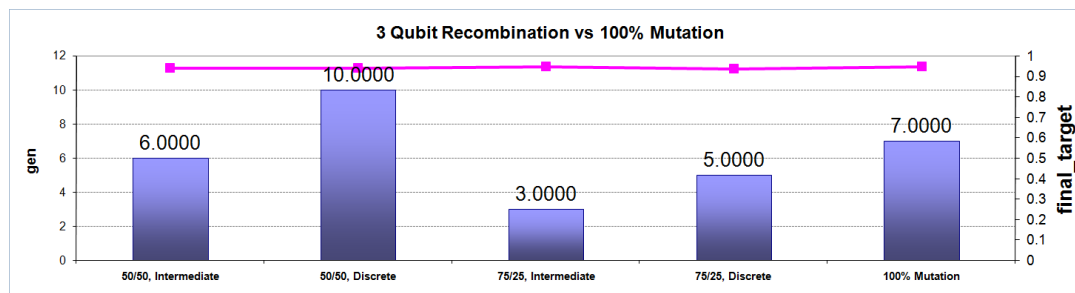


Figure 5.13: 3 qubit recombination versus 100% mutation



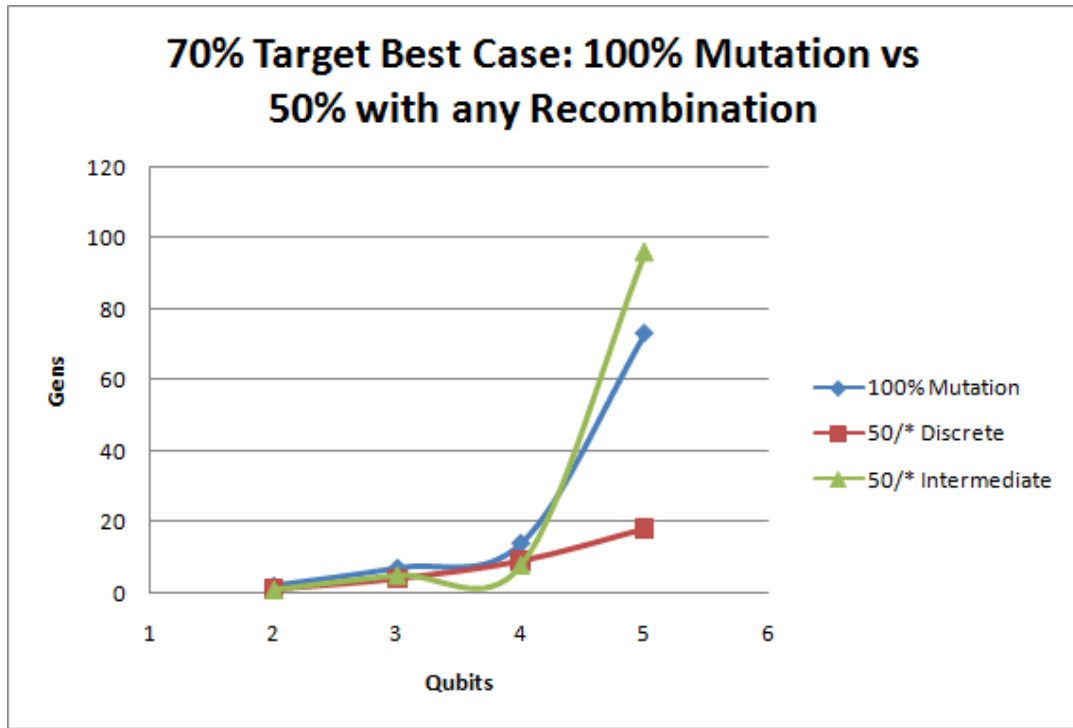


Figure 5.14: Best case 50% Mutation for 70% Target

As a final analysis of the benefits of recombination, we examine the best case results. That is, we look at what was the best simulation where 50% mutation was used and any amount of recombination. We compare that to the best 100% mutation case.

Figures 5.14 and 5.15 summarize the results for the 70% and 90% target cases, respectively. We also break-down the difference between discrete and recombination mode. These figures also indicate that intermediate mode had better convergence behavior for 90% targets. However, in the 70% case, discrete and intermediate yield about the same performance except for the 5 qubit case where discrete had a substantial advantage. Unfortunately, we were unable to converge within 1000 generations on the 5 qubit simulations with a 90% target.

The conclusion of this study is that 100% mutation is a good all-around

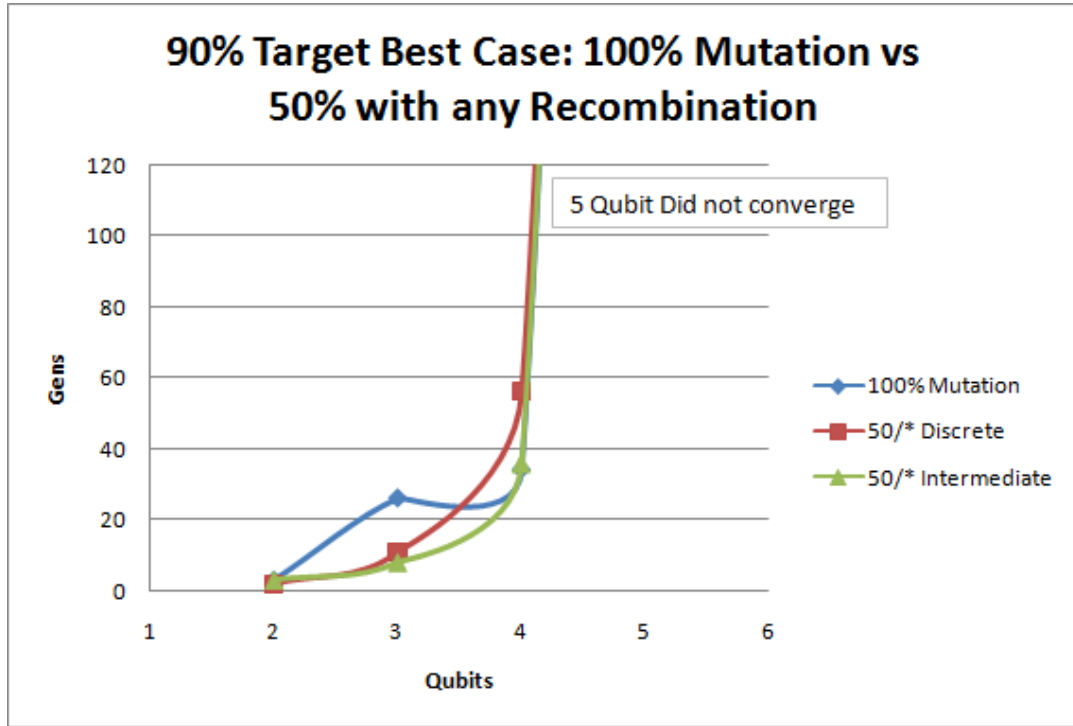


Figure 5.15: Best case 50% Mutation for 90% Target

configuration, yielding very good convergence. However, more optimal behavior can be found by adding recombination although it doesn't have as good of a "one-size-fits-all" nature as 100% mutation. The incredible advantage of the 50/50 case over the 100% case for 5 qubits should definitely prompt one to investigate the benefits of recombination.

### 5.2.3 Using a Complex versus Probabilistic Target Vector

The choice of whether to use a probabilistic (post-measurement) versus a complex (pre-measurement) target vector may depend on the particular type of quantum operator desired. For example, in many cases you want the complex behavior and in other cases it doesn't matter because the type of gate we want is actually implementing a measurement. Therefore, this study does not intend

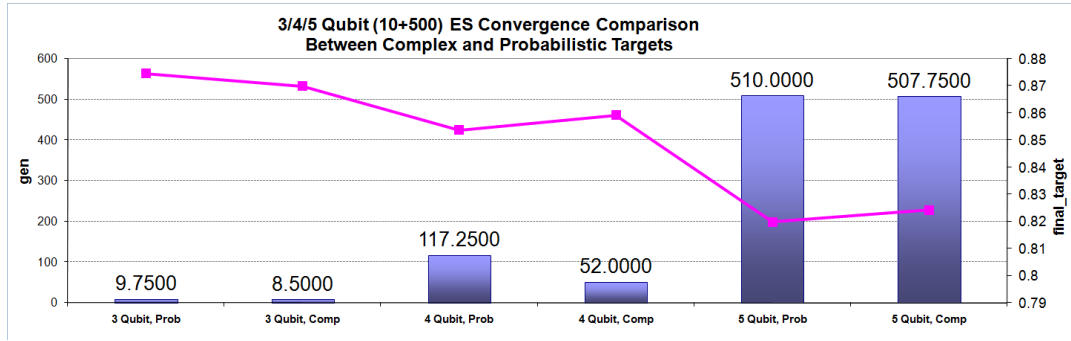


Figure 5.16: Probability vs. Complex Target for 3/4/5 Qubit (10+500) ES

to demonstrate which is better, but how their convergence properties compare.

Our intuition tells us that there should be a clear advantage of using a probabilistic target vector. By taking the mod-squared of the complex target, we essentially create more solutions for the same size state space. For example, if the complex result is a  $-1$  and the target is  $1$ , then we match. Obviously, this is not the case for pre-measurement.

Surprisingly, the graph in figure 5.16 shows us that for our (10, 500) ES, the advantage of using a post-measurement target is not that significant. Although it is a clear win for 4 qubit runs, 3 qubit and even 5 qubit runs didn't find a huge advantage. The average final target fitness was also mildly better in the 4 and 5 qubit cases when in the probability mode.

#### 5.2.4 Initial and Reset $\sigma$ Value

As described in section 2.3.2, the  $\sigma$  parameters represent the step-sizes for adjusting the object parameters. When the strategy parameters become too small, the object parameters will not change significantly enough to traverse the solution space and the strategy parameters will not adapt. Thus, the ES will no longer search that object parameter and we risk getting stuck in a highly

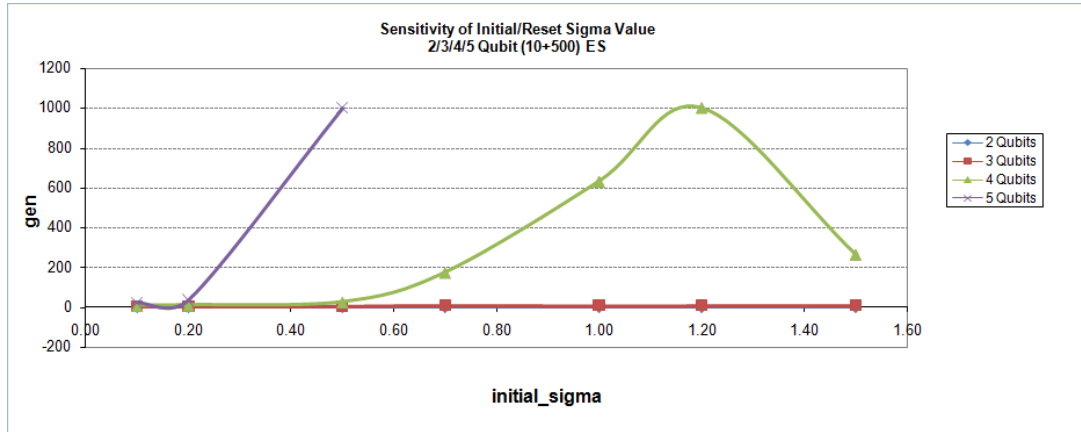


Figure 5.17: Initial/Reset  $\sigma$  Sensitivity for a 2-5 Qubit (10+500) ES

non-optimal state.

When we initialize an individual, we give them an initial  $\sigma$  value. That is, the initial value for the strategy parameters. We don't want that value too large or else we'll never converge and we don't want it too small or else we will pre-maturely converge. A similar event occurs when we hit the boundary condition.

In our ES, if we hit the boundary, we reset to the initial  $\sigma$  value. Thus, our reset and initial  $\sigma$  is the same value. Therefore, in this study, we present a series of results of simulations where the value of the reset/init  $\sigma$  has been varied and compare convergence behavior.

Simulations were run on 2, 3, 4, and 5 qubit systems with an initial/reset  $\sigma$  varying from 0.1 to 1.5. The graph in figure 5.17 shows that lower values of  $\sigma$  are desirable in all cases. The 4 qubit oddity with convergence time increasing until  $\sigma = 1.2$  then going down again is definitely interesting. However, the 5 qubit case shows a clear desire to keep the low initial/reset  $\sigma$  value around 0.1 as even slightly higher and the 5 qubit runs never end up converging (this is shown in the graph as hitting the 1000 generations mark which was the ES

limit). Based on this study, we recommend using a small initial/reset  $\sigma$  of 0.1.

### 5.2.5 Fitness Calculation Methods

The fitness function is important in any stochastic solution space searching paradigm. We gave an overview of the fitness function used in our ES at the beginning of this chapter. We recall the fitness equation used:

$$\text{fitness}(U) = \sum_0^{N-1} \frac{1}{(|c_i|^2 - |c'_i|^2)^2 + \epsilon} \quad (5.2.2)$$

Equation 5.0.3 repeated above is used in the mode where the target of interest is a probability distribution. Had we been interested in matching a desired complex behavior (that is, the target state is a pre-measurement state using complex values), then we would use equation 5.2.3 below:

$$\text{fitness}(U) = \sum_0^{N-1} \frac{1}{(c_i - c'_i)^2 + \epsilon} \quad (5.2.3)$$

We've already done a thorough comparison of the impact between the above fitness equations in our Probabilistic versus Complex target study. However, in this study we propose an alternative fitness function to 5.0.3 which uses the fitness as the 1 over the sum of the errors rather than the sum of 1 over the errors. This is noted in equation 5.2.4:

$$\text{fitness}(U) = \frac{1}{(\sum_0^{N-1} (|c_i|^2 - |c'_i|^2)^2) + \epsilon} \quad (5.2.4)$$

The advantage of this equation is that a perfectly fit individual will always have fitness =  $1/\epsilon$ . In our ES implementation, we use  $\epsilon = 1$ . Therefore, in equation 5.0.3, a perfectly fit individual has fitness =  $N$  and in equation 5.2.4

it would be 1. The other driving force of this simplification is for the case of multiple input and output vectors. In which case, further scaling of the fitness to get it to be in a  $[0, 1]$  boundary is required.

It is not obvious which equation is better and so we use this as part of this study. To get us started, we plot the fitness landscape for both equations 5.0.3 and 5.2.4 in figures 5.18 and 5.19, respectively. The  $x$  and  $y$  axes represent the error for the  $c_0$  and  $c_1$  components (assumed real in this case) for the single qubit state. The  $z$  axis represents the fitness value itself. The ideal solution is at the center of the  $xy$ -plane  $(0, 0)$ . In our case, the original equation 5.0.3 is not scaled so has a peak of  $z = 2$ . For the alternative fitness 5.2.4, the ideal point has  $z = 1$ .

It is interesting to note that equation 5.0.3 has highly fit solutions extending away from the origin (which is considered the perfect solution). Whereas, figure 5.19 shows the periphery is much more attenuated and focuses solutions into the center. With our original fitness equation, the heightened fitness along the  $x$  and  $y$  axes might encourage more individuals towards those points and then slowly towards the center. However, if the slope of the fitness landscape along those axes is not significant enough, it might make convergence time too long to reach the center. In the case of equation 5.2.4, the attenuated peripheral fitness slopes might inhibit outer individuals from finding the center solution, but those near the solution (center) might find it easier to migrate to the exact solution. So, both options have intuitive pros and cons.

We now look at some performance comparisons between the two functions. For the purposes of our graphs, we're calling our original function (5.0.3) “**F0**” and our alternative fitness function (5.2.4) “**F1**”. The graphs in figures 5.20

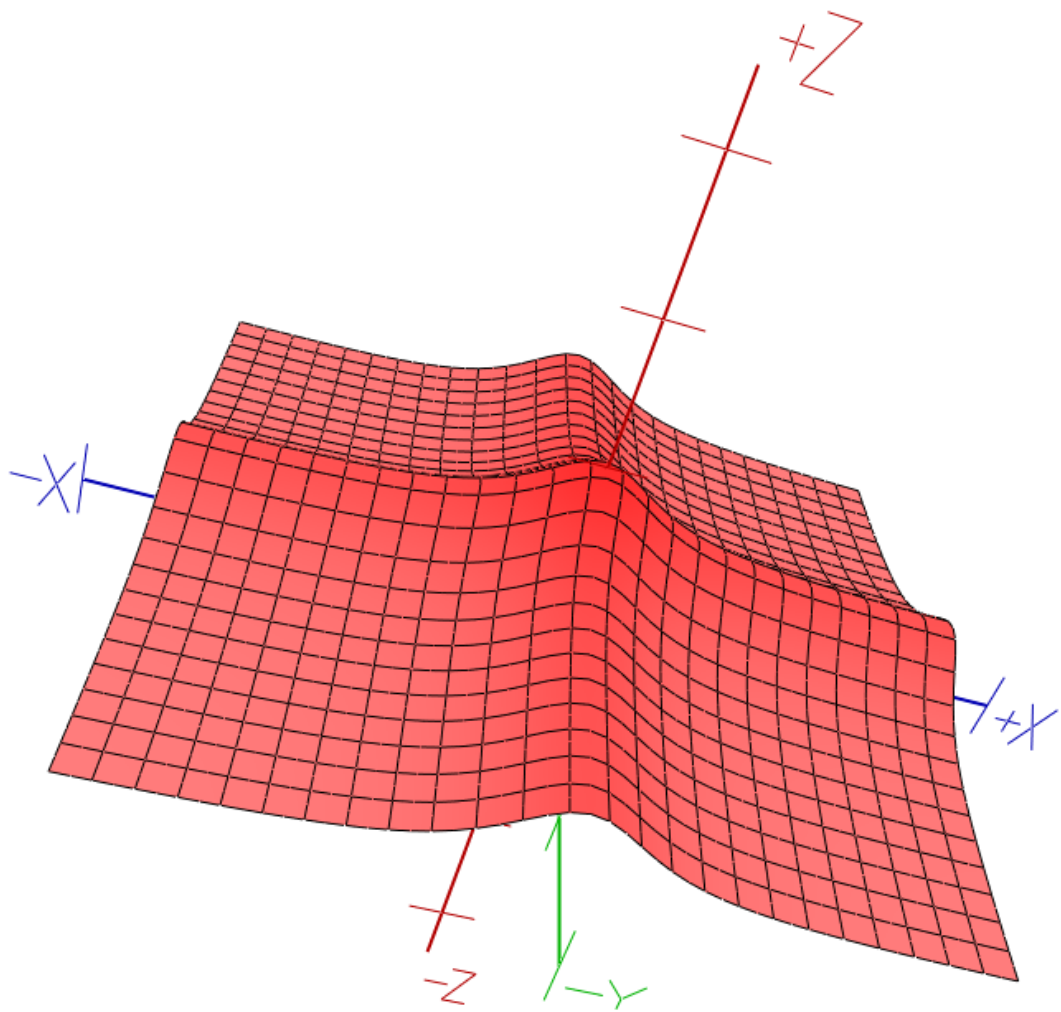


Figure 5.18: 1 Qubit Fitness Landscape plot for Fitness Equation 5.0.3

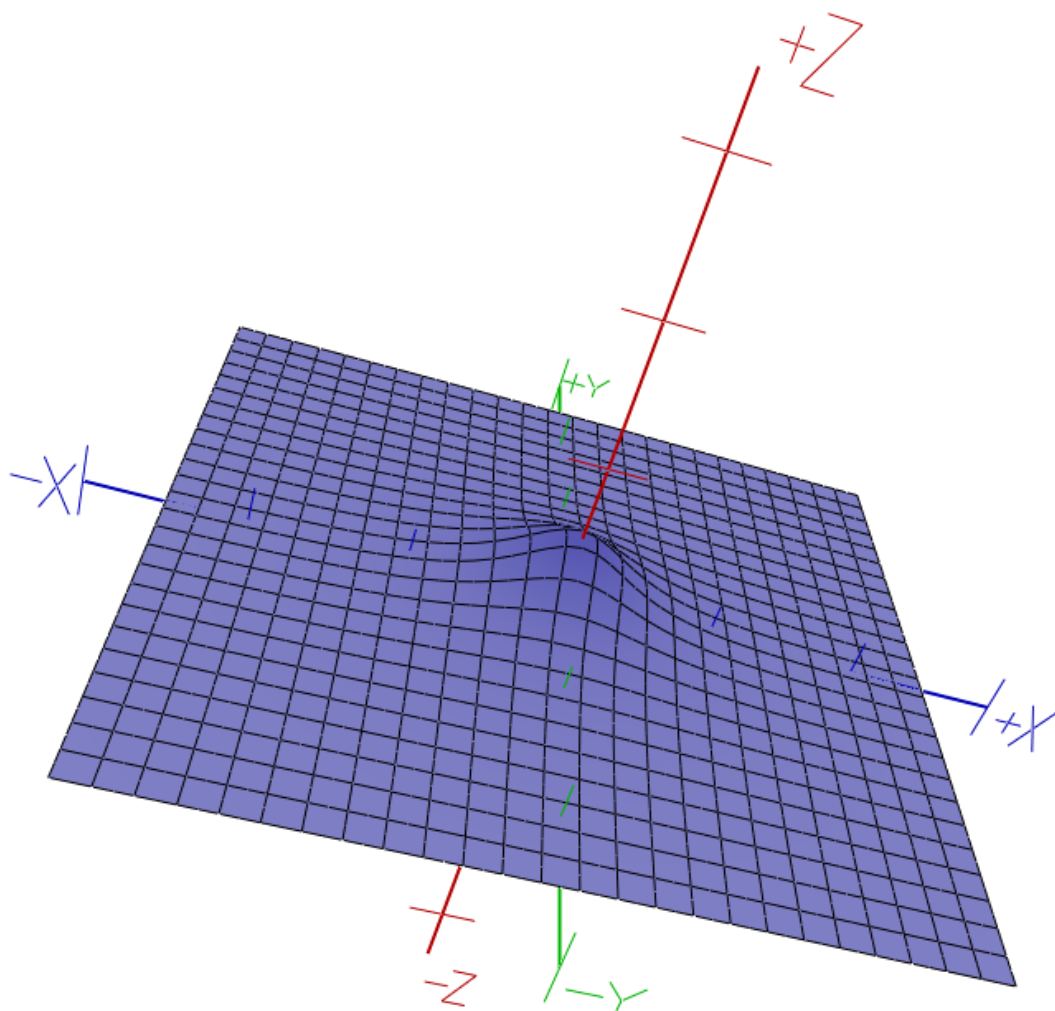


Figure 5.19: 1 Qubit Fitness Landscape plot for Fitness Equation 5.2.4



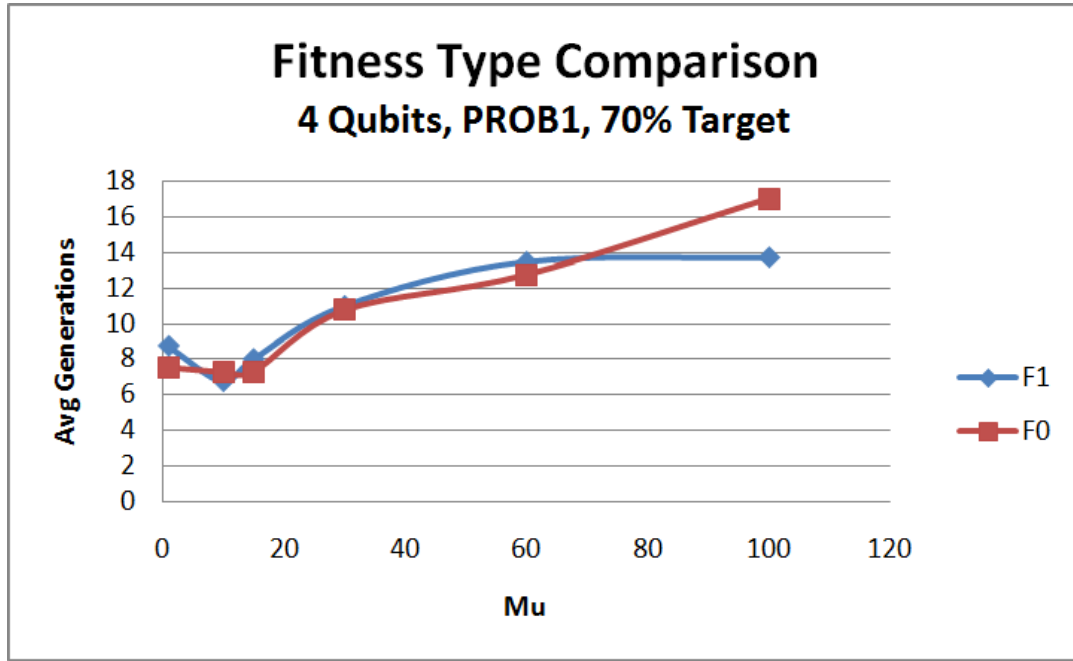


Figure 5.20: Fitness comparison for 4 Qubit, Probabilistic, 70% Target ES

through 5.22 show how the fitness function performed while varying  $\mu$  given a constant number offspring (in this case,  $\lambda = 500$ ).

Luckily, the results continue to emphasize that  $\mu = 10$  is the optimal number of parents for our ES. However, it is a bit of a mixed-bag when it comes to deciding which fitness function is better than the other. An average over all the runs for a  $\mu = 10$  ES using both F0 and F1 fitness functions is shown in figure 5.23. The data indicates there is a slight advantage using our original function, F0 (that is, equation 5.0.3) which had slightly lower generations to target fitness.

This concludes our study of an alternative fitness function and the results indicate that while we understand the fitness function can have a large impact, the benefit of EA's in general is that even a fitness function with broad detail can be sufficient to find good solutions.

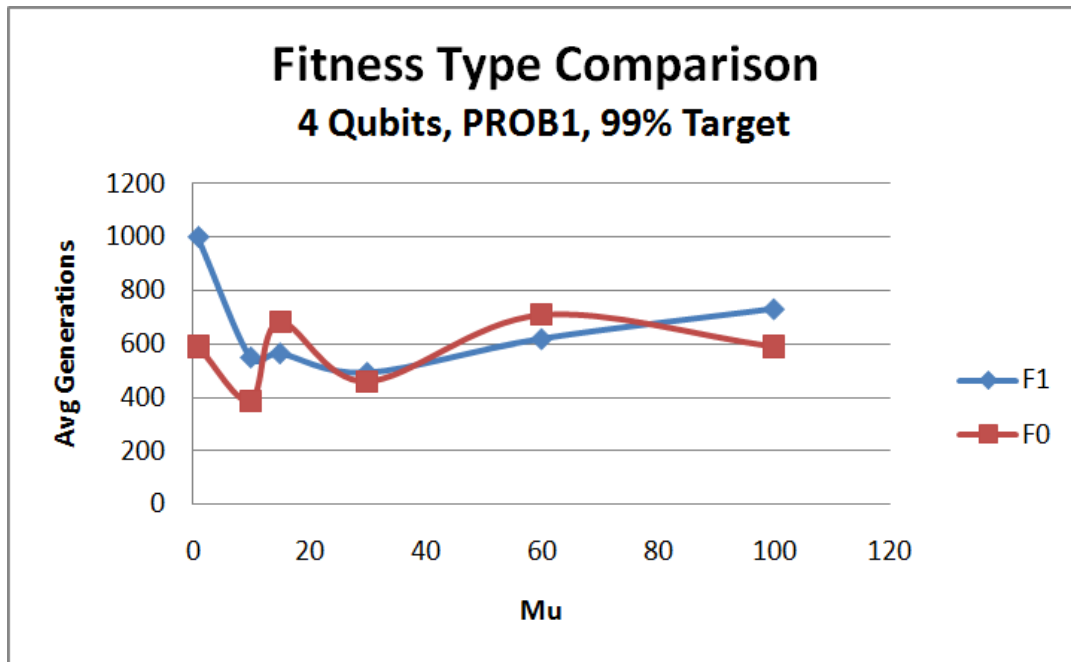


Figure 5.21: Fitness comparison for 4 Qubit, Probabilistic, 99% Target ES

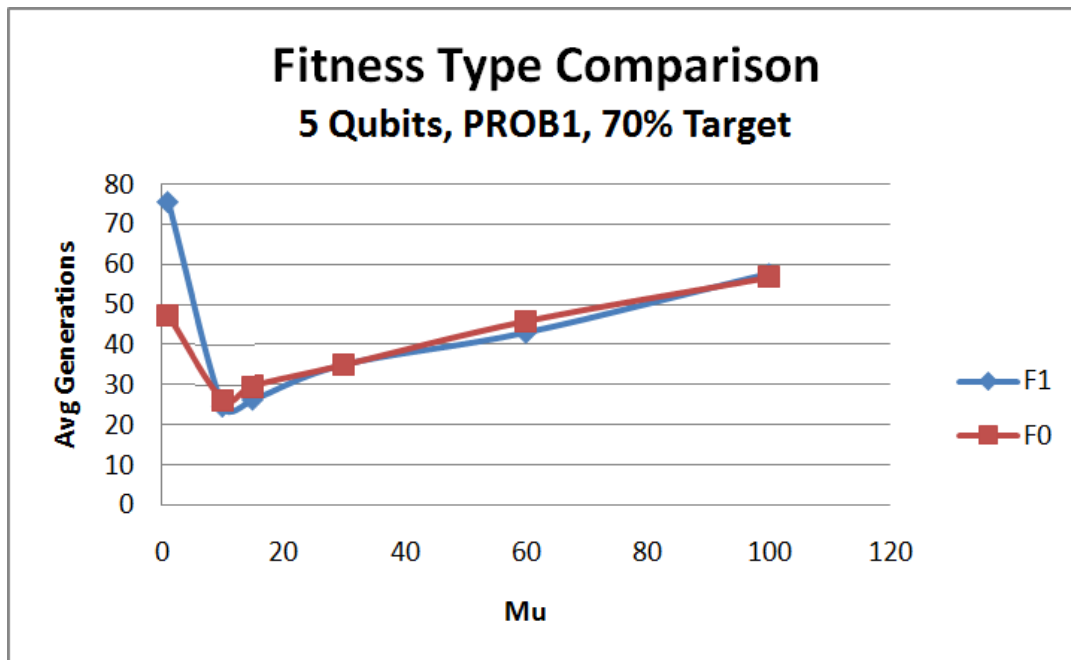


Figure 5.22: Fitness comparison for 5 Qubit, Probabilistic, 70% Target ES

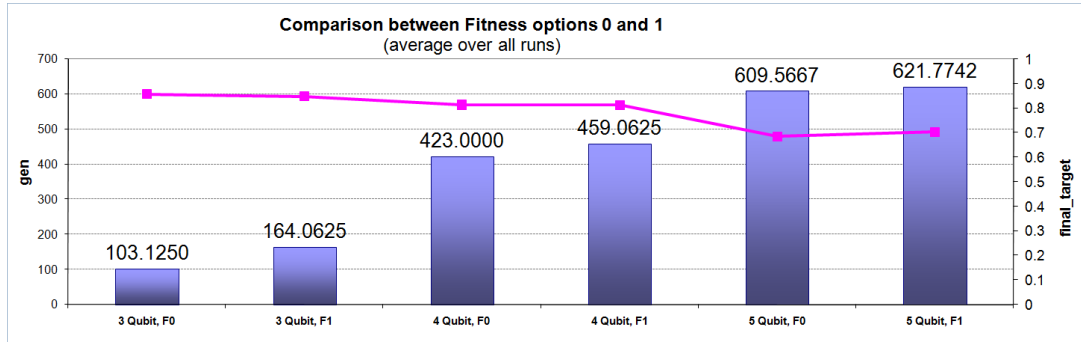


Figure 5.23: Fitness comparison for 5 Qubit, Probabilistic, 70% Target ES

### 5.2.6 Training Behavior

In this study we take a sample from what we deem a very good configuration and examine its training behavior — that is — when the operator is evolving, how does its fitness vary over each generation? This will give us a feel if we converge slowly and steadily or very rapidly in the beginning and taper off in later generations.

For this study we take a  $(15 + 500)$  ES and examine its training behavior using our standard input and target setup — that is — the input vector is the superposition state and the output target is state-0 ( $|00 \dots 0\rangle$ ). We've selected our original fitness function in equation 5.0.3 and use probabilistic target. We examine the behavior for 2 to 5 qubits under a relaxed 70% target as well as the more stringent 99% target. We run the ES for 5 independent runs and graph their behavior. We clarify that what is being tracked for the target is not the result of the fitness function, but rather the percent probability that we are hitting our target state. This is done by getting the probability distribution from the evolving operator using the superposition input and then examining the first state,  $|c_0|^2$ , which is targeting to be 100% if our target vector is  $|00 \dots 0\rangle$ .

We'll start with our less constrained study where we use a 70% target prob-

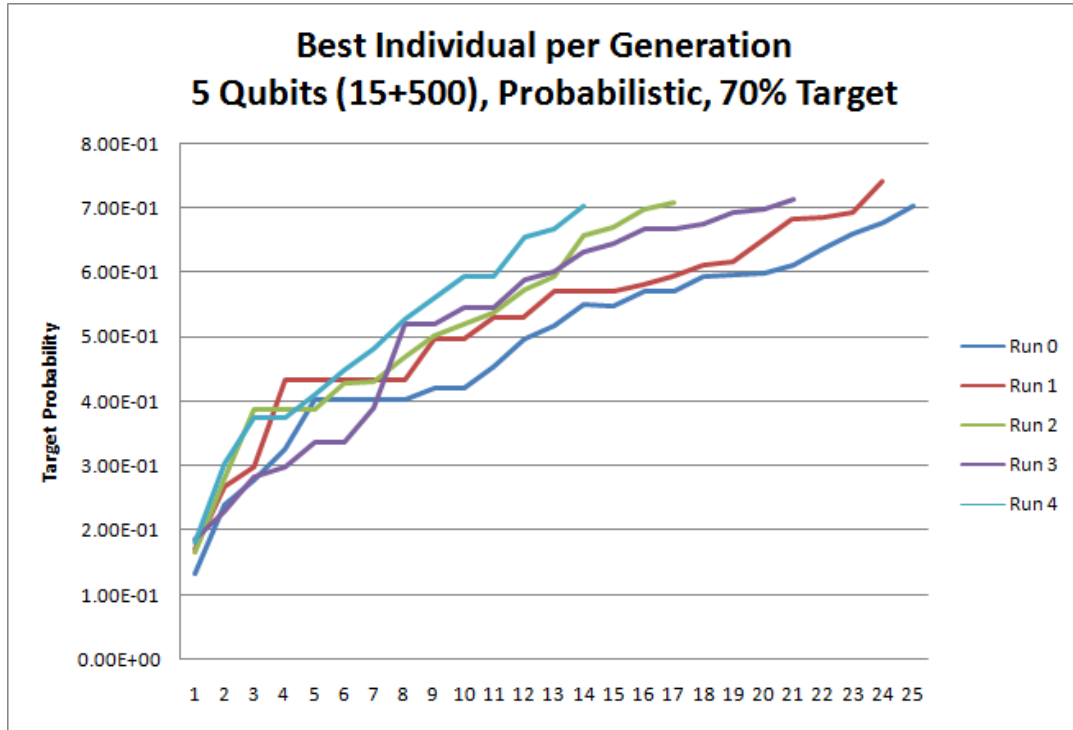


Figure 5.24: Best Individual per Generation, 5 Qubits, 70% Target

ability and reference the graphs in figures 5.24 to 5.27. What further loosens the constraints is that we are using a probabilistic target vector. The first figure is showing the 5 qubit behavior and we see that we have steady convergence and within 30 generations we hit a solution with  $> 70\%$  target probability. Remarkably, the 2, 3, and 4 qubit graphs also have steady convergence behavior, albeit short.

We now contrast the 70% target figures to the 99% target graphs shown in figures 5.28 to 5.31. The 5 qubit runs clearly are not able to converge after 1000 generations (which was our maximum). There is tremendous initial convergence within the first 50 generations to within the 85% to 90% range. However, we quickly flatten out and never peak above 92%. The 4 qubit case is very similar by having two of the runs achieve their target convergence within 50 generations.

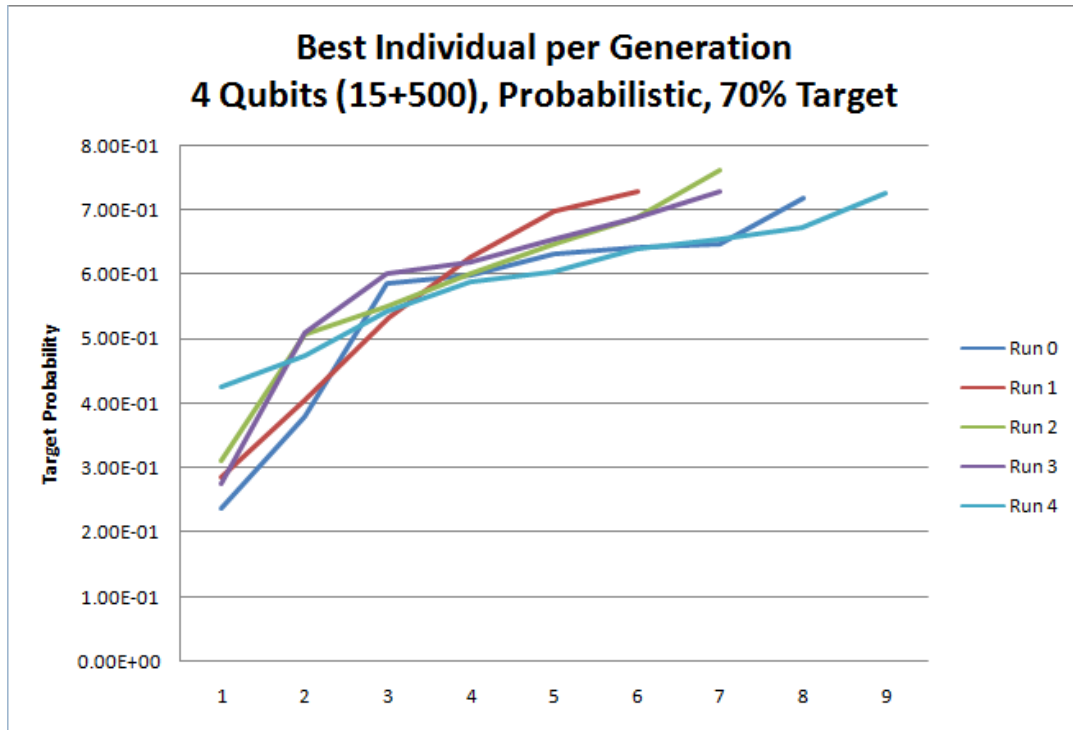


Figure 5.25: Best Individual per Generation, 4 Qubits, 70% Target

However, the two runs that don't quite make the 99% continue to stay flat through the 1000 generations.

The comparison is quite evident that considerable computational power is needed to achieve the extra resolution. Even the considerably relatively easier lower-qubit runs have 3 to 4 times the convergence time in terms of generations. However, in quantum computing, it could very well be that 70% or some number significantly lower than 100% is sufficient, and we see that we may become quickly reliant on pushing on the target probability as low as acceptably possible in order to achieve faster convergence on problems that will require many more qubits to be useful.

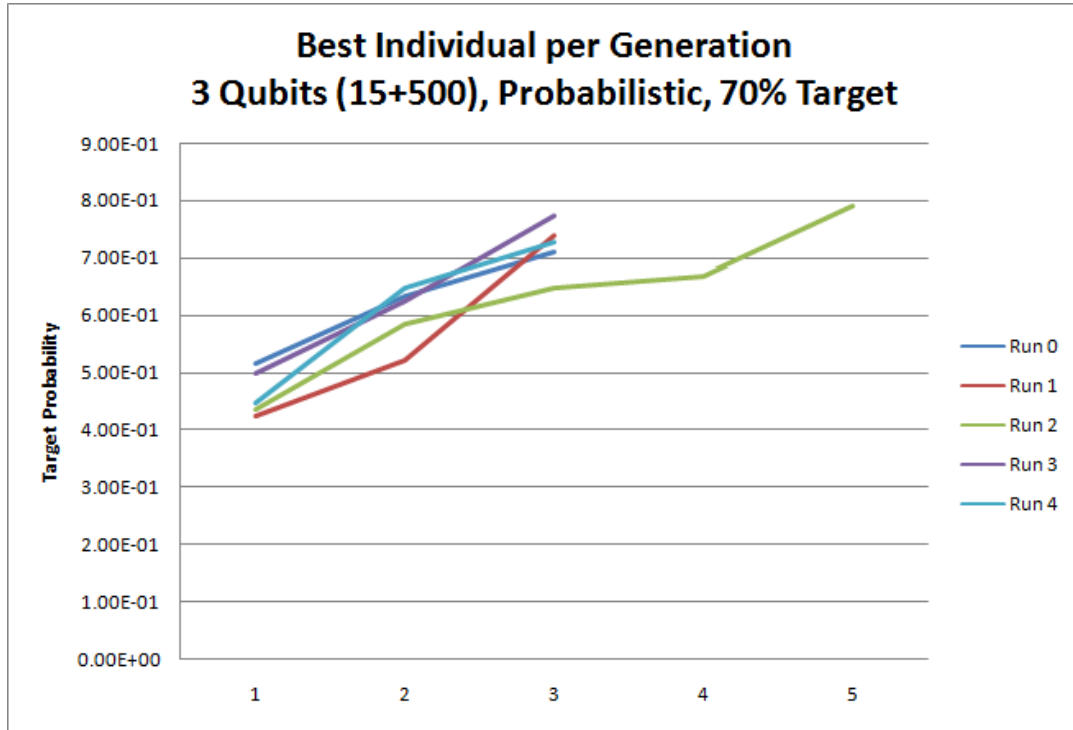


Figure 5.26: Best Individual per Generation, 3 Qubits, 70% Target

### 5.2.7 Best Convergence Results

In this section we'll highlight the best convergence data discovered through the ten-of-thousands of simulations we conducted and rolled up into the best of the best results graph shown in figure 5.32.

The exponential behavior of the problem is evident. No solution was found for 5 qubits at a 99% target fitness. However, the fact that we were able to evolve 5 qubit solutions at 70% within 12 generations is simply staggering! To scale beyond 5 qubits, however, is difficult without severe optimizations or simplification to our random unitary matrix technique to both limit the number of matrix multiplications and the number of angle parameters.

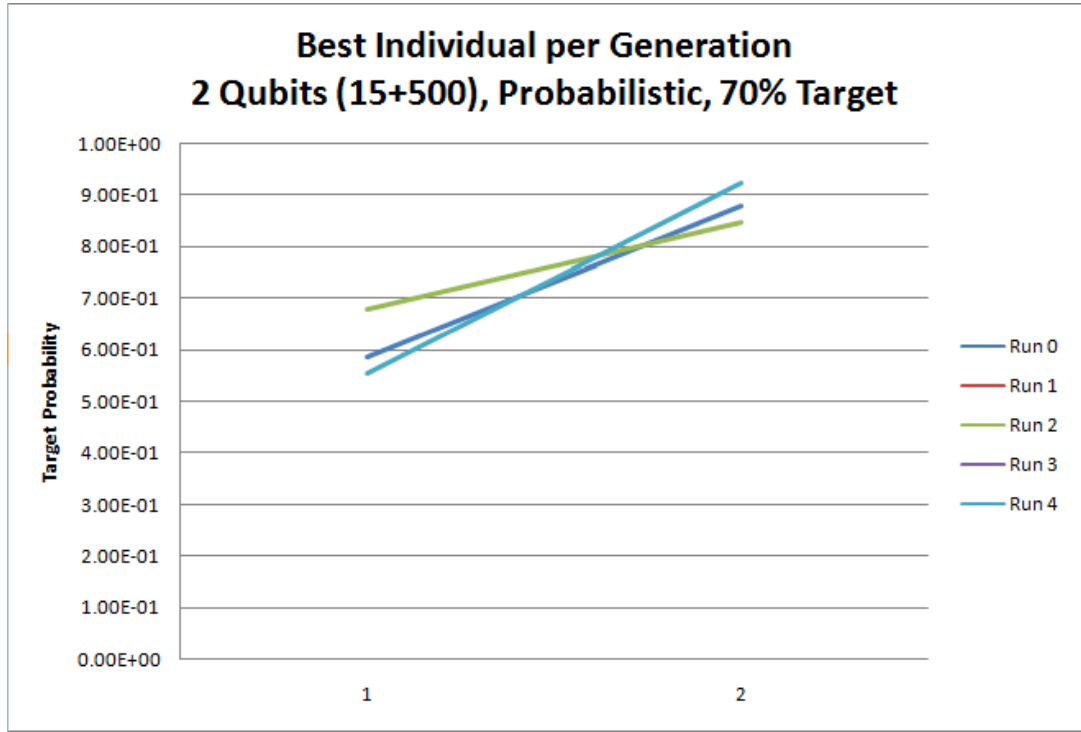


Figure 5.27: Best Individual per Generation, 2 Qubits, 70% Target

### 5.2.8 Simulation Run-Time

The majority of the simulations were launched using Linux netbatch to a pool of Dual and Quad-Core Xeon DP Servers with CPU frequencies ranging from 2.67GHz to 3.6GHz. The system memory ranged from 4GB to 16GB. Hyper-threading was enabled on all processors. A summary of the simulation run-times is presented in table 5.1 for only simulations with  $\mu = 10$  and  $\lambda = 500$ .

The memory requirements are quite small and statically allocated memory for unitary matrices is used to increase matrix operations. In our ES, an individual used 385KB of memory and so a population of 510 individuals consumed 191MB.

The run-time for higher qubits is obviously concerning. In the five qubit case, the worst run-time is 128,961 seconds which lasted for 1000 generations.

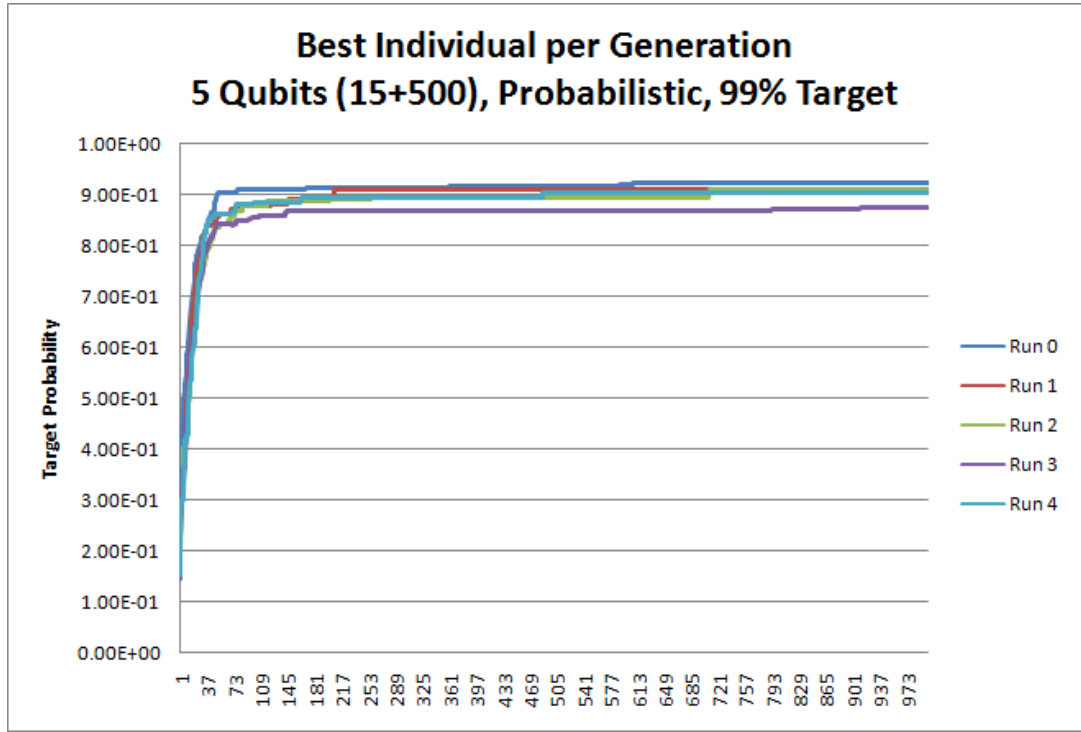


Figure 5.28: Best Individual per Generation, 5 Qubits, 99% Target

This means each generation took around 129 seconds. The total population is 510 (10 parents and 500 offspring). Therefore, to process a single individual took around a quarter of a second. That seems like a lot and the bulk of the time is consumed in matrix multiplications. At 5 qubits the unitary operator of an individual is a  $32 \times 32$  matrix, which is 1K entries, each represented by a double-precision tuple for the real and imaginary components. The method of creating a random unitary matrix as described in section 3.2 requires  $O(2^{2n})$  angle parameters and  $O(\frac{2^n(2^n+1)}{2})$  matrix multiplications to produce the final unitary matrix. At 5-qubits, that's 528  $32 \times 32$  complex double-precision matrix calculations and 1024 angle parameters (plus another 1024 strategy parameters) for the ES to mutate and/or recombine.

Figure 5.33 plots the average, minimum, and maximum run-times for various



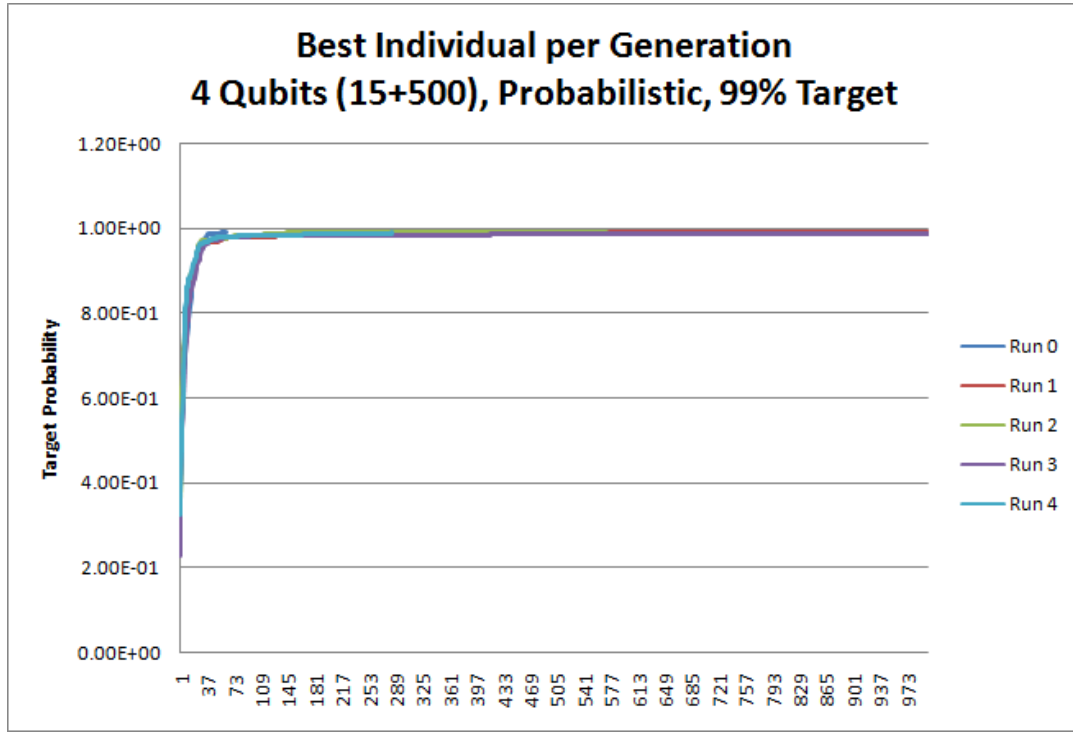


Figure 5.29: Best Individual per Generation, 4 Qubits, 99% Target

qubit, target fitness, and target mode configurations. The characteristic we pull from these graphs is that — understandably — the *average* cases are closer to the *maximum* curves indicating that our best cases — the *minimum* curves — are indeed harder to find over a collection of runs. Also of interest is the obvious difference in convergence behavior between 70% and 99% target fitness which intuitively makes sense as valid solutions are much more difficult to find at this target. This characteristic collapses at 4 and 5 qubits where simulations started to hit the 1000 generation limit. The fact that the end point plot on the 99% for the 5 qubit has the same min, max, and average value is because 5 qubit plots failed to converge at 99% target fitness.

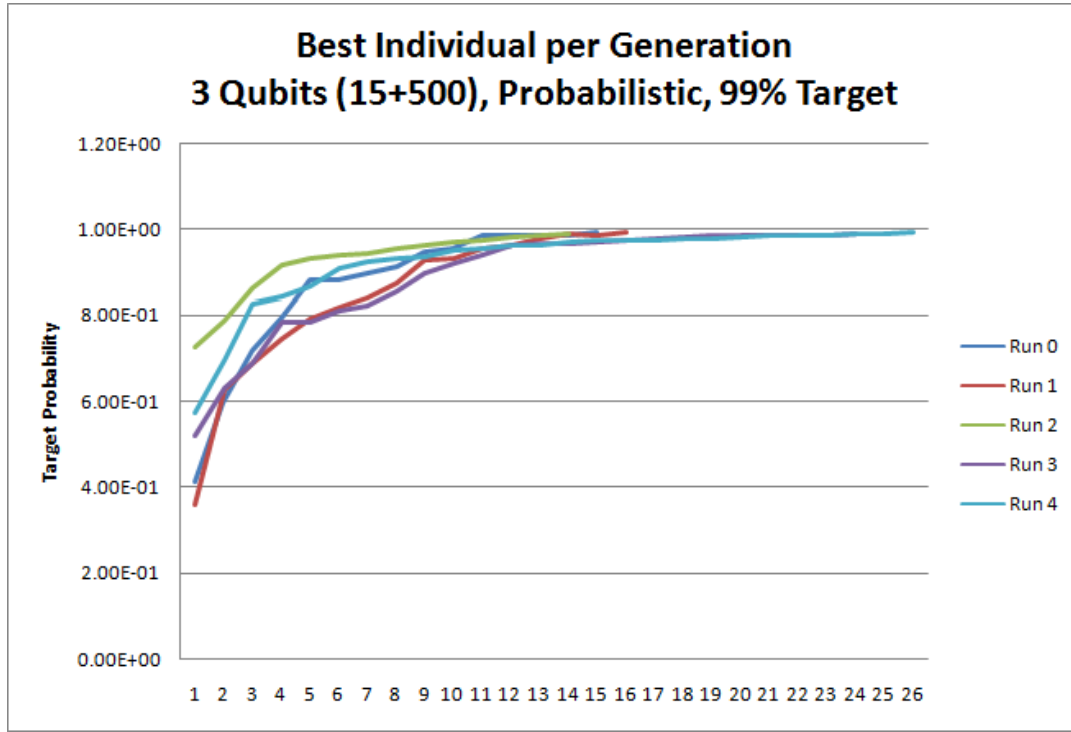


Figure 5.30: Best Individual per Generation, 3 Qubits, 99% Target

### 5.3 Evolving Known Quantum Gates

We put our ES to the test by first attempting to evolve well known single qubit operators such as the inverter ( $X$  gate), then two-input gates like CNOT, then attempt three-input gates like the Toffoli and Fredkin.

We quickly run into a limitation of our chosen random unitary matrix scheme described in section 3.2. Let's revisit the symbolic equation for a 2 input operator:

$$U = \begin{pmatrix} \cos(\phi_{1,2})e^{i\psi_{1,2}} & \sin(\phi_{1,2})e^{i\chi_{1,2}} \\ -\sin(\phi_{1,2})e^{-i\chi_{1,2}} & \cos(\phi_{1,2})e^{-i\psi_{1,2}} \end{pmatrix} \quad (5.3.1)$$

If we disregard the  $e^{i\dots}$  terms, then we have an anti-symmetric matrix (that is,  $A = -A^T$ ). This limits what types of gates we can evolve. With the

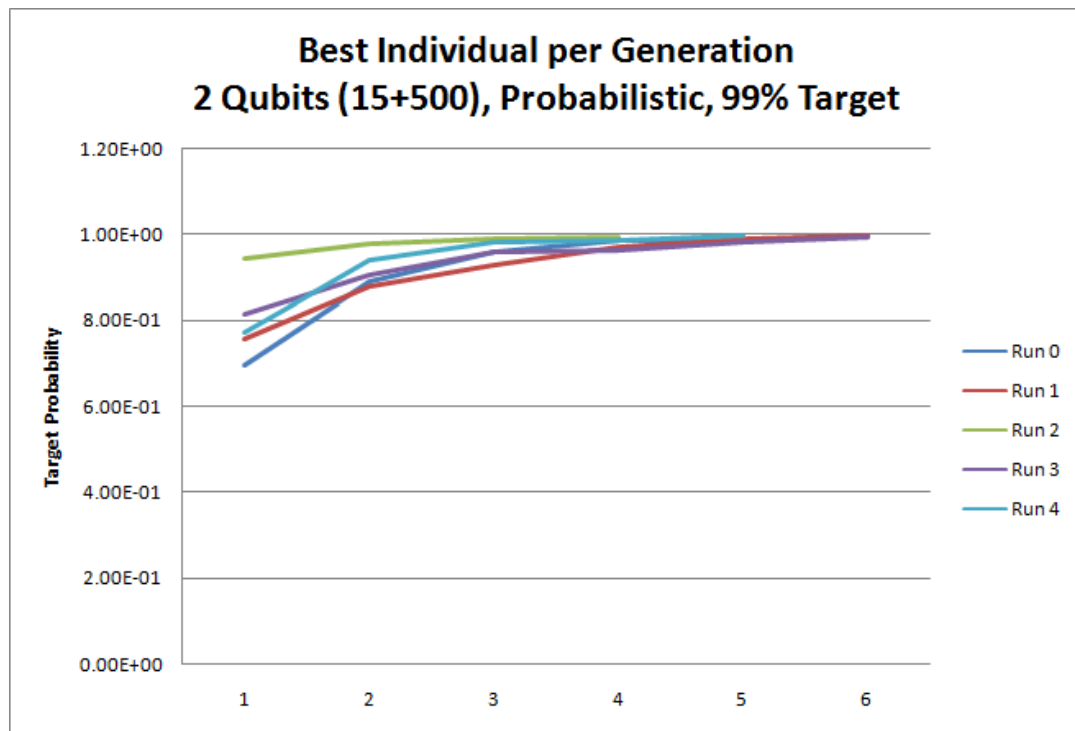


Figure 5.31: Best Individual per Generation, 2 Qubits, 99% Target

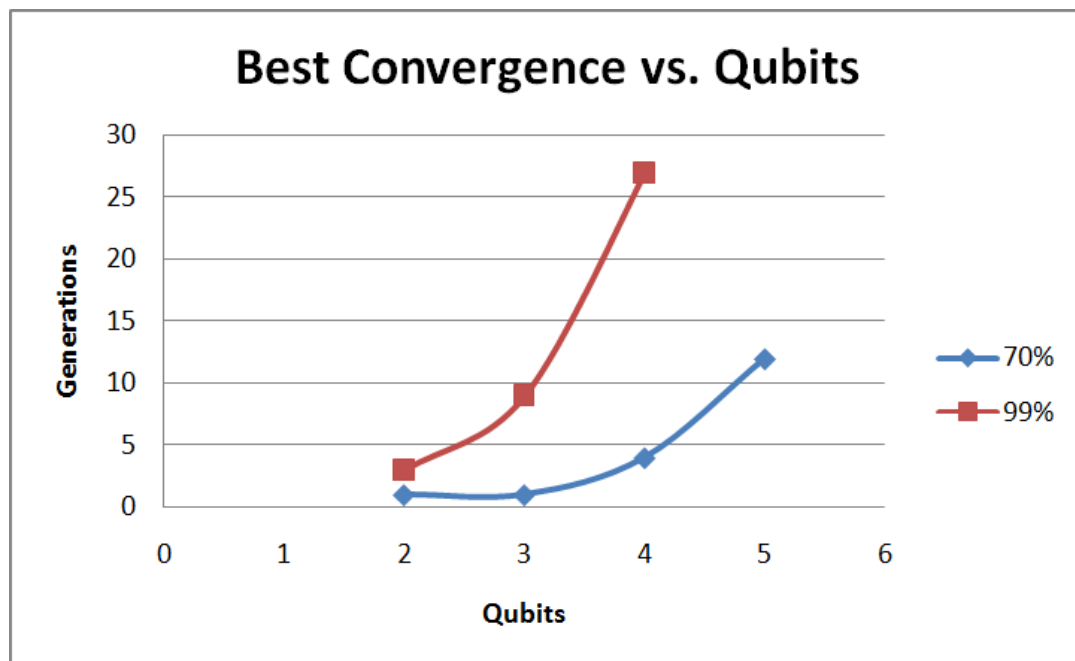


Figure 5.32: Best Fitness Convergence for 70% and 99% Targets

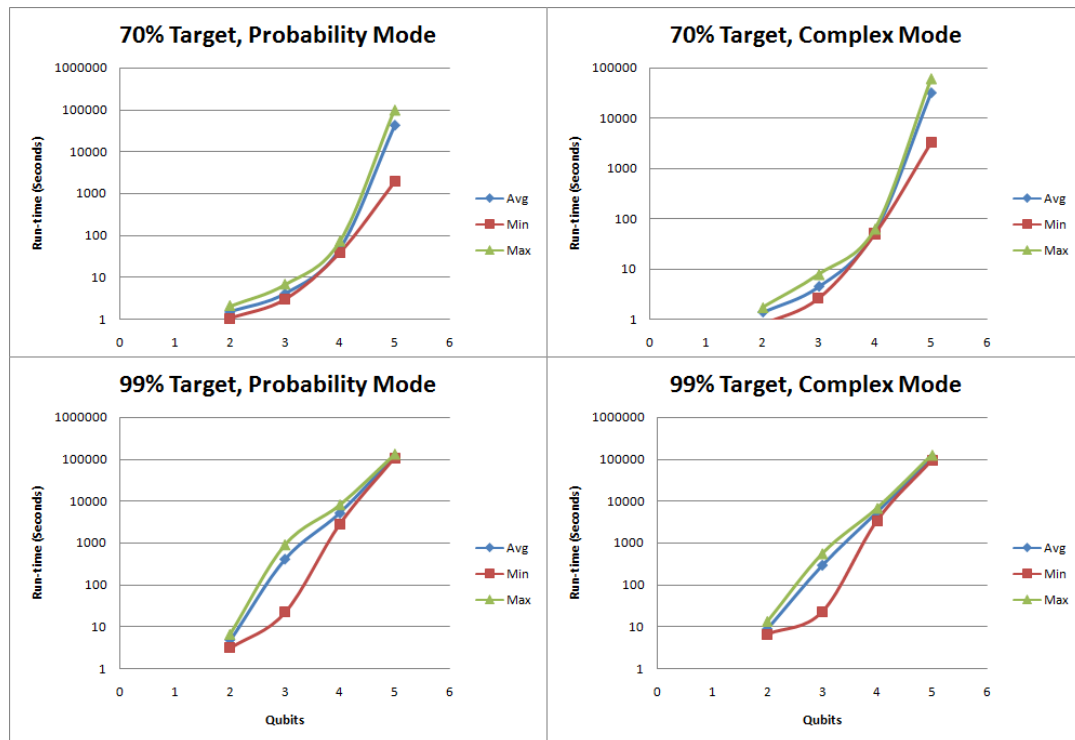


Figure 5.33: Average, Min, and Max Run-Times

	99% Target, Probability Mode		
Qubits	Avg	Min	Max
2	4.77265	3.166	6.6222
3	407.2967	22.1838	898.1082
4	5156.569	2820.702	8029.626
5	118722.187	105576.956	128961.87
	99% Target, Complex Mode		
2	8.9774	6.5688	13.7118
3	295.958	22.792	564.8346
4	5552.3015	3447.532	6839.278
5	109389.199	94558.166	126879.862
	70% Target, Probability Mode		
Qubits	Avg	Min	Max
2	1.52195	1.0402	2.054
3	4.127	2.947	6.6964
4	48.2972	38.7018	72.3034
5	42497.74275	1928.152	100072.068
	70% Target, Complex Mode		
2	1.36265	0.7848	1.7294
3	4.4207	2.6048	7.8712
4	54.19695	49.488	62.6838
5	31170.49975	3257.319	60331.878

Table 5.1: Average, Min, and Max Run-Times (in seconds) for  $\mu = 10$ ,  $\lambda = 500$  ES Simulations

complex terms and higher order matrices we are able to circumvent some of these limitations. Usually, we'll evolve equivalent matrices in terms of having the same probability (or post-measurement) results.

Unless otherwise specified, we used a  $(100 + 100)$  ES using 100% mutation and pre-measurement (complex) target vectors.

### 5.3.1 Evolving the Hadamard Gate

The goal here is to see if an ES can evolve the well known and extremely useful Hadamard quantum gate whose unitary matrix is shown below:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \quad (5.3.2)$$

This gate is extremely useful because it maps the basis states  $|0\rangle$  and  $|1\rangle$  into a superposition of the two states with equal weight. That is,

$$|0\rangle \rightarrow (|0\rangle + |1\rangle)/\sqrt{2}$$

and

$$|1\rangle \rightarrow (|0\rangle - |1\rangle)/\sqrt{2}$$

To set this up for an ES, we need to use two initial states (input states) that map to two final states (output states). The fitness is then averaged over both inputs. Pre-measurement based targets are used since the primary interest is in obtaining complex behavior. The ES produced the following input and target vectors for the single qubit gate problem:

```
INPUT[0]=[[1.0+0.0i],[0.0+0.0i]]
```

```
TARGET[0]=[[0.707+0.0i],[0.707+0.0i]]
```

```
INPUT[1]=[[0.0+0.0i],[1.0+0.0i]]
```

```
TARGET[1]=[[0.707+0.0i],[-0.707+0.0i]]
```

After 100 generations, the most fit individual has the unitary operator:

$$U = \begin{pmatrix} 0.707 & 0.707 \\ -0.707 & 0.707 \end{pmatrix} \approx \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \quad (5.3.3)$$

This result differs from the unitary matrix of the Hadamard gate described in equation (5.3.2). However, the probability distribution of this gate given each input state vector behaves exactly the same as the Hadamard gate. That is,

$$|U|0\rangle|^2 = |H|0\rangle|^2 = \begin{pmatrix} 50\% \\ 50\% \end{pmatrix}$$

and

$$|U|1\rangle|^2 = |H|1\rangle|^2 = \begin{pmatrix} 50\% \\ 50\% \end{pmatrix}$$

Our random unitary matrix method is incapable of producing the standard Hadamard gate. Nevertheless, the gate we evolved is still valid and turns out to be a known gate called the “pseudo-Hadamard” which actually is preferred when implementing quantum algorithms on Nuclear Magnetic Resonance quantum computers[72]. This results shows an ES can evolve alternate gates that might prove useful.

### 5.3.2 Evolving Known Single Qubit Gates

Single qubit gates use two input/output pairs when evolving individuals. Our input vectors are the basis states  $|0\rangle$  and  $|1\rangle$ . The two target output vectors are found in the ES by applying the target operator onto each input vector. During fitness evaluation, both input vectors are applied and the both output state

vectors are compared to the target states (pre-measurement). The single-qubit case is where we see more severe limitation from our random unitary matrix method.

### **The Identity Gate, I**

The matrix is the identity

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

The input/output mappings are defined as

INPUT[0]=[1.0+0.0i], [0.0+0.0i]]

TARGET[0]=[1.0+0.0i], [0.0+0.0i]]

INPUT[1]=[0.0+0.0i], [1.0+0.0i]]

TARGET[1]=[0.0+0.0i], [1.0+0.0i]]

It is trivial to evolve this gate.

### **The Inverter, X Gate**

The  $X$  gate is defined by the matrix

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

The input/output mappings are defined as

INPUT[0]=[1.0+0.0i], [0.0+0.0i]]

TARGET[0]=[0.0+0.0i], [1.0+0.0i]]

INPUT[1]=[0.0+0.0i], [1.0+0.0i]]



TARGET[1]=[[1.0+0.0i],[0.0+0.0i]]

Within 3 generations the ES converged on the following solution which had an 83% fitness

$$U = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

It is noted that this is not the same as the  $X$  matrix, but does have the same post-measurement effect of swapping the input and output state.

### The Y Gate

The  $Y$  gate is defined by the matrix

$$I = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

The input/output mappings are defined as

INPUT[0]=[[1.0+0.0i],[0.0+0.0i]]

TARGET[0]=[[0.0+0.0i],[0.0+1.0i]]

INPUT[1]=[[0.0+0.0i],[1.0+0.0i]]

TARGET[1]=[[0.0-1.0i],[0.0+0.0i]]

After 20 generations the fitness function peaked at 83% with the following best solution

$$U = \begin{pmatrix} 0 & -i \\ -i & 0 \end{pmatrix}$$

This is not the same as  $Y$  but the probability distribution is the same.

## The Z Gate

The  $Z$  gate is defined by the matrix

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

The input/output mappings are defined as

INPUT[0]=[1.0+0.0i], [0.0+0.0i]]

TARGET[0]=[1.0+0.0i], [0.0+0.0i]]

INPUT[1]=[0.0+0.0i], [1.0+0.0i]]

TARGET[1]=[0.0+0.0i], [-1.0+0.0i]]

After 2 generations the fitness function peaked at 83% with the following best solution

$$U = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

This is actually the  $I$  gate and not the  $Z$  gate. Due to the limitations of the random unitary matrix, the ES converged on this result which has the same probabilistic behavior but not the same pre-measurement behavior.

## The Phase Gate, S

The  $S$  gate is defined by the matrix

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

The input/output mappings are defined as

```

INPUT[0]=[1.0+0.0i],[0.0+0.0i]
TARGET[0]=[1.0+0.0i],[0.0+0.0i]
INPUT[1]=[0.0+0.0i],[1.0+0.0i]
TARGET[1]=[0.0+0.0i],[0.0+1.0i]

```

After 2 generations the fitness function peaked at 85% with the following best solution

$$U = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Like the  $Z$  gate, the ES converged on the  $I$  gate instead.

### 5.3.3 2-qubit Oracle for Deutsch's Problem

Deutsch presented a problem[23] for a quantum computer to determine whether a given function,  $f(x)$ , was *balanced* (meaning  $f(x)$  is 1 for half of the inputs and 0 for the other half) or *constant* (meaning  $f(x)$  is always 0 or always 1). This example is important since for  $n$  bits, a conventional computer requires on average  $2^n/2 + 1$  queries, whereas on a quantum computer Deutsch's algorithm exploits quantum parallelism and interference to produce the answer in one query.

The role of a quantum oracle is to make a decision based on its input state in one step. If the inputs of the quantum oracle are in superposition state, then the oracle can make all possible decisions in one step. Figure 5.34 shows how the quantum circuit is implemented. The input state,  $|\psi_0\rangle$ , is set to  $|01\rangle$ . This is then passed through Hadamard gates to put them into a superposition state which is then processed by  $U$ , the oracle we want to design. The output of the oracle will be a superposition of decisions which we can resolve by passing

them again through Hadamard gates. This has become a common structure of quantum algorithms. The objective here is to construct an oracle which will determine if  $f(x) = \text{NOT}(x)$  is balanced or constant.

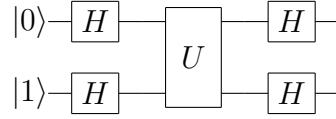


Figure 5.34: General quantum circuit for Deutsch's problem (2 qubits). The blocks labeled with an  $H$  are Hadamard gates while the block labeled with  $U$  is the unitary operator representing the oracle to be evolved.

Obviously  $f(x)$  is a *balanced* function and the desired pre-measurement output state is known to be  $|\psi\rangle = [0, 0, 0, -1]$ . The ES is modified such that it evolves the quantum oracle,  $U$ , until it satisfies the equation below. Note that the two Hadamards in parallel can be represented by the tensor product  $H \otimes H$ , also written as  $H^{\otimes 2}$ .

$$T = H^{\otimes 2} \times U \times H^{\otimes 2} \times (|0\rangle \otimes |1\rangle) \quad (5.3.4)$$

Using the (100 + 100) ES, a perfect solution was found after only 26 generations ( $\sim 5$  seconds runtime). To speed up the convergence, we modified the mutation scheme to increment the rotation angles ( $\phi, \psi, \chi$ ) in steps of size  $\pi/12$ . The resultant oracle was:

$$U = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad (5.3.5)$$

This is actually different from the oracle discovered by Deutsch and could

possibly have desirable properties over the original.

#### 5.4 Single Problem Instances

With a single problem instance we are given a single known final target solution state we'll call  $|\psi_t\rangle$ . The input state does not encode any parameters so we leave it in the superposition state  $((H^{\otimes n})(|0\rangle^{\otimes n}))$ . We want to evolve the quantum operator  $U$  such that it evolves the initial superposition state into the final target state within some acceptable error. That is,

$$U(H^{\otimes n})(|0\rangle^{\otimes n}) = |\psi_t\rangle \quad (5.4.1)$$

While this is the same input condition we had previously been using, we move away from the given target vector of  $|00\dots 0\rangle$  and instead look at real instances of the *Independent Set Problem*(ISP). This is a good choice for our investigation because it is an optimization problem that is known to be NP-hard [73].

The Independent Set Problem is defined as follows:

**Problem instance:** A graph  $G = (V, E)$  where  $V = \{1, 2, \dots, n\}$  is the set of vertices and  $E \subseteq V \times V$  the set of edges. An edge between vertices  $i, j$  is denoted by the pair  $(i, j) \in E$ .

**Feasible solution:** A set  $V'$  of nodes such that  $\forall i, j \in V' : (i, j) \notin E$ .  $V'$  is called an *independent set*.

**Optimal Solution:** Maximal  $|V'|$ —i.e., the max cardinality of the independent set.

Given an  $l$ -node graph, the easiest way to encode the solution of the ISP is let each qubit represent one of the  $l$  nodes. Therefore, our  $l$ -node ISP solution

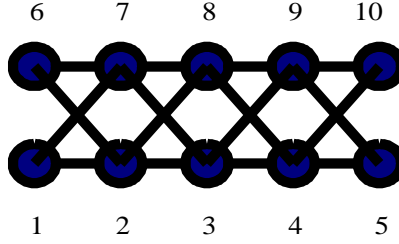


Figure 5.35: An Independent Set Problem instance. Both  $V'_1 = \{2, 4, 7, 9\}$  and  $V'_2 = \{1, 6, 9\}$  are independent sets, but neither one is globally optimum.

can be encoded with  $l$  qubits. Thus, a quantum computer state  $|\psi\rangle$  would be a linear superposition of all  $2^l$  base states, and  $|\psi\rangle$  would be represented as a column matrix of complex amplitudes.

Given the final state,  $|\psi_f\rangle$ , we examine the probability distribution ( $||\psi_f\rangle|^2$ ). The  $i$ 'th element of the probability distribution indicates the probability of that particular quantum state representing the solution to that instance of the ISP.

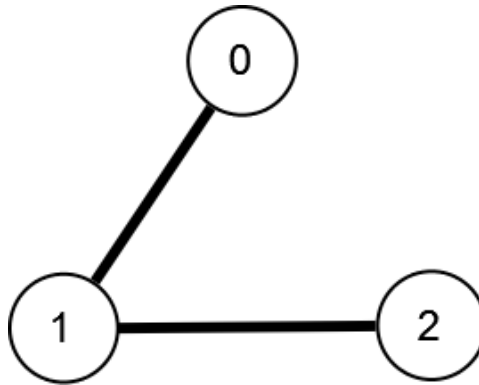


Figure 5.36: An ISP instance of three nodes.

Let's assume we have the graph shown in figure 5.36. The solution to the ISP is set of nodes  $V = \{0, 2\}$ . The output state vector of our quantum computer would have the form:  $|q_2q_1q_0\rangle$ . Where  $q_i$  represents whether the  $i$ 'th vertex is part of the solution set. Since we know the optimal solution, we denote our

target vectors as:

$$|\psi_t\rangle = \left( \begin{array}{c|c} q_2 q_1 q_0 & |c_i|^2 \\ \hline 000 & 0 \\ 001 & 0 \\ 010 & 0 \\ 011 & 0 \\ 100 & 0 \\ 101 & 1 \\ 110 & 0 \\ 111 & 0 \end{array} \right) \quad (5.4.2)$$

Where  $c_i$  is the complex amplitude of the  $i$ 'th element in the final target state,  $ket\psi_t$ . Our target solution is where  $q_0 = 1$  and  $q_2 = 1$  representing the set of vertices  $\{0, 2\}$  and therefore produces the target state  $|101\rangle$ . If we treat the state index as a binary number then the complex amplitude associated with this state is  $c_{101b}$  (note the binary subscript) which is  $c_5$  and the mod-squared value is unity.

Things become a little more interesting when we look at cases where we have more than one optimal solution. This happens when we have two vertex sets,  $V_1$  and  $V_2$ , that both have the same maximum cardinality. In these cases we have a few choice of how we want to evolve our quantum operator,  $U$ . The one we immediately disregard is the ability to have enough qubits to all single and multiple combinations. The reason we disregard this condition is because in quantum computing qubits are expensive. Rather, we examine two potential solutions — the first being that we only have *Single Solution* states. If we

had a graph with more than one solution set, we choose *one* of them for our target state. This results in a target state vector with a single element set to unity. Alternatively, we can have *Multiple Solution* states. That is, if we have  $k$  solutions, then  $k$  of the states in the final target state are set to  $1/\sqrt{k}$  such that – ideally – we have an equal chance of our quantum computer measuring one of our  $k$  solutions.

We will now examine two particular instances of the ISP. The first is a 3-node graph and the second is a 4-node graph. Both graphs have multiple optimal solutions and we'll look at what happens if we evolve *Single Solution* target states versus *Multiple Solution* target states.

#### 5.4.1 3-Node ISP Instance

The 3-node ISP instance we'll examine is shown in figure 5.37 which has two optimal solutions:  $V_1 = \{0, 1\}$  and  $V_2 = \{0, 2\}$ .

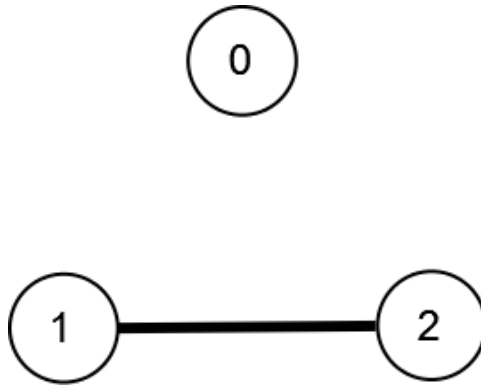


Figure 5.37: 3-Node ISP with multiple solutions.

There are three combinations of solutions. We can evolve a  $U$  such that:

1. the target state has one solution,  $V_1$ .
2. the target state has one solution,  $V_2$ .



3. the target state has equal probability of either  $V_1$  or  $V_2$ .

We use our learnings from section 5.2 to derive the ES configuration we use for this experiment:

- Qubits = 3
- $\mu = 15$
- $\lambda = 500$
- $(\mu + \lambda)$
- Initial/Reset  $\sigma = 0.1$
- Probability Mode

We first look at the first combination where we evolve  $U$  to solve for the single solution  $V_1$  which includes vertices 0 and 1. We'll call this target state  $|\psi_{V_1}\rangle = |011\rangle$ .

The target state vector has probability distribution:

$$||\psi_{V_1}\rangle|^2 = \left( \begin{array}{c|c} q_2q_1q_0 & |c_i|^2 \\ \hline 000 & 0 \\ 001 & 0 \\ 010 & 0 \\ 011 & 1 \\ 100 & 0 \\ 101 & 0 \\ 110 & 0 \\ 111 & 0 \end{array} \right) \quad (5.4.3)$$

After only 38 generations a near-perfect solution was found. The final probability distribution from the ES software was:

```
[4.9245738636457224e-07+0.0000000000000000e+00i]
[8.6433567854445739e-07+0.0000000000000000e+00i]
[1.0587107887294338e-07+0.0000000000000000e+00i]
[9.9999556456800931e-01+0.0000000000000000e+00i]
[4.7075657987933189e-07+0.0000000000000000e+00i]
[3.2862208420697364e-07+0.0000000000000000e+00i]
[9.1080953033475597e-08+0.0000000000000000e+00i]
[2.0823082297455682e-06+0.0000000000000000e+00i]
```

Which had > 99.999% probability of measuring the correct state,  $|011\rangle$ .

We now examine the second solution state which encodes the other solution,  $V_2$ , as the only single solution which we'll call  $|psi_{V_2}\rangle = |101\rangle$  and has the target

probability distribution:

$$||\psi_{V_2}\rangle|^2 = \left( \begin{array}{c|c} q_2 q_1 q_0 & |c_i|^2 \\ \hline 000 & 0 \\ 001 & 0 \\ 010 & 0 \\ 011 & 0 \\ 100 & 0 \\ 101 & 1 \\ 110 & 0 \\ 111 & 0 \end{array} \right) \quad (5.4.4)$$

The near-perfect solution was found in only 23 generations with the final probability solution:

```
[2.9988900142786454e-08+0.0000000000000000e+00i]
[1.4283612477751486e-07+0.0000000000000000e+00i]
[9.2164150627821011e-08+0.0000000000000000e+00i]
[4.0677032198546089e-08+0.0000000000000000e+00i]
[4.4980079544324004e-08+0.0000000000000000e+00i]
[9.9999954844533079e-01+0.0000000000000000e+00i]
[3.8315555592278912e-08+0.0000000000000000e+00i]
[6.2592826005085335e-08+0.0000000000000000e+00i]
```

Which measures the ideal solution with probability > 99.9999%.

Now we examine our third method which encodes both solution  $V_1$  and  $V_2$  (we'll refer to this later as the  $V_1 + V_2$  solution) by having an equal probability

of measuring both. The target solution state is  $|\psi_{V_1, V_2}\rangle = 1/\sqrt{2}(|\psi_{V_1}\rangle + |\psi_{V_2}\rangle)$  and will have the following probability distribution:

$$||\psi_{V_1, V_2}\rangle|^2 = \begin{pmatrix} q_2 q_1 q_0 & |c_i|^2 \\ \hline 000 & 0 \\ 001 & 0 \\ 010 & 0 \\ 011 & 0.5 \\ 100 & 0 \\ 101 & 0.5 \\ 110 & 0 \\ 111 & 0 \end{pmatrix} \quad (5.4.5)$$

The near-perfect solution was found in only 28 generations and had the following probability distribution:

```
[4.9192593319585385e-06+0.0000000000000000e+00i]
[1.7989070108534328e-05+0.0000000000000000e+00i]
[1.9999769563384656e-05+0.0000000000000000e+00i]
[4.9997865578452011e-01+0.0000000000000000e+00i]
[1.7357865943957853e-05+0.0000000000000000e+00i]
[4.9993061808914407e-01+0.0000000000000000e+00i]
[9.2067390232041892e-06+0.0000000000000000e+00i]
[2.1253422365012175e-05+0.0000000000000000e+00i]
```

Which measures  $V_1$  with  $> 49.99\%$  accuracy and  $V_2$  with also  $> 49.99\%$  accuracy.

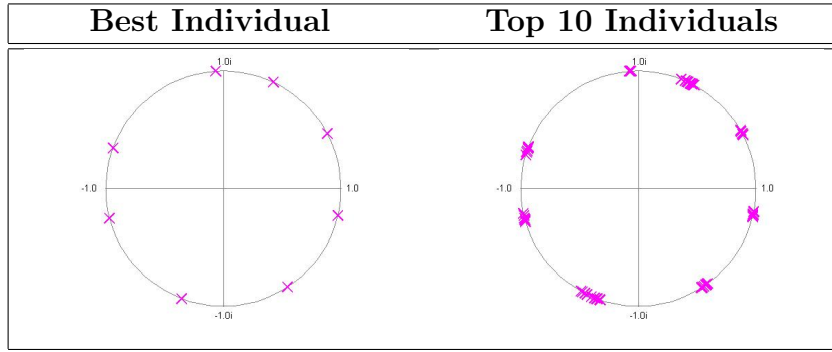


Figure 5.38: Eigenvalue placement for 3-Node ISP with single solution  $V_1$ .

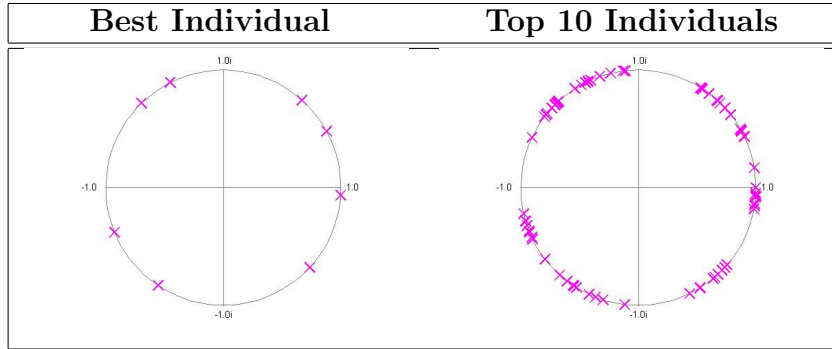


Figure 5.39: Eigenvalue placement for 3-Node ISP with single solution  $V_2$ .

### 3-Node ISP Eigenvalue Placement

We have two solutions to our ISP and we've evolved quantum operators to solve for either solution and for both with equal probability. Now we are curious how the eigenvalue placement of these operators compare as well as the eigenvalue placement comparison between the top ten candidates.

Since 3-node ISP instances are represented with only 3 qubits, we only have  $2^3 = 8$  eigenvalues to plot. As we've done previously, we'll show how the best solution placement compared against the top ten individuals combined.

We also compare how the eigenvalue placements for the  $V_1$  and  $V_2$  compare when super-imposed and placed next to the  $V_1 + V_2$  solution placement as shown

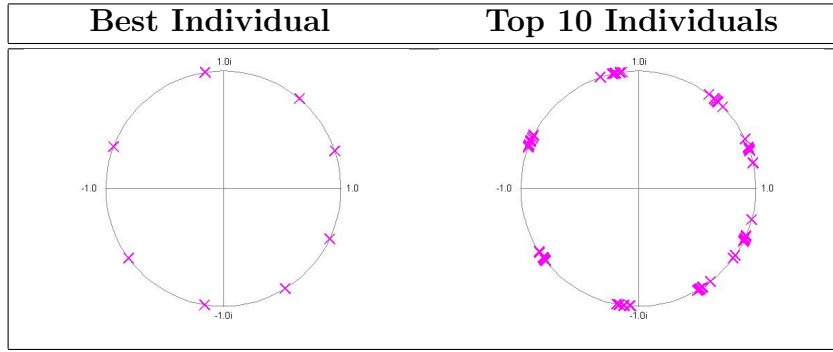


Figure 5.40: Eigenvalue placement for 3-Node ISP with both solutions  $V_1$  and  $V_2$ .

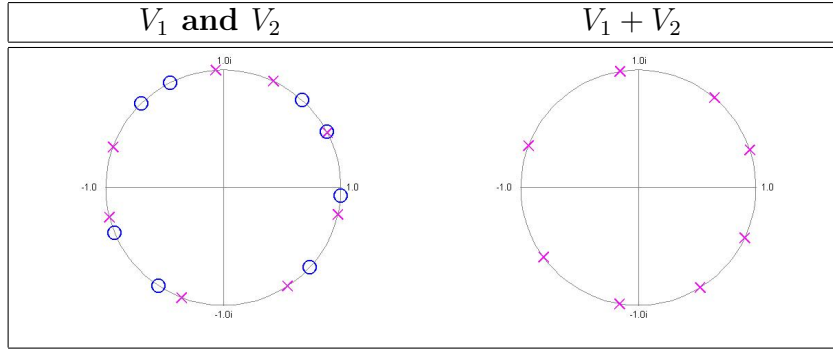


Figure 5.41: Comparison of 3-Node ISP  $V_1$  (X's) superimposed with  $V_2$  (O's) against  $V_1 + V_2$  eigenvalue plots.

in figure 5.41.

While it is interesting that the  $V_1$  top individuals showed more clumping than the  $V_2$  candidates, it really doesn't tell us much. As we saw in our earlier studies, the eigenvalue placement is well distributed on the unit circle with no clear tendency toward degeneracy. Again, this is likely an artifact of our random unitary matrix generation method. When comparing the super-imposed  $V_1$  and  $V_2$  eigenvalue plots with the multi-solution  $V_1 + V_2$  plot, there's not a clear conclusion that can be made. One might be inclined to think that an average of the  $V_1$  and  $V_2$  eigenvalues might be similar to the  $V_1 + V_2$ . This is true in the

upper-right quadrant of the plot, but the other quadrants seem to be similar to the  $V_2$  eigenvalues rather than  $V_1$ .

#### 5.4.2 4-Node ISP Instance

The 4-node ISP instance we'll examine is shown in figure 5.42 which has three optimal solutions:  $V_1 = \{0, 3\}$ ,  $V_2 = \{1, 2\}$ , and  $V_3 = \{1, 3\}$ .

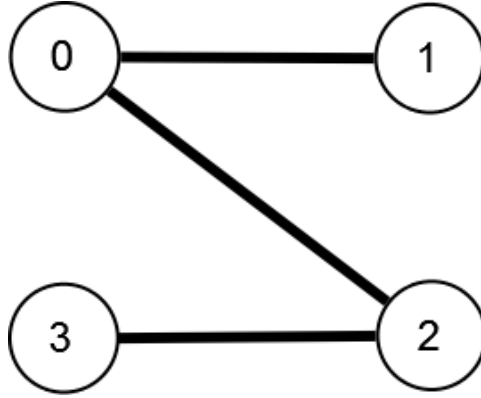


Figure 5.42: 4-Node ISP with multiple solutions.

There are seven combinations of solutions. We can evolve a  $U$  such that:

1. the target state has one solution,  $V_1$ .
2. the target state has one solution,  $V_2$ .
3. the target state has one solution,  $V_3$ .
4. the target state has equal probability of either  $V_1$  or  $V_2$ .
5. the target state has equal probability of either  $V_1$  or  $V_3$ .
6. the target state has equal probability of either  $V_2$  or  $V_3$ .
7. the target state has equal probability of either  $V_1$  or  $V_2$  or  $V_3$ .

We'll use the same ES parameters as in the 3-node ISP case except with 4 qubits.

Because we have 4 qubits, our probability distribution vector has 16 elements. For now, we'll simplify by using ket-notation instead. Also, rather than showing the final distribution for all 16 elements from the ES, we'll only show those we are interested in (the solution states!).

$V_1$ : In the case we want to evolve a  $U$  which solves for only solution  $V_1$ , our target state is  $|1001\rangle$ . The ES was able to find a best individual with  $> 99.9999\%$  fitness within 328 generations. The probability of measuring the target state ( $|1001\rangle$ ) was  $> 99.64\%$ .

$V_2$ : In the case we want to evolve a  $U$  which solves for only solution  $V_2$ , our target state is  $|0110\rangle$ . The ES was able to find a best individual with  $> 99.9999\%$  fitness within 759 generations. The probability of measuring the target state ( $|1001\rangle$ ) was  $> 99.57\%$ .

$V_3$ : In the case we want to evolve a  $U$  which solves for only solution  $V_3$ , our target state is  $|1010\rangle$ . The ES was able to find a best individual with  $> 99.9999\%$  fitness within 864 generations. The probability of measuring the target state ( $|1001\rangle$ ) was  $> 99.63\%$ .

$V_1 + V_2$ : In the case we want to evolve a  $U$  which solves for only solutions  $V_1$  and  $V_2$ , our target state is  $1/\sqrt{2}(|1001\rangle + |0110\rangle)$ . The ES was able to find a best individual with  $> 99.9999\%$  fitness within 278 generations. The probability of measuring the  $V_1$  solution was  $> 49.72\%$ . The probability of measuring the  $V_2$  solution was  $> 49.79\%$ .

$V_1 + V_3$ : In the case we want to evolve a  $U$  which solves for only solutions  $V_1$  and  $V_3$ , our target state is  $1/\sqrt{2}(|1001\rangle + |1010\rangle)$ . The ES was able to find a best



<b>Solutions</b>	<b>Generations</b>	$> P(V_1)$	$> P(V_2)$	$> P(V_3)$
$V_1$	328	99.64%		
$V_2$	759		99.57%	
$V_3$	864			99.63%
$V_1 + V_2$	278	49.72%	49.79%	
$V_1 + V_3$	676	49.73%		49.70%
$V_2 + V_3$	170		49.75%	49.79%
$V_1 + V_2 + V_3$	1000	32.72%	33.19%	32.49%

Table 5.2: Summary Table of 4-Node ISP Single Instance Training Results.

individual with  $> 99.9999\%$  fitness within 676 generations. The probability of measuring the  $V_1$  solution was  $> 49.73\%$ . The probability of measuring the  $V_3$  solution was  $> 49.70\%$ .

$V_2 + V_3$ : In the case we want to evolve a  $U$  which solves for only solutions  $V_2$  and  $V_3$ , our target state is  $1/\sqrt{2}(|0110\rangle + |1010\rangle)$ . The ES was able to find a best individual with  $> 99.9999\%$  fitness within 170 generations. The probability of measuring the  $V_2$  solution was  $> 49.75\%$ . The probability of measuring the  $V_3$  solution was  $> 49.79\%$ .

$V_1 + V_2 + V_3$ : In the case we want to evolve a  $U$  which solves for all solutions  $V_1$ ,  $V_2$ , and  $V_3$ , our target state is  $1/\sqrt{3}(|1001\rangle + |0110\rangle + |1010\rangle)$ . The ES was only able to find a best individual with  $> 99.999\%$  fitness within the limit of 1000 generations. The probability of measuring the  $V_1$  solution was  $> 32.72\%$ . The probability of measuring the  $V_2$  solution was  $> 33.19\%$ . The probability of measuring the  $V_3$  solution was  $> 32.49\%$ .

We summarize the above data in table 5.2.

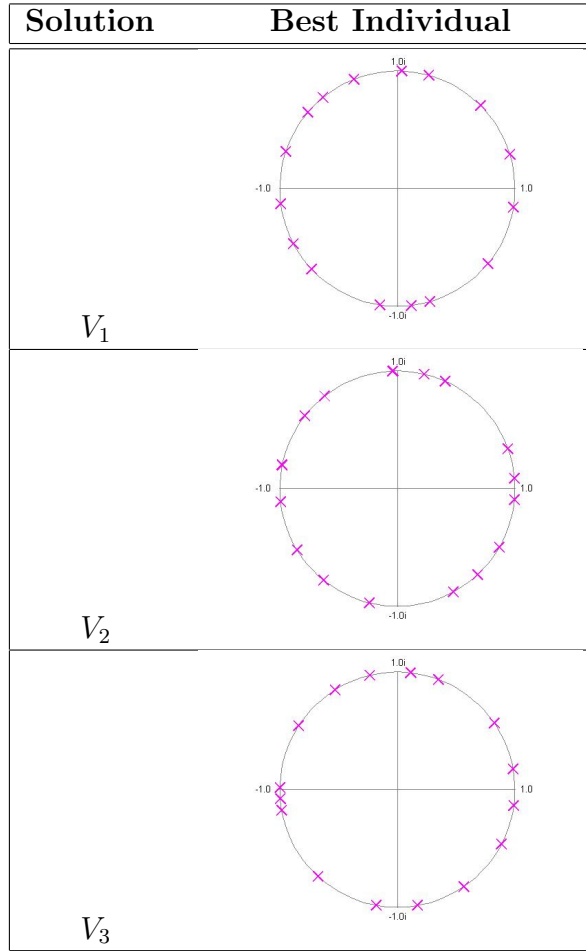


Figure 5.43: 4-Node ISP Single-Solution Instance Eigenvalue placements.

#### 4-Node ISP Eigenvalue Placement

Our 4-node ISP instances use 4 qubits and thus have  $2^4 = 16$  eigenvalues. For brevity, we show in figures 5.43 and 5.44 the eigenvalue placement for only the best individual.

It's also interesting to look at how the eigenvalue plots for the individual solutions ( $V_1$ ,  $V_2$ , and  $V_3$ ) when super-imposed on to one plot compare the multi-solution placement plot ( $V_1 + V_2 + V_3$ ). This is shown in figure 5.45.

As with the 3-node ISP instance problem, it is difficult to make any clear

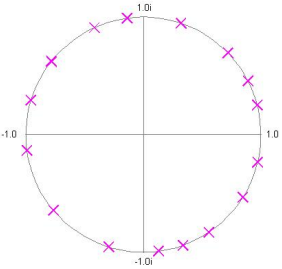
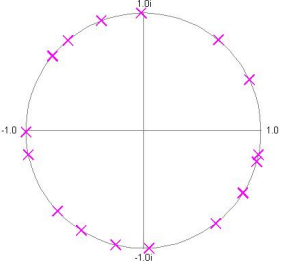
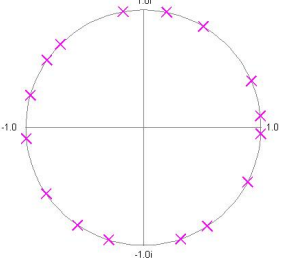
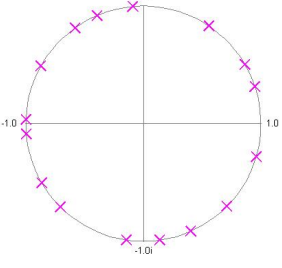
Solution	Best Individual
$V_1 + V_2$	
$V_1 + V_3$	
$V_2 + V_3$	
$V_1 + V_2 + V_3$	

Figure 5.44: 4-Node ISP Multi-Solution Instance Eigenvalue placements.

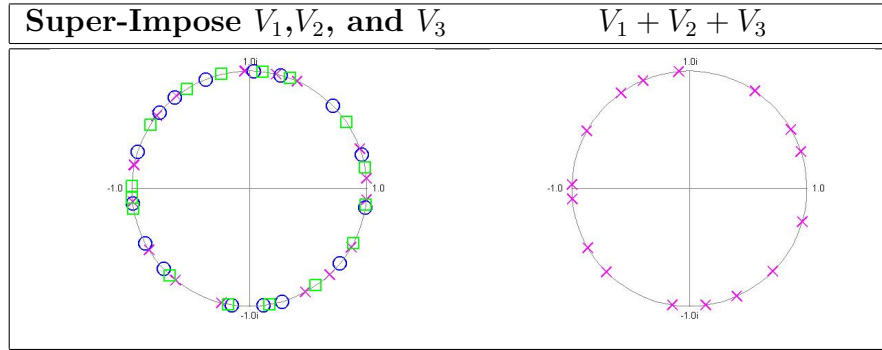


Figure 5.45: 4-Node ISP instance eigenvalue placement comparison.  $V_1$  (X's),  $V_2$  (O's),  $V_3$  (Squares)

conclusions from the eigenvalue placement. Eigenvalues are again well distributed with no cases of degeneracy. However, we definitely have cases where eigenvalues are close enough to be equal. For example, the  $V_3$  solution plot has three eigenvalue closely bunched to real axis near  $-1.0$ . Another example is in the  $V_1 + V_2$  multi-solution plot, there are two eigenvalues very close in value near the real axis at  $1.0$ . In the single solution plots, we notice that there is generally an even distribution about the four quadrants with each quadrant having 4 eigenvalue points. However, in the multi-solution plots, we see this is not true. In fact, the  $V_1 + V_3$  has only two eigenvalue points in the upper-right quadrant. In the comparison plot shown in figure 5.45, it is again not a simple conclusion although an average of the eigenvalues (not shown) wouldn't be a far estimation of the multi-solution plot. However, when we start have more eigenvalues as we increase the size of the problem, it becomes harder to discriminate the eigenvalues as being close or apart.

### 5.4.3 Single Instance Conclusions

Overall, our ES did an excellent job of evolving operators which were able to solve our 3-node and 4-node ISP instances with very high accuracy. Even in our most difficult 4-node case where we wanted to solve for any of the three solutions with equal probability, we achieved a probability within 1% of the ideal  $1/3$ . This gives us very high confidence that we can use our ES to design quantum operators for such problem instances for both *single* and *multiple* solution scenarios. Although the eigenvalue plots showed some interesting behavior and no degeneracy, it is unfortunately inconclusive. Because these are probability based targets, it could be that our ES has too much freedom finding valid candidates. Complex targets would be much more restrictive although for the types of problems, it's the desired measuring point. Rather, it would perhaps be more beneficial to restrict the ES to forming a unitary matrix with far less angle parameters.

## 5.5 General Problem Solving

In *General Problem Solving* we make a very subtle yet substantial change in how we evolve our quantum operators. Instead of a single initial state (we sometimes call this the input state) that is the equal superposition state, we allow the input state to encode the problem to be solved and allow multiple inputs and outputs. This impacts how we evolve our quantum operators and greatly increases the difficulty since we now have a number of input-output pairs to test rather than a single input-output pair. As with the single instance solutions we covered in section 5.4, we'll be focusing on the *Independent Set Problem* (ISP).

We first describe how we encode the input state to describe a graph for our

ISP. This will introduce us to some of the immediate constraints when forming our quantum circuit. The most fundamental problem being that the number of qubits required to encode all possible graphs increases at a much faster rate than the number of qubits required to encode the solution.

Next we'll describe the methodology used to generate the graphs and their ISP solutions. Just like we saw in single instance problems, there are cases where we have multiple best solutions so we examine how to deal with single-solution and multi-solution target states.

Since encoding general problem instances requires multiple input states which have an impact on the ES performance, we spend some time to look at what ES parameters are best for multiple input-output training.

The goal of this section is to use our ES to evolve a quantum operator which can find the general solution to the ISP for any graph. The input state would somehow encode our graph connections and the resultant state would represent the best ISP solution(s).

### 5.5.1 Encoding General Graphs and ISP Solutions

We've already explained in section 5.4 how we encode the output of the ISP. To re-iterate, for  $n$  nodes, we need  $n$  qubits assuming we encode multiple "best" solutions by splitting the probability amongst the two or more best solution states (we called this *multi-solution* encoding) or simply choosing to encode only one of the best solutions (we called this *single solution* encoding). Our first question now is how do we encode the graph and how many qubits are required?

The number of *number of connections* for given graph of  $n$  is given by the

simple equation:

$$C(n) = \frac{n(n-1)}{2} \quad (5.5.1)$$

Now, let's say we needed a certain number of qubits to encode all possible connections. The equation is simple to derive as we simply need a qubit which determines where a given connection is active or not. Since we have  $C(n)$  connections which can be in either one of 2 state, then it's a simple equation given by:

$$G(n) = 2^{C(n)} \quad (5.5.2)$$

We name the function  $G(n)$  to denote that it is encoding the number of possible “graphs”.

So, the two parameters we want to encode are our input which describes the graph and therefore needs to encode up to  $G(n)$  possibilities. Also, output which we already know must be at least  $n$  qubits. To determine the number of qubits necessary to encode the graph, we use the following equation:

$$\log_2(G(n)) = \log_2(2^{C(n)}) = C(n) \quad (5.5.3)$$

Therefore, we say the number of qubits needed is the maximum between the qubits needed to encode the solution and the qubits needed to encode the graph. Table 5.3 shows how the number of qubits required quickly becomes dominated by the number connections (which determines the number of graphs) rather than output solutions. To encode a 4-node general solution would require 6 qubits even though only 4 are needed for the solution.

For this section we limit our studies to 3 node graphs. This is the size we used in our single instance problems and is the smallest graph size of interest

Node $n$	Solutions $2^n$	Connections $n(n-1)/2$	Graphs $2^{(n(n-1)/2)}$	Qubits $\text{MAX}(n, n(n-1)/2)$
1	2	0	1	1
2	4	1	2	2
3	8	3	8	3
4	16	6	64	6
5	32	10	1024	10
6	64	15	32768	15
7	128	21	2097152	21
8	256	28	268435456	28
9	512	36	...	36
10	1024	45	...	45

Table 5.3: Number of qubits required to encode an  $n$  node graph for general ISP solutions.

for the ISP. It also turns out the 3 node graphs have a very special property — that is, it requires the same number of qubits to encode all possible graph combinations as to encode the possible ISP solutions... 3 qubits. As a note, 4-node graph were attempted, however, these required 6 qubits and run-time became an issue.

The method we use to encode the graph connections into the input state is relatively straightforward and uses the following algorithm for  $N$  qubits:

```

cnt = 0
for i=1:N-1
    for j=i+1:N
        q[cnt++] = 1 if node[i] connects to node[j]
    end
end
end

```

We visually show how this works for a 6 qubit graph encoding all possible 4-node graphs in figure 5.46.



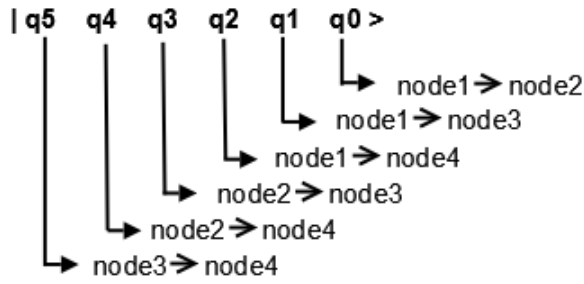


Figure 5.46: Encoding a 4-node graph for a general ISP.

Input State	Connections
$ 000\rangle$	No connections
$ 001\rangle$	$\text{node1} \rightarrow \text{node2}$
$ 010\rangle$	$\text{node1} \rightarrow \text{node3}$
$ 011\rangle$	$\text{node1} \rightarrow \text{node2}, \text{node1} \rightarrow \text{node3}$
$ 100\rangle$	$\text{node2} \rightarrow \text{node3}$
$ 101\rangle$	$\text{node1} \rightarrow \text{node2}, \text{node2} \rightarrow \text{node3}$
$ 110\rangle$	$\text{node1} \rightarrow \text{node3}, \text{node2} \rightarrow \text{node3}$
$ 111\rangle$	$\text{node1} \rightarrow \text{node2}, \text{node1} \rightarrow \text{node3}, \text{node2} \rightarrow \text{node3}$

Table 5.4: Input state encoding for 3-node ISP graphs.

The 3-node case which uses 3 qubits can be written out explicitly in terms of the input state and what connections it represents:

### 5.5.2 ES Input Methodology

Certain steps are taken to generate the input for the ES. We essentially need a list of input states and their associated target output states. To do this, we do the following:

1. Generate all possible graphs.
2. Solve the ISP for each graph, noting single and multi-solution versions.
3. Form the input-output training pairs for the ES.

We developed a greedy algorithm which solves the ISP for all graphs fed to it. One of the parameters is whether we should support single or multiple best solutions. In the single solution case, only one solution is chosen. In the multiple best solution case, we specify that there is an equal probability of selecting any of the best solutions.

As an example, let's look again at the 3-node case. According to table 5.3, it has 8 possible graphs and thus requires 3 qubits. We now show the input state and target state for the single and multi-solution generated from our greedy algorithm below:

Input State	Target State Single Solution	Target State Multi-solution
$ 000\rangle$	$ 111\rangle$	$ 111\rangle$
$ 001\rangle$	$ 101\rangle$	$1/\sqrt{2}( 101\rangle +  110\rangle)$
$ 010\rangle$	$ 011\rangle$	$1/\sqrt{2}( 011\rangle +  110\rangle)$
$ 011\rangle$	$ 110\rangle$	$ 110\rangle$
$ 100\rangle$	$ 011\rangle$	$1/\sqrt{2}( 101\rangle +  110\rangle)$
$ 101\rangle$	$ 101\rangle$	$ 101\rangle$
$ 110\rangle$	$ 011\rangle$	$ 011\rangle$
$ 111\rangle$	$ 000\rangle$	$ 000\rangle$

Table 5.5: 3-Node ISP graph encoding with single and multiple solution state encodings.

### 5.5.3 Tuning the ES Parameters for General Problems

We suspect that some of our best ES parameters for single instance problems need to be adjusted for multiple input-output training. In particular, we suspect the training step size,  $\sigma$  needs to be larger. However, we vary many of the basic parameters (such as  $\mu$  and  $\lambda$  to again convince ourselves we are specifying a

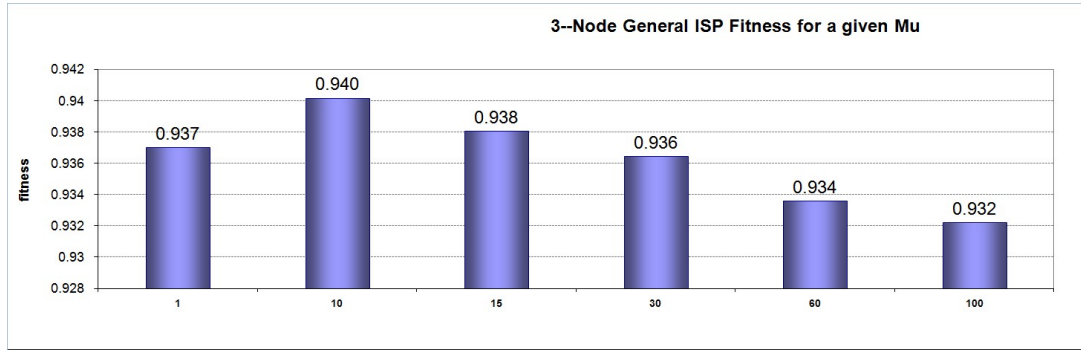


Figure 5.47: 3--Node General ISP Fitness given  $\mu$ .

near optimal set of parameters for our ES.

For our regressions we vary the following parameters on both 3--node single and multi--solution ISP configurations:

- $\mu$  is varied between 1, 10, 15, 30, 60, and 100.
- $\lambda$  is varied between 100, 200, 500, and 1000.
- We look at both  $(\mu + \lambda)$  and  $(\mu, \lambda)$ .
- Initial/Reset  $\sigma$  is varied from 0.1, 0.2, 0.5, and 0.7.
- We look at *Probabilistic* versus *Complex* targets.

We recall from our single instance ES runs that a  $(10 + 1000)$  or  $(15 + 1000)$  ES with Initial/Reset  $\sigma$  around 0.1, and probabilistic mode targets. We'll do a similar analysis by picking out key graphs to expose the best parameters.

### Number of Parents, $\mu$

To get an idea of the better number of parents (that is, the value of  $\mu$ ), we plot the average fitness of all simulations for each  $\mu$ . This is shown in figure 5.47.

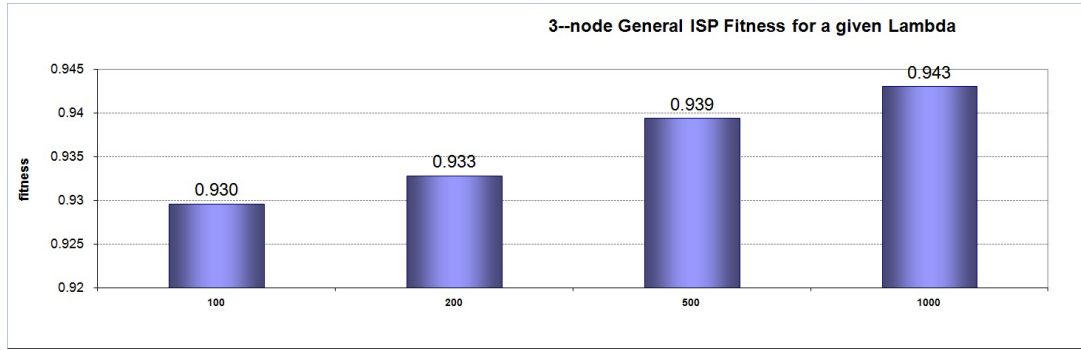


Figure 5.48: 3-Node General ISP Fitness given  $\lambda$ .

Figure 5.47 hints that – like our single instance cases – a smaller  $\mu$  like 10 or 15 is more desirable as, on average, the fitness was higher in these cases.

### Number of Children, $\lambda$

We do a similar analysis for the number of children by plotting the average fitness for each  $\lambda$ . The result is shown in figure 5.48. The result of which is exactly the same behavior as we saw before — the more children the better. So, larger values of  $\lambda$ , like 500 or 1000, are more desirable.

### Initial/Reset $\sigma$

Now that we have a good idea of what values of  $\mu$  and  $\lambda$ , we can average the fitness of those simulations and categorize by the initial/reset  $\sigma$  value. This graph is shown in figure 5.49.

Interestingly, this is different from the single instance cases which trained better with lower  $\sigma$ 's. Rather, for our general solutions, a higher reset/initial  $\sigma$  like 0.5 is desirable.

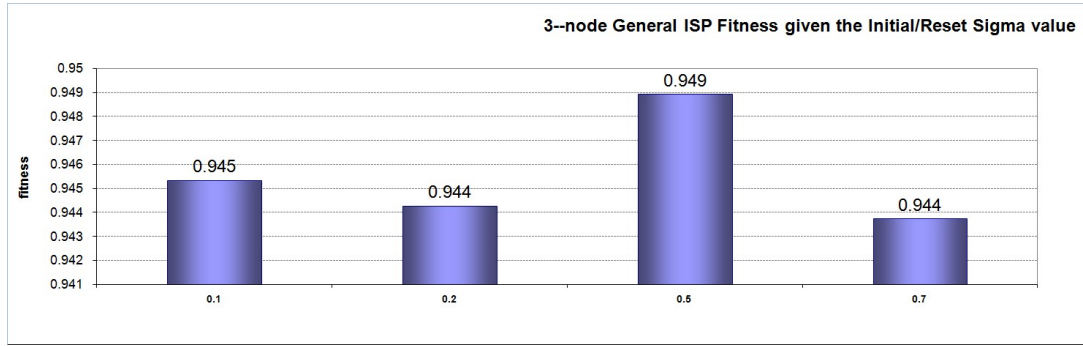


Figure 5.49: 3-Node General ISP Fitness given a reset/initial  $\sigma$ .

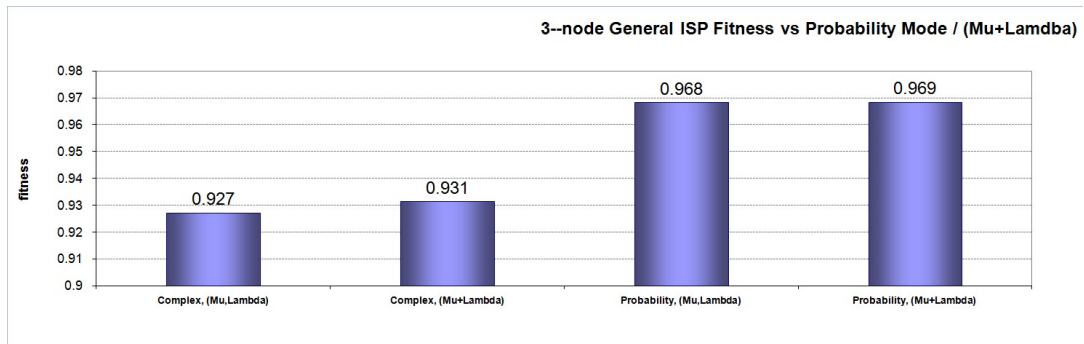


Figure 5.50: 3-Node General ISP Fitness for Probability/Complex and  $(\mu + \lambda)$  vs.  $(\mu, \lambda)$  mode.

### Probability/Complex Mode and $(\mu + \lambda)$ vs. $(\mu, \lambda)$

We continue to hone-in on our best parameters and look at how having a probabilistic versus a complex target as well as having a  $(\mu + \lambda)$  ES versus a  $(\mu, \lambda)$  ES influence our overall training fitness. The results are shown in figure 5.50. Not surprisingly, having a probabilistic target results in a less restrictive matrix and allows the ES to find better candidates. The impact of a  $(\mu + \lambda)$  versus a  $(\mu, \lambda)$  ES was very small and in the noise. There is a slight favor the  $(\mu + \lambda)$ .

Input Graph	Target Solution	Probability of Measuring Solution	Error
$ 000\rangle$	$ 111\rangle$	88.91%	11.09
$ 001\rangle$	$ 101\rangle$	49.60%	50.40
$ 010\rangle$	$ 011\rangle$	24.47%	75.53
$ 011\rangle$	$ 110\rangle$	90.39%	9.61
$ 100\rangle$	$ 011\rangle$	00.50%	99.5
$ 101\rangle$	$ 101\rangle$	50.37%	49.63
$ 110\rangle$	$ 011\rangle$	02.01%	97.99
$ 111\rangle$	$ 000\rangle$	48.38%	51.62

Table 5.6: Comparison of target solution for 3 Node ISP Single Solution.

### Parameter Selection

From the previous analysis, we conclude that for the 3 node generalized ISP, the ES parameters should be:

- (10, 1000) ES.
- Initial/Reset  $\sigma$  of 0.5.
- Probabilistic target.

#### 5.5.4 Test and Fitness for the Generalized 3 Node ISP

Using the parameters we determined, the ES was ran for 1000 generations for both the single and multiple best solution cases. We'll look at the probability distribution and compare it to the target distribution for each input graph condition. The best way to show the fitness is to compare only the probability of measuring the output states of interest.

Table 5.6 shows the target single solution for a given input encoded graph state and the actual probability of our evolved operator measuring the correct solution. For a couple of these, such as input states  $|000\rangle$  and  $|011\rangle$ , the results

Input Graph	Target Solution	Probability of Measuring Solution
$ 000\rangle$	$ 111\rangle$	89.94%
$ 001\rangle$	50% $ 101\rangle$ + 50% $ 110\rangle$	21.83% $ 101\rangle$ + 7.0% $ 110\rangle$
$ 010\rangle$	50% $ 011\rangle$ + 50% $ 110\rangle$	24.93% $ 011\rangle$ + 25.95% $ 110\rangle$
$ 011\rangle$	$ 110\rangle$	65.73%
$ 100\rangle$	50% $ 101\rangle$ + 50% $ 110\rangle$	18.39% $ 101\rangle$ + 0.01% $ 110\rangle$
$ 101\rangle$	$ 101\rangle$	57.39%
$ 110\rangle$	$ 011\rangle$	5.38%
$ 111\rangle$	$ 000\rangle$	3.41%

Table 5.7: Comparison of target solution for 3 Node ISP Single Solution.

are good at  $> 88\%$  and  $> 90\%$ , respectively. However, for other input states, the performance is definitely subpar with embarrassingly poor performance on input states  $|100\rangle$  and  $|110\rangle$  which completely missing the target. Overall, the performance is not very desirable, but we understand that training with multiple input–output pairs is a significantly more difficult task than instance problems.

We do a similar performance analysis for the multi–solution case which is shown in table 5.7. Unfortunately, the performance for multi–solution is also undesirable with only one input state ( $|000\rangle$ ) having its target solution being measured with a good probability at 89.94%. Again, we see cases of extremely poor performance such as input states  $|110\rangle$ ,  $|111\rangle$ , and we complete miss one of the solutions for input state  $|100\rangle$ .

The overall performance for the generalized problems using the ISP was lower than expected. For only 3 qubits, we should be able to achieve a much higher fitness. The limitation is likely from multiple aspects. For one — as we have previously mentioned — the increased complexity from having 8 input–output states makes finding better fit individuals very difficult. Also, it is possible that our random unitary matrix method creates strong dependencies that make training multiple input–output pairs difficult. It seems odd that some output

states are measured at close to 0% while others are around 90%. There is likely some push and pulling happening.

Better parameter selection is probably not the solution. We need to go back to the random unitary matrix and architect one that could potentially have fewer dependencies although it is not straight-forward how this could be done. However, having far less angle parameters would be a good start. We address this concept in our future work in section 9.



## Chapter 6

### Eigenanalysis

The eigenanalysis of a quantum operator involves gathering the eigenvalues and eigenvectors from one or more related unitary operators and determining if there are any key characteristics. In particular, we are looking for revealing information that would aid us in designing more accurate quantum operators and/or enhancing existing operators to yield better measurements.

We start with examining the eigenvalue placement of our previously generated unitary operators. This reviews how we plot the eigenvalues and their properties. We're interested how various parameters of the ES altered the behaviors of the eigenvalues. For example, we would expect that a family of operators evolved using probabilistic targets would have less restrictions and possible result in a more varied eigenvalue plots.

We then do a deep dive into what the eigenvalues mean. For example, had I been given the eigenvalues and eigenvectors for a NOT gate, it would be difficult at first glance to tell had we not known it was a NOT gate. We develop a methodology to relate the eigenvalues and eigenvectors back to a well outer-product format<sup>1</sup> which typically uses the standard computational basis

---

<sup>1</sup>Unfortunately, the outer-product method is not well defined in the literature but is widely used.

vectors we've reviewed before.

Once we have the eigenvalues, can we adjust them. For example, if we see certain eigenvalues wanting to be close, we might assume they are tending toward the same value – thus introducing degeneracy. In section 6.4, we look at how eigenvalues can be adjusted, how we can re-construct the new unitary operator, and their implications.

## 6.1 Eigenvalue Placement

As part of our journey, we are very interested how the eigenvalues for evolved solutions land on the unit circle. That is, if we evolve a family of unitary operators which solve the same problem but under varied conditions, how would their eigenvalue placement vary?

One might expect to see complex conjugate pairs, however, and as we will see in our eigenvalue placement plots, we don't always see complex conjugates. The definition below helps us to understand why this is the case:

**Definition 19** *Let  $A$  be a unitary operator and  $|b\rangle$  be an eigenket (eigenvector or sometimes called an eigenfunction) with (complex) eigenvalue  $b$ . Then  $b^*$  is the eigenvalue for  $A^t$  (the transpose of  $A$ ) with the same eigenket. Put another way, if  $b$  is an eigenvalue of  $A$ , then  $b^*$  is an eigenvalue of  $A^t$  and not  $A$ . In addition,  $b$  and  $b^*$  will have the same eigenket.*

For this study, we ran numerous simulations using the setup conditions mentioned in equations 5.0.1 and 5.0.2. For the target vector, we allow both *Complex* (that is, pre-measurement) targets and *Probabilistic* (that is, post-measurement) targets. For the target fitness, we look at 70% fitness and 99%

fitness. In figures 6.1 through 6.4, we present eigenvalue placement plot of the best individual and the superposition plot of the top 10 individuals. In Complex target mode, the constraint is harder because sign and complex value need to be met therefore we expect less scattering for pre-measurement targets. For the Probabilistic mode, only the magnitude needs to match. When we relax the target fitness to 70%, we expect more variability in the superposition plot since we individuals don't have to be as exact as in the 99% case.

Our general intuition is met by the eigenvalue placement figures. Where we found the best individual, the top 10 individual eigenvalue positions are in the general vicinity. For the two-qubit cases this is easier to see. In the complex target cases, we see the result of much harder restrictions, especially in the 99% target fitness mode. The eigenvalues are tighter clumped. In fact, in the two-qubit case for 99% and complex target, all top 10 individuals had the same eigenvalue placement. With probabilistic mode, there is some variability but not much. In the 5-qubit case, we see a more even spreading of the eigenvalues in the probabilistic target case as compared to the complex target case. Of interest is that the two-qubit case with 99% target fitness actually seems to have more variability in probabilistic mode than in the 70% case. This may be an artifact of the particular simulation results chosen but definitely noteworthy.

## 6.2 Degeneracy

The question of eigenvalue degeneracy and whether it plays a role in the formation or exploitation of the matrix evolved by the ES is asked. Before we continue, we present the following definition of eigenvalue degeneracy:

**Definition 20** *Given operator  $A$  defined by a  $N \times N$  square matrix. An eigen-*

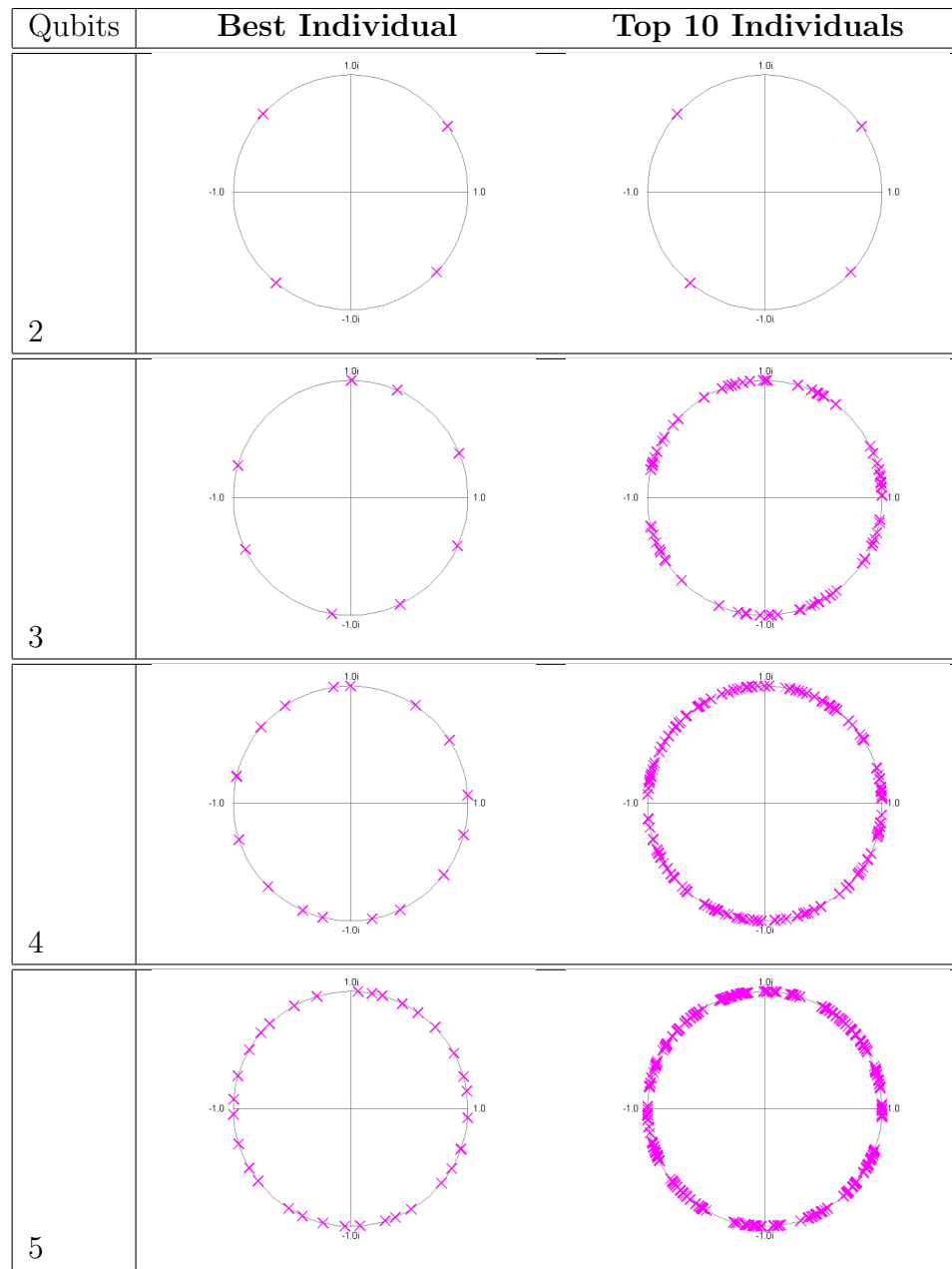


Figure 6.1: Eigenvalue plots for Complex Target, 70% Target Fitness

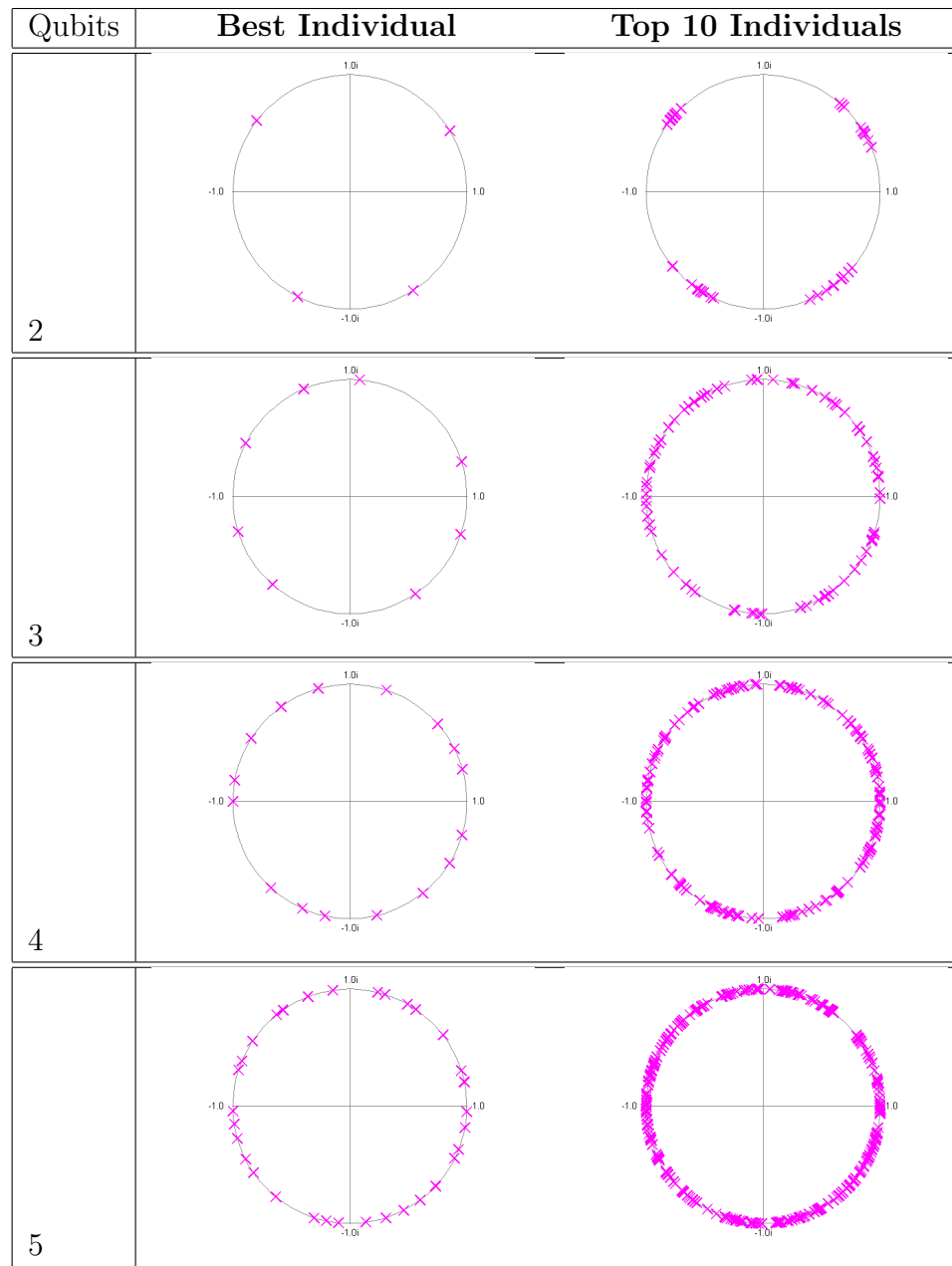


Figure 6.2: Eigenvalue plots for Probabilistic Target, 70% Target Fitness

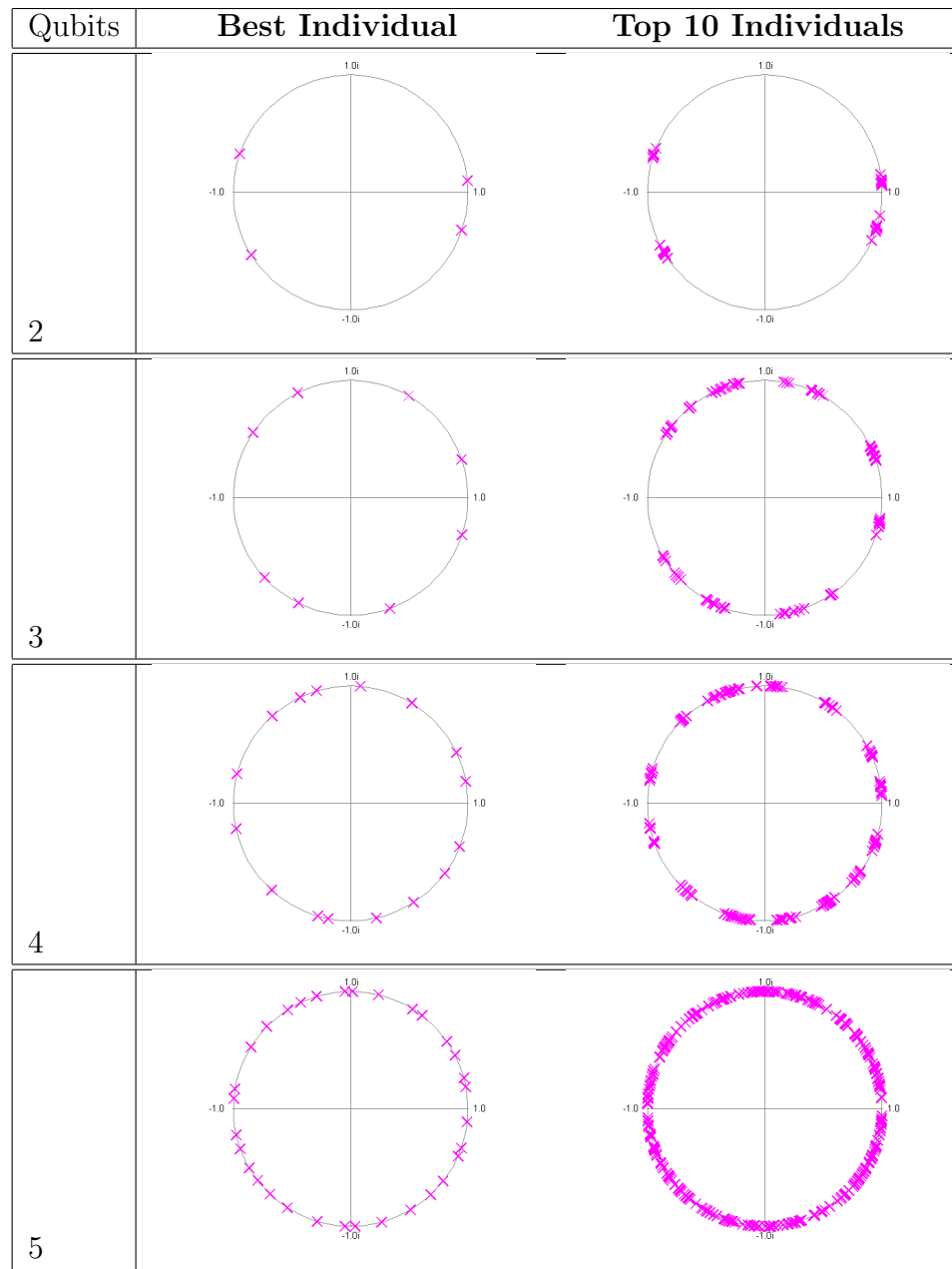


Figure 6.3: Eigenvalue plots for Complex Target, 99% Target Fitness

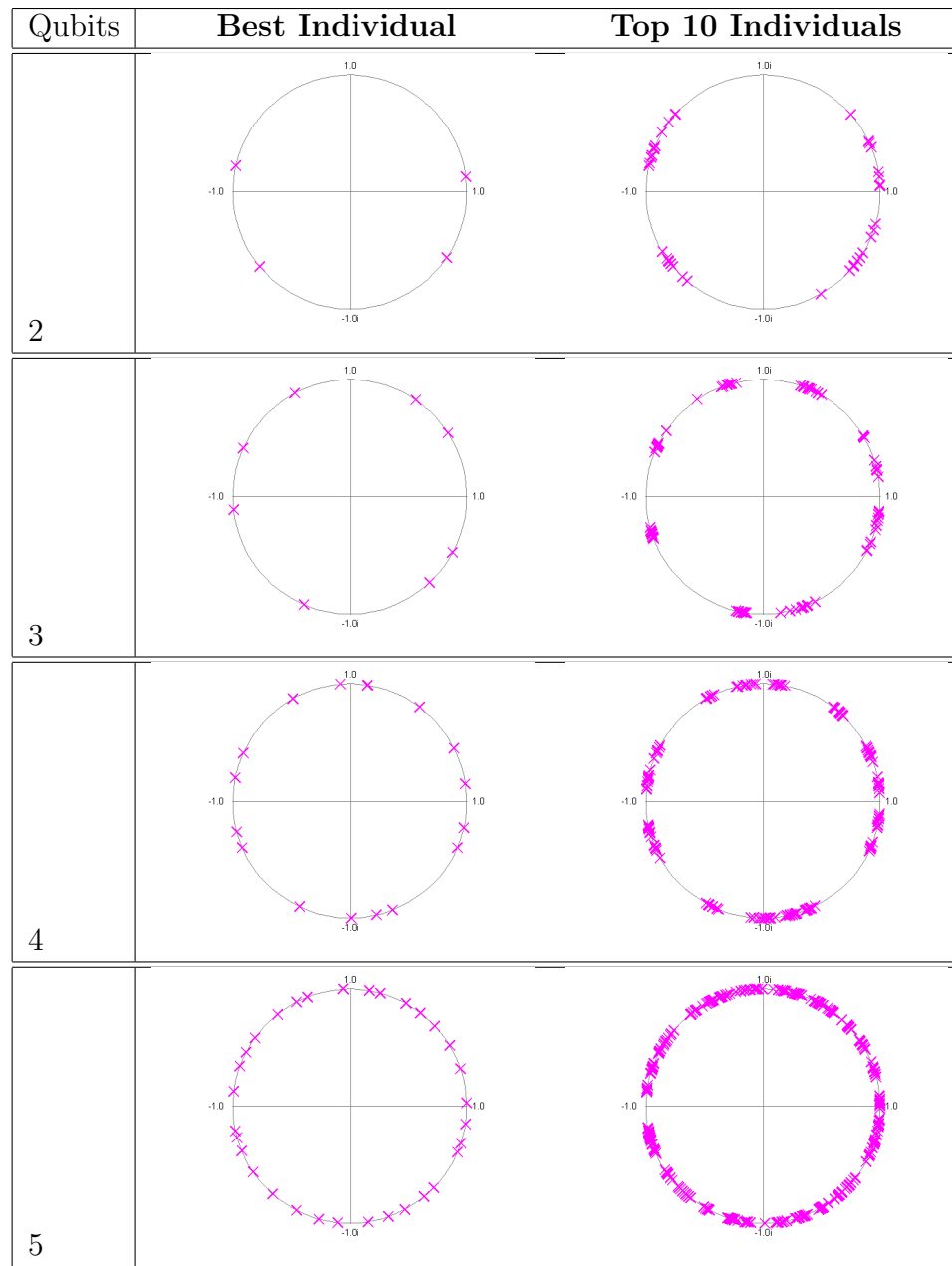


Figure 6.4: Eigenvalue plots for Probabilistic Target, 99% Target Fitness

value  $e_i$  of a operator  $A$  is said to be  $k$ -fold **degenerate** if there exists  $k$  linearly-independent eigenvectors  $\{|u_1\rangle, |u_2\rangle, \dots\}$  with the same eigenvalue. Therefore,  $A|u_i\rangle = e_i|u_i\rangle$  for  $i = 1, 2, \dots, k$ . Thus, the system defined by  $A$  is degenerate. The implication is also that  $A$  is no longer defined by  $N$  linearly-independent eigenvectors, but less depending on the amount of degeneracy.

However, our definition of a unitary operator from section 2.1.6 appears to form a contradiction that implies unitary matrices of order  $N$  must have a complete defined orthonormal set of  $N$  eigenvectors. This is still true, but with degeneracy, we simply will have more than one linearly independent set of basis vectors.

To look at this problem further we first examine a couple definitions which impose constraints on eigenvalues and eigenvectors for Hermitian and Unitary matrices and then show the proof that it is legal to have a unitary matrix with degeneracy.

**Definition 21** *The eigenvalues,  $\{w\}$ , of a Hermitian matrix,  $W$ , are real.*

Given  $W$  and one of its eigenvalues,  $w$ , then we have the relationship:  $W|w\rangle = w|w\rangle$ . This implies that the relationship  $\langle w|W|w\rangle = w\langle w|w\rangle$ . Since  $W$  is hermitian,  $W = W^T$ , and therefore  $\langle w|W|w\rangle = w^*\langle w|w\rangle$ . Subtracting the left side from the right and setting to zero we get  $(w - w^*)\langle w|w\rangle = 0$ . Since  $\langle w|w\rangle > 0$ , (if  $W$  were unitary it would have an absolute value  $|w| = 1$ ), then we must conclude that  $w = w^*$  meaning it does not have a complex component and therefore is real valued.

**Definition 22** *The eigenvalues,  $\{u\}$ , of a unitary matrix,  $U$ , are complex with unit magnitude and mutually orthonormal.*



Given unitary matrix,  $U$  with any two arbitrary eigenvectors  $|u_i\rangle$  and  $|u_j\rangle$  with their associated eigenvalues  $u_i$  and  $u_j$ , we have the relationship:  $U|u_i\rangle = u_i|u_i\rangle$  and  $U|u_j\rangle = u_j|u_j\rangle$ . Then,  $\langle u_j|U^T U|u_i\rangle = u_j^* u_i \langle u_j|u_i\rangle$ . Since  $U$  is unitary,  $U^T U = I$  and therefore,  $\langle u_j|u_i\rangle = u_j^* u_i \langle u_j|u_i\rangle$ . Through subtraction we get:  $(1 - u_j^* u_i) \langle u_j|u_i\rangle = 0$ . If  $i = j$  then  $\langle u_j|u_i\rangle = 1$  and therefore  $u_j^* u_i = 1$ . If  $i \neq j$  then  $(1 - u_j^* u_i) = 0$ . Because  $u_j \neq u_i$ , this implies  $u_j^* u_i \neq u_i^* u_i \Rightarrow u_j^* u_i \neq 1$ . Therefore,  $\langle u_j|u_i\rangle = 0$  and must be orthogonal.

**Definition 23** A Unitary matrix,  $U$ , with eigenvectors  $\{|u\rangle\}$  and associated eigenvalues  $\{u\}$  can be degenerate.

Assume  $U$  has two eigenvectors,  $|u_1\rangle$  and  $|u_2\rangle$ , associated with eigenvalue  $u$  so that it is degenerate. We know that  $U|u_1\rangle = u|u_1\rangle$  and that  $U|u_2\rangle = u|u_2\rangle$ . Then  $U(a_1|u_1\rangle + a_2|u_2\rangle) = u(a_1|u_1\rangle + a_2|u_2\rangle)$  for any  $a_1, a_2$ . Thus, there exists a whole subspace spanned by eigenvectors  $|u_1\rangle$  and  $|u_2\rangle$  with elements that are eigenvectors of  $U$  with eigenvalue  $u$ .

Let's look at the single qubit case and analyze the behavior of the eigenvalues and eigenvectors. For brevity, we'll use the single qubit form from our random unitary matrix method as shown in equation 6.2.1.

$$\begin{pmatrix} \cos(\phi)e^{i\psi} & \sin(\phi)e^{i\chi} \\ -\sin(\phi)e^{-i\chi} & \cos(\phi)e^{-i\psi} \end{pmatrix} \quad (6.2.1)$$

First we'll look at the eigenvalues and the condition(s) that allow us to have equivalent eigenvalues. If we defined a general  $2 \times 2$  matrix as so,

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

Then the generalized equation for the two eigenvalues is given by the *characteristic equation* as

$$\lambda_0 = + \frac{\sqrt{a^2 - 2ad + 4bc + d^2 + a + d}}{2}$$

$$\lambda_1 = - \frac{\sqrt{a^2 - 2ad + 4bc + d^2 + a + d}}{2}$$

By substituting  $a$ ,  $b$ ,  $c$ , and  $d$  with the expressions from our general unitary matrix, we get the following expressions for the eigenvalues:

$$\lambda_0 = \cos(\psi) \cos(\phi) + i \sqrt{1 - \cos^2(\psi) \cos^2(\phi)}$$

$$\lambda_1 = \cos(\psi) \cos(\phi) - i \sqrt{1 - \cos^2(\psi) \cos^2(\phi)}$$

We note that the eigenvalues are not dependent on  $\chi$ .

The condition where  $\lambda_0 = \lambda_1$  is such when the term under the radical is zero. That is,

$$1 - \cos^2(\psi) \cos^2(\phi) = 0 \tag{6.2.2}$$

which is satisfied when both cosine terms are unity which occurs when both  $\psi$  and  $\phi$  are integer multiples of  $\pi$ .

The most basic example is the identity gate who is realized in equation 6.2.1 when  $\phi = \psi = 0$ .

There is no issue with a quantum operator having equal eigenvalues as long as there is a complete set of linearly-independent (and orthonormal) eigenvectors.

Looking back at our eigenvalue plots from figures 6.1 to 6.4, we notice that the eigenvalue placement of the best individual is well distributed around the unit circle. This behavior seems to be standard for many of the unitary operators

evolved by our ES. The likely reason for this behavior stems from our random unitary matrix method described in section 3.2 which was designed for CUE (circular uniform ensembles). Translated roughly, our random unitary matrices will tend to have a spectrum of eigenvalues that are even spread on the unit circle.

### 6.3 Interpretation of Eigenvalues

Certainly the understanding of eigenvalues and eigenvectors has been investigated. We highlight example research in our overview of quantum state separability in section 3.3 using eigenvalues and eigenstates to detect the degree of entanglement [61]. Kitaev gives an approach to a general class of problems in [74] using eigenvalue estimation. Also, the use of eigenvalues for unitary operators to reveal common structures of quantum algorithms and relations to interferometry experiments is presented by Cleve et al in [75]. Galindo and Martin-Delgado do a superb analysis of the eigenvalue nature in [76] for a family of Grover’s algorithms. Shevi et al perform a detailed mathematical analysis of the role of the eigenvalues and their placement on the unit circle in [77] for their construction and analysis of a quantum random walk based searching algorithm. Ambainis et al conduct a similar investigation in [78] where eigenanalysis is used to analyze the use of coins to speed up quantum walks searching a  $\sqrt{N} \times \sqrt{N}$  grid. Mosca and Ekert expand on Kitaev’s approach and eloquently relate using eigenvalue estimation for solving Abelian hidden subgroup problems in [79]. Similar techniques of eigenvalue approximation using a coarse grid are used by Abrams [80] and Jaksch [81].

We take a much simpler approach and examine two methods of expressing

a unitary operator, namely, spectral representation and outer-product form.

Spectral representation is a well known method and was described in section 2.1.8. The outer-product method of quantum logic is less formally defined and more-or-less an adopted method used in common quantum logic books to express a quantum operator using input and output state mappings and is used in section 2.2.2.

We again define the spectral representation of a unitary quantum operator as:

$$U = \sum_{i=0}^{N-1} \lambda_i |v_i\rangle \langle v_i| \quad (6.3.1)$$

Where the  $|v_i\rangle$  vectors form a set of orthonormal eigenvectors with  $\lambda_i$  being the corresponding set of eigenvalues.

The definition of an outer-product expression of a quantum operator can be summarized this way:

**Definition 24** *Given an  $n$  qubit  $N \times N$  unitary operator,  $U$ , such that  $N = 2^n$ , a complete standard computational basis set of input ket vectors,*

$$\mathbf{I} = \{|00\rangle, |01\rangle, \dots, |N-1\rangle\},$$

*a complete set of target bra vectors,*

$$\mathbf{B} = \{\langle B_0|, \langle B_1|, \dots, \langle B_{N-1}|\},$$

*and the mapping,  $\mathbf{M}$ , of each input ket vector to an output bra vector through*

an outer-product operation such that

$$\mathbf{M} = \{|00\rangle\langle B_0|, |01\rangle\langle B_1|, \dots, |N-1\rangle\langle B_{N-1}|\},$$

the outer-product expression of a unitary operator,  $U$ , is defined as the sum of the outer-products:

$$U = \sum_{j=0}^{N-1} M_j = \sum_{j=0}^{N-1} |j\rangle\langle B_j| \quad (6.3.2)$$

We now have two definitions of a unitary operator,  $U$ . We make the assumption that we can set these two expressions to be equal to each other. That is,

$$\sum_{i=0}^N \lambda_i |v_i\rangle\langle v_i| = \sum_{j=0}^{N-1} |j\rangle\langle B_j| \quad (6.3.3)$$

The goal is to relate the set of eigenvalues back to the more commonly used outer-product method. The reason for doing this is twofold. For one, we typically examine the behavior of a quantum operator on the standard computational basis. For example, we think of the inverter ( $X$ ) as operating on the basis  $|0\rangle$  or  $|1\rangle$  rather than, for example, the set  $(1/\sqrt{2})(-|0\rangle + |1\rangle)$  and  $(1/\sqrt{2})(|0\rangle + |1\rangle)$ . Both are perfectly acceptable but the later is obviously more cumbersome to deal with. Secondly, the outer-product form explicitly shows the mapping from input state to output state. Not only that, but the output state will be a superposition of the standard computational basis states. This is also desirable because it gives us a more intuitive sense of the *function* of the operator. Going back to our example with the quantum inverter (which we use as a detailed example later), it is not obvious from the spectral representation that the gate functions as an inverter. However, in outer-product form, it is blatantly obvious.

To derive a mapping from spectral representation to outer-product form, we dive a little deeper into equations and consider the generalized single qubit case. In this case, we have the mapping:

$$|0\rangle \implies \langle a|$$

$$|1\rangle \implies \langle b|$$

This results in the outer-product expression for  $U$ :

$$U = |0\rangle \langle a| + |1\rangle \langle b| \quad (6.3.4)$$

For the spectral representation, we construct  $U$  such that it has arbitrary eigenstates  $a'$  and  $b'$  to derive:

$$U = \lambda_1 |a'\rangle \langle a'| + \lambda_2 |b'\rangle \langle b'| \quad (6.3.5)$$

Setting these two expression equal yields:

$$|0\rangle \langle a| + |1\rangle \langle b| = \lambda_1 |a'\rangle \langle a'| + \lambda_2 |b'\rangle \langle b'| \quad (6.3.6)$$

And more appropriately, we show this in matrix form noting  $\langle a| = [a_1, a_2]$ ,  $\langle b| = [b_1, b_2]$ , and likewise:

$$\begin{pmatrix} a_1 & a_2 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ b_1 & b_2 \end{pmatrix} = \lambda_1 \begin{pmatrix} a'_1 a'_1 & a'_1 a'_2 \\ a'_2 a'_1 & a'_2 a'_2 \end{pmatrix} + \lambda_2 \begin{pmatrix} b'_1 b'_1 & b'_1 b'_2 \\ b'_2 b'_1 & b'_2 b'_2 \end{pmatrix} \quad (6.3.7)$$

We can now develop a general mapping by equating the position of each matrix component on the left to the summation of the same indexed components multiplied by their respective eigenvalues on the right side of the equation.

Namely,

$$a_1 = \lambda_1 a'_1 a'_1 + \lambda_2 b'_1 b'_1$$

$$a_2 = \lambda_1 a'_1 a'_2 + \lambda_2 b'_1 b'_2$$

$$b_1 = \lambda_1 a'_2 a'_1 + \lambda_2 b'_2 b'_1$$

$$b_2 = \lambda_1 a'_2 a'_2 + \lambda_2 b'_2 b'_2$$

We proceed to the 2 qubit case but only enough to show the general pattern.

For the 2 qubit case we introduce two new target states,  $\langle c|$  and  $\langle d|$ , and form the new outer-product expression:

$$U = |00\rangle \langle a| + |01\rangle \langle b| + |10\rangle \langle c| + |11\rangle \langle d| \quad (6.3.8)$$

And we'll set this equal to the spectral representation:

$$U = \lambda_1 |a'\rangle \langle a'| + \lambda_2 |b'\rangle \langle b'| + \lambda_3 |c'\rangle \langle c'| + \lambda_4 |d'\rangle \langle d'| \quad (6.3.9)$$

Without expanding on the matrix form, we are able to develop the mapping for the first target state,  $\langle a|$ :

$$a_1 = \lambda_1 a'_1 a'_1 + \lambda_1 b'_1 b'_1 + \lambda_1 c'_1 c'_1 + \lambda_1 d'_1 d'_1$$

$$a_2 = \lambda_1 a'_1 a'_2 + \lambda_1 b'_1 b'_2 + \lambda_1 c'_1 c'_2 + \lambda_1 d'_1 d'_2$$

$$a_3 = \lambda_1 a'_1 a'_3 + \lambda_1 b'_1 b'_3 + \lambda_1 c'_1 c'_3 + \lambda_1 d'_1 d'_3$$

$$a_4 = \lambda_1 a'_1 a'_4 + \lambda_1 b'_1 b'_4 + \lambda_1 c'_1 c'_4 + \lambda_1 d'_1 d'_4$$

We now see the pattern emerge and to generalize we change our notation so that a target vector is denoted by  $\vec{v}$  and eigenvectors are denoted by  $\vec{e}$ . From

our expressions above,  $\langle a|$  is the target vector for input  $|00\rangle$  and so with our new notation,  $\langle v_1| = \langle a|$ . Likewise,  $\langle v_2| = \langle b|$ ,  $\langle v_3| = \langle c|$ , etc. This way we can refer to the  $i^{th}$  target vector by  $\vec{v}_i$  whose  $j^{th}$  component is given by  $v_{ij}$ . We likewise represent the eigenvectors of in the spectral representation basis using  $\vec{e}$  such that  $\vec{e}_1 = |a'\rangle$ ,  $\vec{e}_2 = |b'\rangle$ , and so on and so forth. Thus, the  $i^{th}$  eigenvector is  $\vec{e}_i$  whose  $j^{th}$  component is given by  $e_{ij}$ .

Using this new convention we go back to the single qubit case and we can re-write the expression for the output vectors as:

$$\vec{v}_1 = \begin{bmatrix} \lambda_1 e_{11} e_{11} + \lambda_2 e_{21} e_{21} & \lambda_1 e_{11} e_{12} + \lambda_2 e_{21} e_{22} \end{bmatrix} \quad (6.3.10)$$

$$\vec{v}_2 = \begin{bmatrix} \lambda_1 e_{12} e_{11} + \lambda_2 e_{22} e_{21} & \lambda_1 e_{12} e_{12} + \lambda_2 e_{22} e_{22} \end{bmatrix} \quad (6.3.11)$$

Finally, we are able to construct our general equation which relates the output vectors in the outer-product form (which uses the standard computational basis set) to the eigenvalues and eigenvectors used in the spectral representation:

$$v_{ij} = \sum_{k=1}^N \lambda_k e_{ki} e_{kj} \quad (6.3.12)$$

Let's consider an example, the Inverter (X):

Recall the X gate matrix,  $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ , and note that asking MATLAB to produce its eigenvalues and eigenvectors yields:



$$\lambda_1 = -1$$

$$\lambda_2 = 1$$

$$\vec{e}_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

$$\vec{e}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Therefore we have the spectral representation:

$$U = (-1) \frac{1}{2} \begin{bmatrix} -1 \\ 1 \end{bmatrix} \begin{bmatrix} -1 & 1 \end{bmatrix} + (1) \frac{1}{2} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \end{bmatrix} \quad (6.3.13)$$

Had we been given the eigenvalues and eigenvectors, it's not obvious that they are simply for a quantum inverting gate. Using equation 6.3.12 we can derive the outer-product form of the operator using standard computational basis vectors as inputs. We do this now knowing that we should get  $\vec{v}_1 = \langle 1|$  and  $\vec{v}_2 = \langle 0|$ :

$$\begin{aligned} \vec{v}_1 &= \begin{bmatrix} \lambda_1 e_{11} e_{11} + \lambda_2 e_{21} e_{21} & \lambda_1 e_{11} e_{12} + \lambda_2 e_{21} e_{22} \end{bmatrix} \\ &= \begin{bmatrix} (-1) \frac{-1}{\sqrt{2}} \frac{-1}{\sqrt{2}} + (1) \frac{1}{\sqrt{2}} \frac{1}{\sqrt{2}} & (-1) \frac{-1}{\sqrt{2}} \frac{1}{\sqrt{2}} + (1) \frac{1}{\sqrt{2}} \frac{1}{\sqrt{2}} \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 \end{bmatrix} = \langle 1| \\ \vec{v}_2 &= \begin{bmatrix} \lambda_1 e_{12} e_{11} + \lambda_2 e_{22} e_{21} & \lambda_1 e_{12} e_{12} + \lambda_2 e_{22} e_{22} \end{bmatrix} \\ &= \begin{bmatrix} (-1) \frac{1}{\sqrt{2}} \frac{-1}{\sqrt{2}} + (1) \frac{1}{\sqrt{2}} \frac{1}{\sqrt{2}} & (-1) \frac{1}{\sqrt{2}} \frac{1}{\sqrt{2}} + (1) \frac{1}{\sqrt{2}} \frac{1}{\sqrt{2}} \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \end{bmatrix} = \langle 0| \end{aligned}$$

We can thus write the outer-product form as simply:

$$U = |0\rangle \langle \vec{v}_1| + |1\rangle \langle \vec{v}_2| = |0\rangle \langle 1| + |1\rangle \langle 0| \quad (6.3.14)$$

And thus we've shown how the eigenvalues and eigenvectors relate to the outer-product form. Although we show an example with a relative simple quantum gate, the method is general and extensible to more complicated unitary operators. The limitation is that we assume a standard computational basis. In the more complicated case where we have an input state which is not a standard basis state but a linear superposition then it is possible that the target states for basis states is not that interesting. In such as case, it is debatable what the target mappings for basis states represent.

#### 6.4 Adjusting Eigenvalues

An interesting question is do we have the ability to adjust eigenvalues on the unit circle for a given unitary operator and what are its implications? For example, if we see two or more eigenvalues that are close in value, it might indicate that a certain level of degeneracy is desired. In this section we'll briefly analyze if this is possible and what it can do for us.

First we'll start with a very simple operator, the NOT gate (X) which is defined by the unitary matrix:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

The eigenvalues for the NOT gate are  $e_1 = 1$  and  $e_2 = -1$ . The eigenvectors are:

$$\vec{v}_1 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

and

$$\vec{v}_2 = \frac{1}{\sqrt{2}} \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

As we've established, the eigenvalues for unitary operators have modulus 1 and are of the form  $e^{i\theta}$ . So, the first question we may ask is can we substitute the known eigenvalues with arbitrary values as long as their modulus is unity?

The spectral decomposition method we described in section 2.1.8 allows us to test this hypothesis. Using spectral decomposition, the NOT gate can be expressed as:

$$X = e_1 |v_1\rangle \langle v_1| + e_2 |v_2\rangle \langle v_2| \quad (6.4.1)$$

We now re-write this expression substituting  $e_1$  with  $e^{ix}$  and  $e_2$  with  $e^{iy}$  but keeping the same eigenvectors.

$$X(x, y) = e^{ix} |v_1\rangle \langle v_1| + e^{iy} |v_2\rangle \langle v_2| \quad (6.4.2)$$

Next, we'll see what  $X(x, y)$  becomes as we sample some typical angle combinations for  $x$  and  $y$ :

$$X(0, 0) = I$$

$$X(0, \pi) = X$$

$$X(\pi, 0) = -X$$

$$X(\pi, \pi) = -I$$

We note that all the above instances of  $X(x, y)$  are known to be unitary and

in fact, a general unitary test reveals:

$$X(x, y)^\dagger X(x, y) = I$$

Therefore, the general form of  $X(x, y)$  is in fact unitary! This means we can adjust the eigenvalue placement for the NOT operator and reconstruct it using the spectral decomposition method and we would still have a valid unitary operator.

We'll examine if this same method works on the Hadamard gate which is defined by the matrix:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

The eigenvalues for the  $H$  gate are  $e_1 = 1$  and  $e_2 = -1$ . The eigenvectors are:

$$\vec{v}_1 = \begin{pmatrix} 0.924 \\ 0.383 \end{pmatrix}$$

and

$$\vec{v}_2 = \begin{pmatrix} -0.383 \\ 0.924 \end{pmatrix}$$

We again make the same generalization and assume we can re-write the spectral decomposition of the Hadamard operator using  $e_1 = e^{ix}$  and  $e_2 = e^{iy}$  while still using the eigenvectors from the Hadamard:

$$H(x, y) = e^{ix} |v_1\rangle \langle v_1| + e^{iy} |v_2\rangle \langle v_2| \quad (6.4.3)$$

Again we sample with various angles:

$$H(0, 0) = I$$

$$H(0, \pi) = H$$

$$H(\pi, 0) = -H$$

$$H(\pi, \pi) = -I$$

And again, we prove that the general expression  $H(x, y)$  is unitary by noting that  $H(x, y)^\dagger H(x, y) = I$ .

Therefore, we make the broad hypothesis:

*Given a unitary operator,  $\mathbf{U}$ , with the set of eigenvalues  $\{e_1, \dots, e_n\}$  and corresponding set of eigenvectors  $\{\vec{v}_1, \dots, \vec{v}_n\}$ , a new generalized unitary operator can be constructed using the spectral decomposition of  $\mathbf{U}$  and by substituting one or more eigenvalues with an arbitrary value of the form  $e^{i\theta}$  but keeping the original eigenvectors. Thus, the new general unitary operator,  $\mathbf{U}'$ , would have the form:*

$$\mathbf{U}'(\theta_1, \dots, \theta_n) = \sum_{i=1}^n e^{i\theta_i} |v_i\rangle \langle v_i| \quad (6.4.4)$$

Now that we have a method of adjusting eigenvalues for a given unitary operator and then re-constructing a new unitary operator, let's apply it to a real example. We'll choose the best 5-qubit best individual from figure 6.2 and shown larger in figure 6.5 where we notice there are a few pairs of eigenvalues (circled in blue) which are very close together — enough that perhaps degeneracy was being approached and would be, in fact, desirable.

We'll call this matrix,  $U$ , and note that as a 5-qubit operator it has  $2^5 = 32$

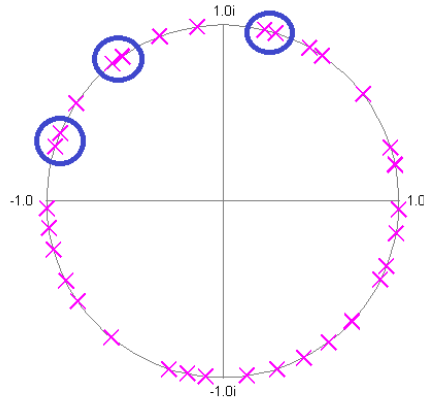


Figure 6.5: Best 5 qubit, probability mode, 70% target, pairs close in value to be made degenerate are circled in blue.

eigenvalues which are listed below,

$$e[1] = 0.7958 + 0.6056i$$

$$e[2] = 0.5638 + 0.8259i$$

$$e[3] = 0.4915 + 0.8709i$$

$$e[4] = 0.2988 + 0.9543i$$

$$e[5] = 0.2363 + 0.9717i$$

$$e[6] = 0.9532 + 0.3025i$$

$$e[7] = 0.9784 + 0.2067i$$

$$e[8] = -0.1450 + 0.9894i$$

$$e[9] = 0.9988 - 0.0486i$$

$$e[10] = 0.9828 - 0.1846i$$

$$e[11] = -0.3585 + 0.9335i$$

$$e[12] = 0.9293 - 0.3693i$$

$$e[13] = -0.5713 + 0.8207i$$

$$e[14] = 0.8945 - 0.4471i$$

$$e[15] = -0.6259 + 0.7799i$$

```

e[16] = 0.7311 - 0.6822i
e[17] = -0.8306 + 0.5569i
e[18] = -0.9234 + 0.3838i
e[19] = -0.9505 + 0.3108i
e[20] = 0.5994 - 0.8004i
e[21] = -0.6317 - 0.7752i
e[22] = 0.4597 - 0.8881i
e[23] = -0.8235 - 0.5673i
e[24] = -0.8910 - 0.4539i
e[25] = -0.9992 - 0.0404i
e[26] = 0.3071 - 0.9517i
e[27] = -0.3048 - 0.9524i
e[28] = -0.9613 - 0.2756i
e[29] = -0.9883 - 0.1528i
e[30] = -0.2017 - 0.9794i
e[31] = 0.1350 - 0.9908i
e[32] = -0.0988 - 0.9951i

```

For brevity we omit showing the eigenvectors.

We've identified 3 pairs of eigenvalues which are within an absolute difference ( $|e_i - e_j|$ ) of 0.08 (this is an arbitrary limit). The way we determine these pairs is through a simple search using the following MATLAB routine:

```

% get our eigenvalues and eigenvectors
[V,D] = eig(U);
% for each eigenvalue, determine abs
% difference to another eigenvalue

```

```

for i=1:31
    for j=i+1:32
        % only look at unique eigenvalues
        if(i ~= j)
            % note pairs with < 0.08 difference
            if(abs(D(i,i)-D(j,j))<0.08)
                disp(i)
                disp(j)
                disp(abs(D(i,i)-D(j,j)))
            end
        end
    end
end
end
end

```

The 3 pairs are  $\{e_4, e_5\}$ ,  $\{e_{13}, e_{15}\}$ , and  $\{e_{18}, e_{19}\}$ . To form the degenerate pairs, we'll set each element of the pair to an average of their values such that given pair  $e_i, e_j$ , the new value will be  $e'_i = e'_j = (e_i + e_j)/2$ . If we had a case of a larger number of eigenvalues within a certain distance of each other, then perhaps we could consider setting them all to an average of their eigenvalues.

Our new degenerate pairs are formed:

$$\begin{aligned}
 e_4 = e_5 &= \frac{((0.2988+0.9543i)+(0.2363+0.9717i))}{2} \\
 &= 0.2676 + 0.9630i \\
 e_{13} = e_{15} &= \frac{((0.2988+0.9543i)+(0.2363+0.9717i))}{2} \\
 &= -0.5986 + 0.8003i \\
 e_{18} = e_{19} &= \frac{((0.2988+0.9543i)+(0.2363+0.9717i))}{2} \\
 &= -0.9369 + 0.3473i
 \end{aligned}$$

To reconstruct the new unitary operator, we implement our spectral decom-



position form in MATLAB and override our eigenvalues with our newly formed degenerate pairs:

```
% get eigenvalues and eigenvectors
[V,D] = eig(U)

% override the eigenvalues with our degenerate pairs
% D(4,4) = D(5,5) = 1/2*(D(4,4)+D(5,5))
% D(13,13) = D(15,15) = 1/2*(D(13,13)+D(15,15))
% D(18,18) = D(19,19) = 1/2*(D(18,18)+D(19,19))

% zero out our 32x32 U matrix
U = eye(32)-eye(32)

% re-construct using spectral representation
for i=1:32
    U = U + D(i,i)*(V(:,i)*ctranspose(V(:,i)))
end
```

This forms our new  $U$  operator which should now have 3 degenerate eigenvalue pairs and should be unitary. To verify, we test its unitary property ( $U^\dagger U = I$ ). This is done in MATLAB using:

```
ctranspose(U)*U
```

At this point, some number accuracy is lost but the result of the above expression is within a small error margin of being an identity matrix.

We also look at the new eigenvalue placement which should show all other values untouched except for our new degenerate pairs. We show this comparison in figure 6.6. The squares indicate where our degenerate pairs were formed by using their average values.

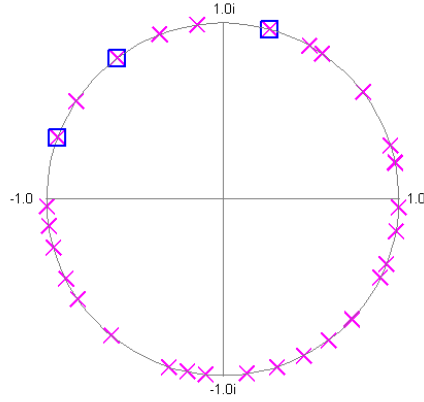


Figure 6.6: Adjusted 5-qubit operator eigenvalue placement. Squares indicate our newly formed degenerate pairs.

Next we look at the impact to our behavior, or rather, the fitness of the new unitary operator versus the original before adjusting some of the eigenvalues. The hope is that by predicting a level degeneracy was desired, that by manually adjusting, that our fitness will also increase. We recall the original  $U$  was constructed with a goal such that with an initial state in an equal superposition state, the final state after applying  $U$  would be  $|0 \dots 0\rangle$ . And in particular, we were looking for a probability distribution of the output vector showing that we would measure state  $|00000\rangle$  at least 70% of the time.

In figure 6.7 we show the probability distribution of the original operator as compared to our new eigenvalue-adjusted operator. The probability of measuring the desired state (state-1) with the original evolved operator is 70.68%. Remarkably, the probability of measuring the same desired state with our adjusted operator is 70.87%. It is quite possible that it would have taken many generations of the ES just to increase the fitness even by that small amount, especially since we're dealing with 5-qubit operators.

In summary, we described a method to adjust the eigenvalues of a unitary

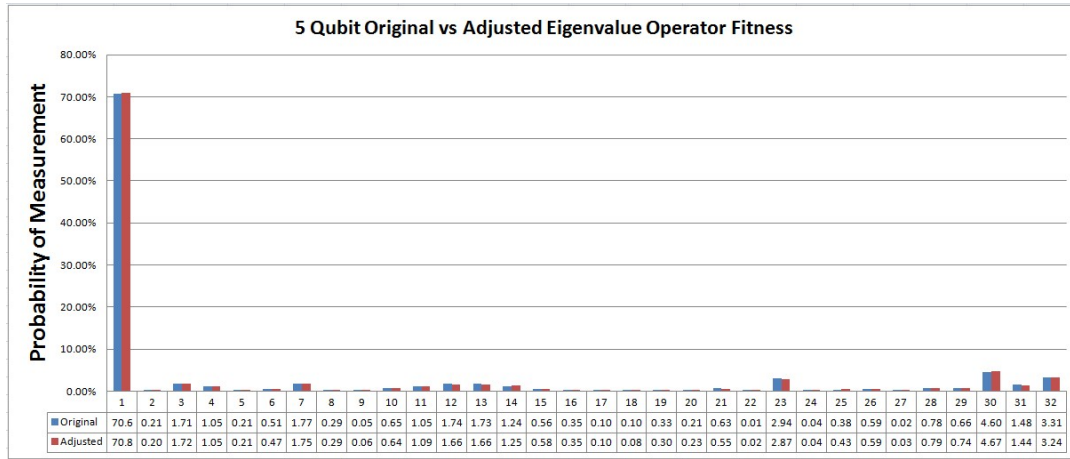


Figure 6.7: 5-qubit probability distribution of original operator (blue) versus one with adjusted eigenvalues to add degeneracy (red).

operator which results in a new operator which still retains its unitary property. We demonstrated at least one case where we used our method to adjust the eigenvalues and the resulting operator had better measurement accuracy. We showed this with a 5-qubit example where the probability of measuring the desired state was enhanced by simply setting our 3 closest eigenvalue pairs to their respective average values, thus introducing degeneracy. This was entirely visual and this definitely paves the way for more intricate methods to adjust the eigenvalues and examine the benefit.

## Chapter 7

### Logic Synthesis Results

For logic synthesis we used the Qubiter [3] software designed by Dr. Robert Tucci. Qubiter is provided for free to researchers on Dr. Tucci's website at <http://www.ar-tiste.com/qubiter.html> and is covered by US Patent 6,456,994. At the time of this research, Qubiter was the only publicly available quantum synthesis tool capable of handling general unitary quantum operators. Although other quantum compilers existed, they were either not mature enough or too limited. We give a brief overview of quantum compilers in section 3.4.5.

The goal of the logic synthesis was to look for more information that could help us design or enhance quantum operators. For example, do we see patterns in the usage of specific quantum gates or structures? How does the quantum gate count change as we alter the unitary matrix or increase the complexity?

Unfortunately, we hit a very fundamental barrier with Qubiter which we'll explain right away since it quickly limited our planned studies. Namely, Qubiter uses a pre-determined structure for synthesizing a quantum operator based on *Cosine-Sine Decomposition* (or CSD). We described CSD in section 3.4.4. There is nothing wrong with CSD, however, Qubiter does little to no optimizations on the resultant quantum netlist and the algorithm relies heavily on phase and rotation gates of arbitrary angles. In fact, Qubiter synthesized netlists were

made up of only 4 types of gates: CNOT, C-Phase, ROTY, and potentially one overall PHASE gate. This means that we had one static gate (CNOT) and the rest used arbitrary angles. To make matters worse, we see approximately the same circuit structure for a given number of qubits no matter how different the function of the operators. This is also because of the static mapping Qubiter uses from a given unitary operator to a circuit by following a deterministic algorithm. We'll see this in more detail later, however, it's sufficient to say this made doing a structural analysis of the quantum netlist mostly fruitless.

In section 7.1 we describe how the Qubiter software is compiled and run since this was a very difficult. In sections 7.2 and 7.3 we give an overview of the input collateral is needed and how we interpret the output of the software. In section 7.4, we show how we synthesized the results from our best single instance operators as reviewed in our eigenanalysis done in section 6. We give our summary and conclusions in section 7.7.

## 7.1 Compiling and Running Qubiter

Qubiter has a somewhat surprising characteristic, it was developed on an older MacOS. In fact, the original version we used (version 1.01) was developed on Mac OS9 using CodeWarrior<sup>1</sup> Professional (CWP) versions 1 and 2. Although the sourcecode is considered C++, it has CWP-specific compiling commands which make porting the sourcecode difficult. In addition, Qubiter uses a modified version of BLAS<sup>2</sup> as well as math libraries geared towards the PowerPC (PPC) architecture. Qubiter did not use the STL (Standard Template Library),

---

<sup>1</sup>CodeWarrior is produced by FreeScale and more information can be found at <http://www.freescale.com/codewarrior>

<sup>2</sup>BLAS stands for "Basic Linear Algebra System", more information can be found at <http://www.netlib.org/blas/>

but rather used a customized template library written by a 3rd-party developer which was included with CodeWarrior. Lastly, there are many files — hundred, perhaps — and libraries which make up Qubiter and the top-level project file which maintains the order and linking commands to generate the libraries and the final Qubiter executable is in a proprietary format and there was no way to convert it to a GNU Makefile or Microsoft Visual-C++ project file.

We made an attempt to port the Qubiter sourcecode so that we could compile and run it on a standard linux OS using GNU C++ (g++). However, given the highly customized and proprietary nature of the code to an older Mac OS and libraries, it became a very difficult process and we decided to abandon the effort. We kept in good communication with Dr. Tucci throughout this process. We finally resolved to simply purchase an older Mac computer (an iMac G3, PowerPC based, 450MHz) which had both OS-X and OS9 running in Mac-Classic mode. However, we still had issues with compilations because of lacking 3rd-party libraries. Eventually, Dr. Tucci uploaded his Qubiter executable to us which was able to run on our “old” Mac.

## 7.2 Qubiter Input Files

Upon executing Qubiter, it looks for a file called “qbtr-params.in” which specifies the parameters of the synthesis. An example params file is shown:

```
//file prefix indicating matrix name?  
5q_best_p1_t99  
//Do compilation?(0,1)  
1  
//Do decompilation?(0,1)  
1
```

```

//Do light right optimization?(0,1)
1
//Do complex D matrix optimization?(0,1)
1
//Specify pmut_opt (permutation optimization level) (0,1,2)
1
//If pmut_opt=1, should I run through all bit permutations?(0,1)
1
//If pmut_opt=1, should I keep results for the identity permu.?(0,1)
1
//If pmut_opt=1 & answered NO to prev. question, then give another permu.
//Give length on 1st line, permu. on 2cd (0 1 2 ... is identity).
4
2      1      3      0

```

The first entry is the name of file containing the description of the quantum unitary matrix to be synthesized. In the above example, we are pointing to the unitary matrix of our most fit individual from the 5-qubit, probability mode case. We won't cover the other parameters, for the most part we left these at their suggested default values.

The file which describes the unitary matrix is also a “.in” file. In the above example, the matrix file would be “5q\_best\_p1\_t99.in”. The format of the input matrix file is rather simple. We first specify how many qubits are represented by the matrix and then describe the matrix using real-imaginary pairs. We specify the entries of the matrix as a string of columns. For example, we first write down column 1, then append column 2, and so on. For an  $n$  qubit matrix, we should have  $2^n$  entries. We show an example from a 2 qubit unitary matrix:

```

//number of bits

```

2

```
//matrix u, as a string of columns
4.5488998475545628e-01 -2.7300421615946958e-02
1.2462256983509622e-01 -1.8462155396393282e-01
7.0185746166411134e-01 8.9098769276772558e-02
3.8208328574244116e-01 -3.1013506932374924e-01
4.5100703438276363e-01 -2.7275921393111951e-02
-5.9658489318826580e-01 -1.6165649550398006e-01
1.3737996029357016e-01 -4.6714541565998940e-02
-1.7557008766957286e-01 6.0159950478873636e-01
5.3810841699814282e-01 -2.0609961987595526e-02
3.8601128516668837e-01 5.8409387424338000e-01
-8.4172067771916253e-03 -6.7626353441249858e-03
-4.6875111969062311e-01 0.0000000000000000e+00
5.4603387750316812e-01 -6.3271984062368680e-03
1.0034604220263768e-02 -2.7940415189115725e-01
-6.8873987078214327e-01 -6.2632409534129727e-02
3.0048327528278729e-01 -2.3465713445274747e-01
```

### 7.3 Qubiter Output Files

Qubit produces a number of output files after synthesis. We briefly summarize them here:

The file that is most interesting to us is the “engl.out” file which contains the final result of the quantum circuit. This file can be lengthy for so we’ll show



File	Description
chk.out	This is a copy of the input matrix “.in” file after being loaded by Qubiter.
engl.out	This is the detailed result of the synthesis describing the gates and connectivity.
err.out	This is the error matrix produced from the difference between the input and result matrices.
log.out	This is the detailed log file from Qubiter.
pict.out	This is an ASCII visual representation of the synthesized circuit.
pmut.out	This is the result of various permutations tried and the number of steps (gates) that resulted.

Table 7.1: Table of Qubiter output files.

an example from a 2 qubit run which corresponds to figure 7.9.

```

2
{ angles: 0.00000000 ~ 0.00000000 ~ 0.00000000 ~ 0.00000000
}

=====

CPHA    0      T      25.0747572
CPHA    1      T      24.7735822
CPHA    1      T      0      T      112.522734
{ angles: 45.4729213 & 3.03109044
  ROTY    0      24.2520059
  CNOT    1      T      0
  ROTY    0      21.2209154
  CNOT    1      T      0
}

CPHA    0      T      180.000000
CPHA    1      T      0      T      158.336638

```

```

=====
{ angles: 0.00000000 ^ 0.00000000 ^ 0.00000000 ^ 0.00000000
}

=====
{ angles: 49.4416822 86.6913857
  ROTY    1      68.0665339
  CNOT    0      T        1
  ROTY    1      341.375148
  CNOT    0      T        1
}

=====
{ angles: 0.00000000 ^ 0.00000000 ^ 0.00000000 ^ 0.00000000
}

=====
CPHA    0      T      231.741129
CPHA    1      T      300.430356
CPHA    1      T      0      T      302.815461
{ angles: 40.6683918 & 52.0097530
  ROTY    0      46.3390724
  CNOT    1      T      0
  ROTY    0      354.329319
  CNOT    1      T      0
}

CPHA    0      T      267.169649
CPHA    1      T      0      T      207.391615

```

```

=====
{ angles: 0.00000000 ^ 0.00000000 ^ 0.00000000 ^ 0.00000000
}
=====
PHAS      280.138651
{ number of steps = 22
}

```

Starting at the top of the file, it describes the sequence of elementary gates are encountered, their parameters, and their connectivity. To better understand how this encodes the quantum netlist, we need to understand the format.

Qubiter uses a small set of elementary gates: CNOT, CPHA (Controlled Phase), ROTY (Pauli-Y Rotation), and PHAS (Phase Gate that operates on all qubits). The format in the “engl.out” file is encoded as such:

```
CNOT <qubit1> <T|F> [<qubit2> <T|F>] [...] <target_qubit>
```

The *CNOT* first specifies a list of one or more controlling qubits and their assertion levels (T means true, implying a positive control node. F is a negative control node). The last term is the target qubit to be flipped if the input nodes and their assertion levels are valid.

```
ROTY <target_qubit> <angle>
```

The *ROTY* gate rotates the Y-component (using the Pauli-Y matrix denoted by  $\sigma_y(\theta)$ ) of a the target qubit by an arbitrary angle defined in degrees.

```
CPHA <qubit1> <T|F> [<qubit2> <T|F>] [...] <angle>
```

The *CPHA* (Controlled-Phase) gate has as arguments the gates involved in its control and their assertion levels, similar to the CNOT gate. However, the target qubit is *all* qubits which will have their phase adjusted by the angle as specified in degrees.

**PHAS** <angle>

The *PHAS* (Phase) gate has only one argument which specifies the angle in degrees to shift the phase of all qubits. There is no control condition. There are either one or no instances of this gate in the resulting synthesized quantum netlist.

Using our example “engl.out” file above, we can see the first three gates in the array are CPHA gates. The first CPHA gate uses qubit 0 as the control using a positive assertion level. When qubit 0 is  $|1\rangle$ , all qubits will have their phase shifted by  $25.07^\circ$  degrees.

## 7.4 Synthesis Results

Now that we’ve shown how we run Qubiter, define its inputs, and interpret its output we’re prepared to test some of our evolved unitary operators. We’ll do this with the best case individuals we found while optimizing our ES parameters in section 5.2 and are the “best” individuals we show in our eigenanalysis shown in figures 6.1 to 6.4.

As we noted in our introduction to this chapter, the deterministic nature of Qubiter resulted in largely the same gates and structures for all operators of the same number qubits. The arbitrary angles of the gates were different and some circuits concluded with a final PHAS gate whereas some did not. But apart from that, visually, the circuits were the same.

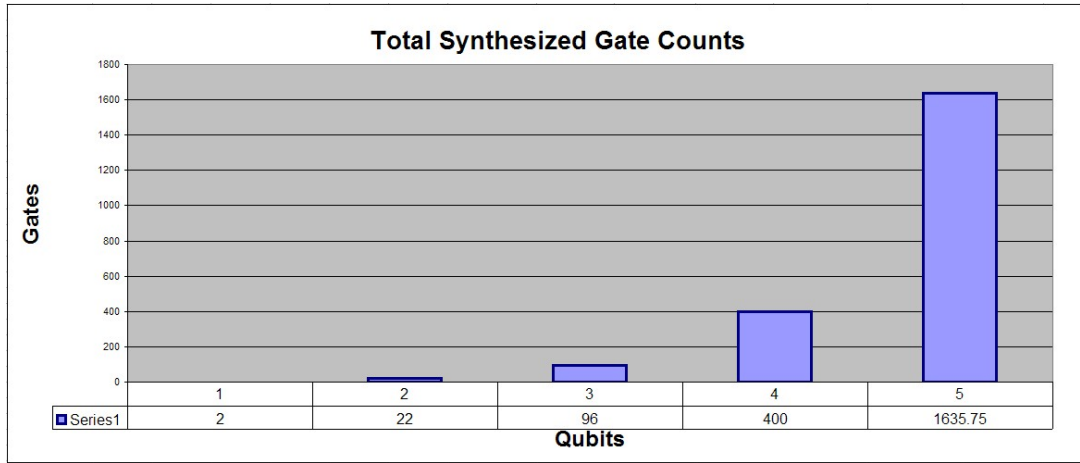


Figure 7.1: Average Number of Synthesized Gates

We wrote a small utility to read in the “engl.out” file from each run and convert it into a visual quantum “schematic”. Unfortunately, we run into a limitation of how many gates we can visualize. To give an idea of how many gates are used in synthesis we refer to figure 7.1 which shows the average total number quantum gates and figure 7.2 which shows the average breakdown of the different types of gates used.

There is roughly a  $4\times$  increase in gates as we increment the number of qubits which relates to the size of the matrix in terms of entries. For example, a 2 qubit operator has a  $4 \times 4$  matrix, a 3 qubit matrix is  $4\times$  larger at  $8 \times 8$ , a 4 qubit matrix is  $4\times$  larger at  $16 \times 16$ , and so on.

We can estimate the curve which determines the total number of gates by using the approximate equation,

$$T_G(q) = [10(2^{q-3})]^2 = 25[2^{2(q-2)}] \quad (7.4.1)$$

We show the accuracy of the above equation in the table 7.2 which shows

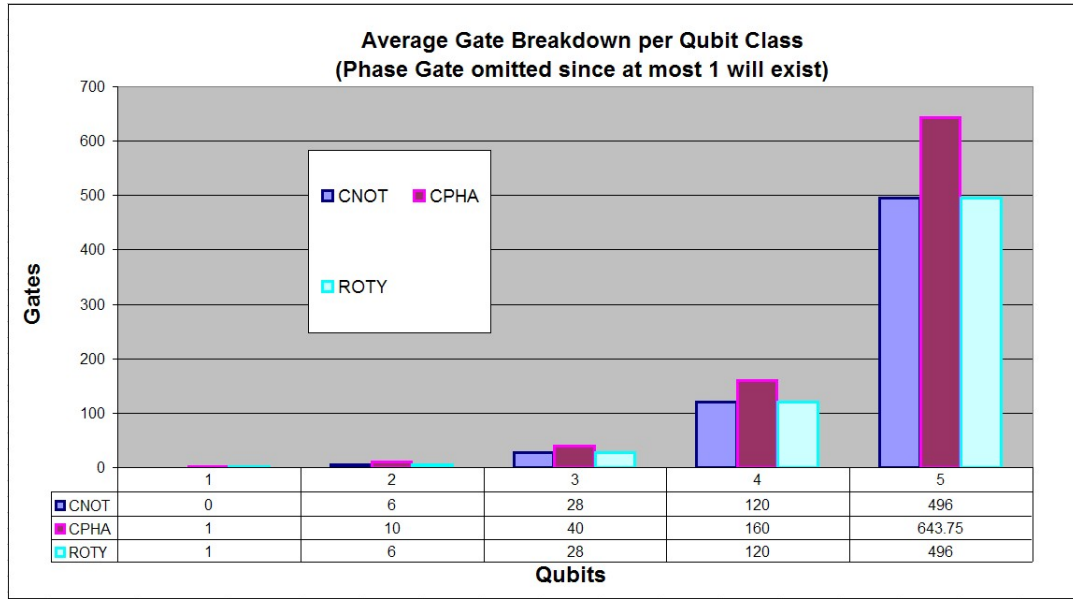


Figure 7.2: Breakdown of Gates used in Synthesis

Qubits	Average Total Gates	Predicted Total Gates
1	2	6
2	22	25
3	96	100
4	400	400
5	1600	1600

Table 7.2: Average total synthesized gates versus our predicted total gates.

that equation 7.4.1 does a very accurate job of predicting the total number of gates with some inaccuracy at single qubit.

We can display roughly the first 15 to 20 gates which means for 3 qubit and above, we cannot show the entire circuit. However, we now refer to figures 7.3 to 7.20. The CPHA gates are depicted by the “C” gates and the PHAS gate at the end (if one exists) is depicted as a “P” gate. CNOT use their traditional style of using an EXOR symbol ( $\oplus$ ) and ROTY gates are depicted as “Y” gates although we note these are arbitrary rotations about the Y-axis. Therefore,

the “C”, “P”, and “Y” gates have associated angle parameters not shown in the figures.

The single qubit showed the most variety due to their limited number of gates and either had one of two structures: (1) a single ROTY gate, or (2) a CPHA–ROTY–CPHA–PHAS structure. The second structure occurs when we used probability mode. This is perhaps because there was more freedom to adjust the phases of the qubits as long as the probability distribution was highly fit.

The 2 and 3 qubit synthesis results are prime examples that the structure is shared for the same number of qubits. These are small enough to visualize all the gates. All the 2 qubit schematics (figures 7.7 to 7.10) are visually the same. The same is true for the 3 qubit schematics (figures 7.11 to 7.14). Of course their angles will be different. In both 2 and 3 qubit netlists, a general PHAS gate is found at the end.

The 4 and 5 qubit synthesis results (figures 7.15 through 7.20) are so large that they are visually unappealing. An analysis of the circuits show they share the same structure as well as being terminated with a PHAS gate.

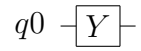


Figure 7.3: 1 qubit, complex mode, 70% target synthesized circuit.

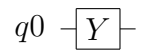


Figure 7.4: 1 qubit, complex mode, 99% target synthesized circuit.

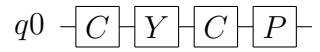


Figure 7.5: 1 qubit, probability mode, 70% target synthesized circuit.

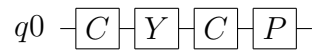


Figure 7.6: 1 qubit, probability mode, 99% target synthesized circuit.



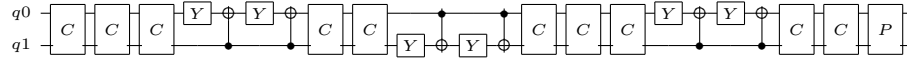


Figure 7.7: 2 qubit, complex mode, 70% target synthesized circuit.

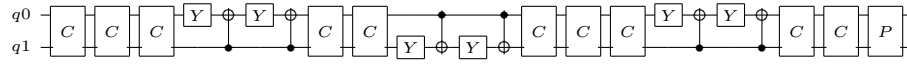


Figure 7.8: 5 qubit, complex mode, 99% target synthesized circuit.

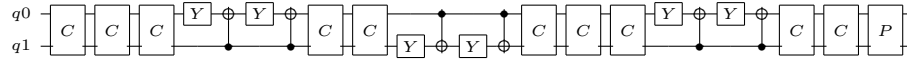


Figure 7.9: 2 qubit, probability mode, 70% target synthesized circuit.

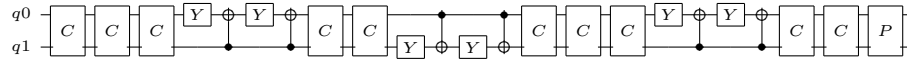


Figure 7.10: 2 qubit, probability mode, 99% target synthesized circuit.

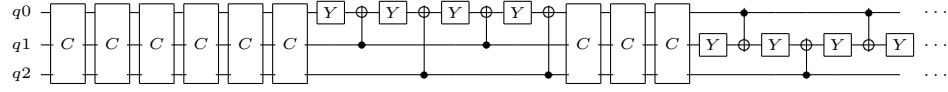


Figure 7.11: 3 qubit, complex mode, 70% target synthesized circuit.

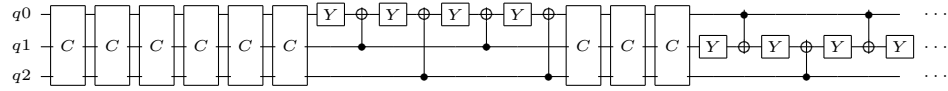


Figure 7.12: 3 qubit, complex mode, 99% target synthesized circuit.

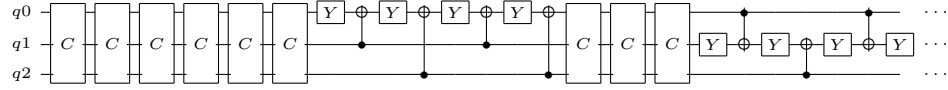


Figure 7.13: 3 qubit, probability mode, 70% target synthesized circuit.

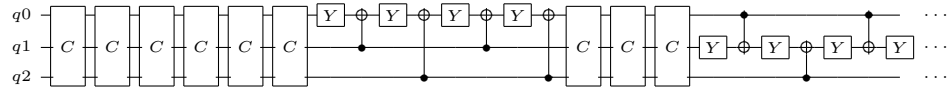


Figure 7.14: 3 qubit, probability mode, 99% target synthesized circuit.

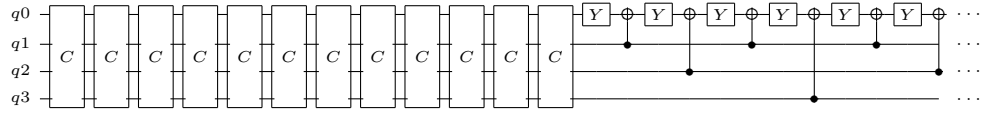


Figure 7.15: 4 qubit, complex mode, 70% target synthesized circuit.

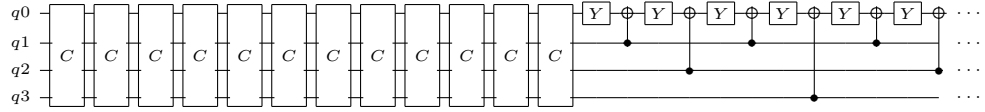


Figure 7.16: 4 qubit, complex mode, 99% target synthesized circuit.

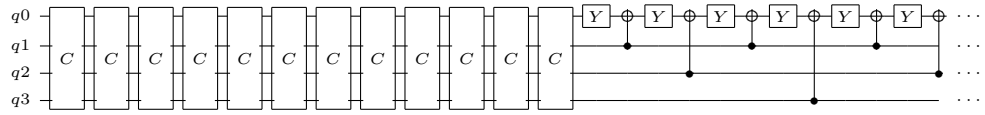


Figure 7.17: 4 qubit, probability mode, 70% target synthesized circuit.

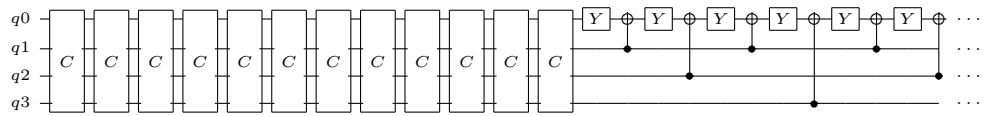


Figure 7.18: 4 qubit, probability mode, 99% target synthesized circuit.

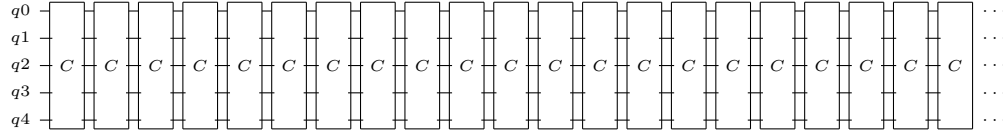


Figure 7.19: 5 qubit, complex mode, 70% target synthesized circuit.

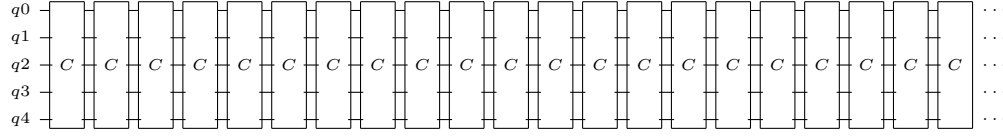


Figure 7.20: 5 qubit, probability mode, 70% target synthesized circuit.

The CPHA gates are indeed control gates that work on all the qubits. We don't show the controls we have concluded that the same controls and assertion levels are used for same-size operators. Again, confirming that a structural analysis won't lead to any valuable conclusions.

## 7.5 Input, Loaded, and Result Matrix Errors

Qubiter provides us with two output files that allow us to check the difference between the loaded and input unitary matrices and the difference between the loaded and synthesized matrices. The “chk.out” file is a reproduction of the input matrix after it has been loaded into Qubiter's internal data-structures. Qubiter may have its own set of dependencies on loading a structure, floating point accuracy, or potential bugs. The other file is the “err.out” file. This compares the loaded unitary matrix with the resultant unitary matrix after synthesis. Even though Qubiter uses many arbitrary angles, it still uses CNOT gates which are static, and each angle has a limit on its floating-point accuracy. We expect that large qubit simulations which have much higher error.

For example, in 5 qubit matrices resulted in 1600 gates and roughly 70% use arbitrary angle and we expect the errors quickly add up.

As our first test we compared our evolved unitary matrix with the matrix in the “chk.out” file. In particular, we looked at the average error per matrix element and we averaged this over the various simulations. Figure 7.21 shows the average per-element error. The 1 and 2 qubit errors were very small — on the order of  $10^{-9}$ . However, for 3 qubit the error shoots up to 0.094 per element. Although the input and loaded matrices had some difference, they were both tested unitary. An in-depth look at individual elements showed that some elements had very small errors while others had larger errors. It’s possible that Qubiter adjusted the values to fit its internal data structures. Oddly, the 4 qubit per element error was almost half of the 3 qubit error at 0.054. The 5 qubit was the worst at 0.179 per element.

The next test was to simply look at Qubiter’s “err.out” file to see how the loaded matrix compared to the synthesized matrix. Figure 7.22 shows a graph of the per element error between the loaded matrix and the synthesized matrix. Remarkably, we see a resemblance to figure 7.21 where 1 and 2 qubit error was extremely small — also on the order of  $10^{-9}$ . The 4 qubit error at 0.054 was about half of the 3 qubit error of 0.094, and the 5 qubit was the worst at 0.143 per element. This is more than just a resemblance because the numbers are almost identical. It seems that the error that was added in loading the unitary matrix is almost added again after synthesis.

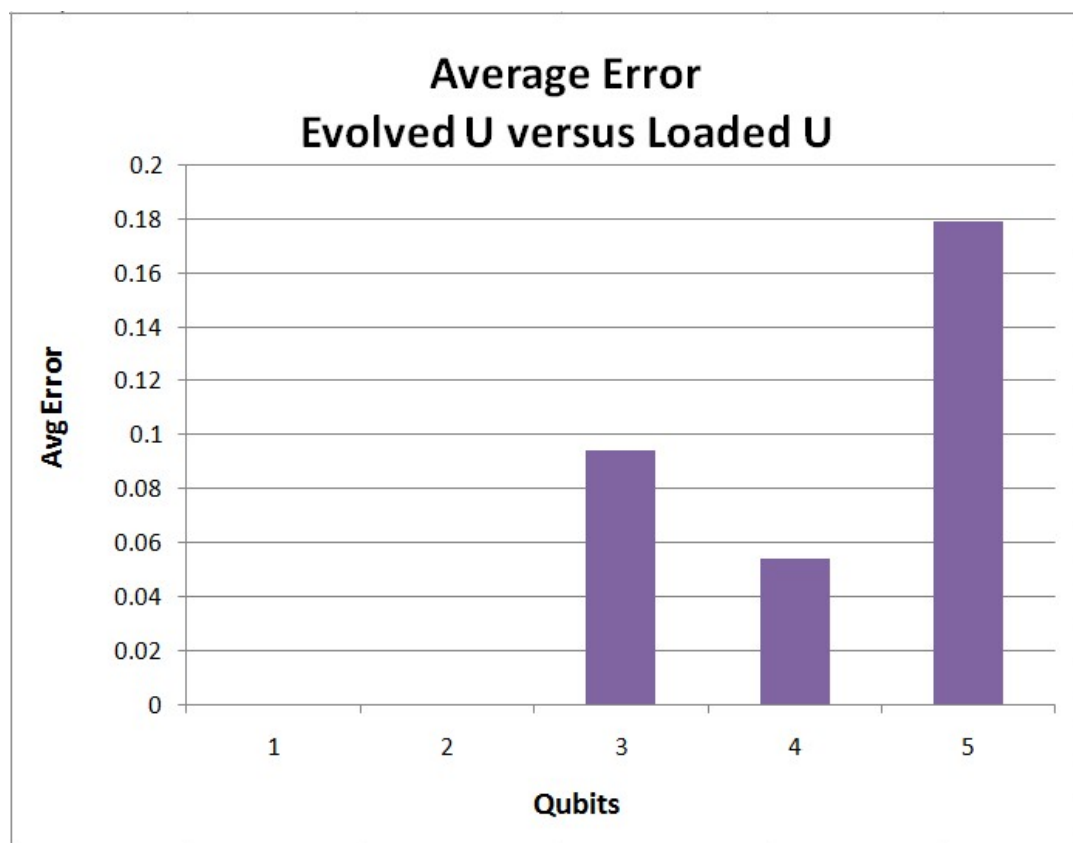


Figure 7.21: Average Error between the Evolved  $U$  and the Loaded  $U$

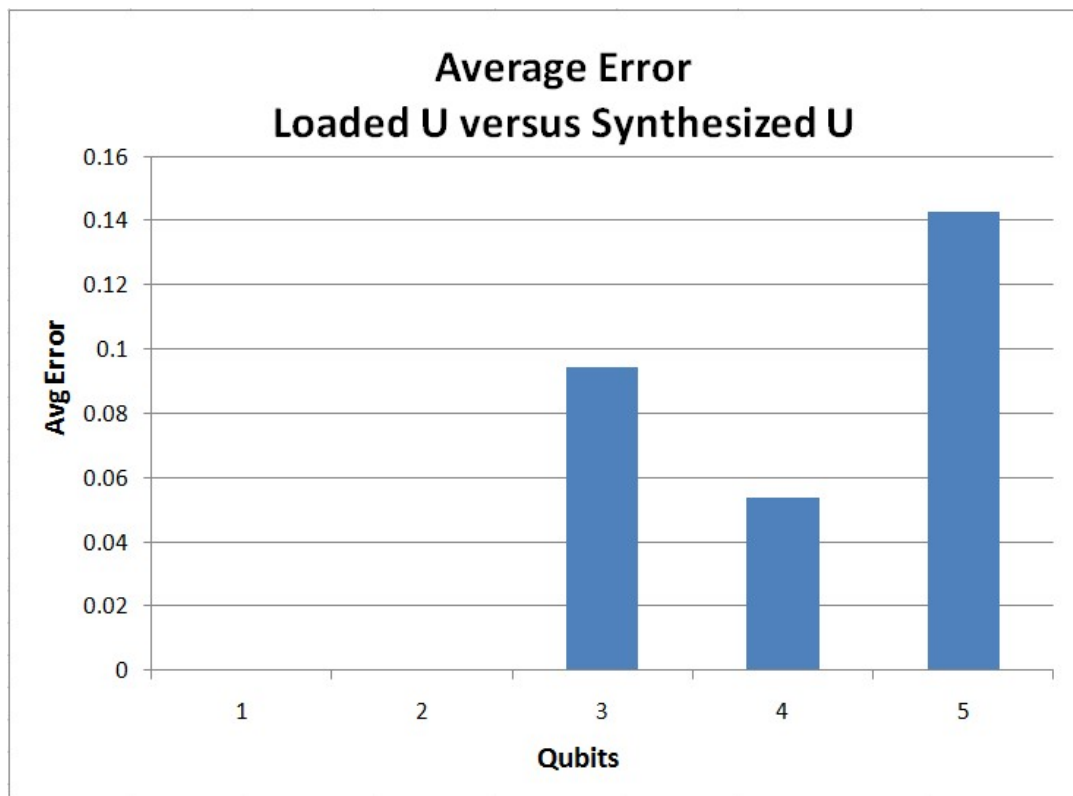


Figure 7.22: Average Error between the Loaded  $U$  and the Synthesized  $U$

## **7.6 Synthesis Runtimes, Limitations, and Memory Usage**

Since Qubiter was developed on a much older platform ( 100MHz and 32MB of memory), it made to be very efficient and run on very little memory. Although a limit to the number of qubits is not explicitly stated, Qubiter handled up to 5 qubit unitary matrices with no issues. Based on the studies we ran, Qubiter could likely handle potentially up to 7 or even 8 qubits.

The machine we ran Qubiter on was an iMac G3 running at 450MHz with 384MB of RAM. Run times for 1 to 3 qubit operators was on the order of a few seconds. For 5 qubit operators, it typically took a few minutes but some of the overhead also came from issues with the Mac–Classic emulator. Overall, runtime and memory were not an issue with Qubiter.

## **7.7 Synthesis Conclusions**

Putting aside the challenges we had getting Qubiter up and running given its development on an outdated platform which made porting to a more modern system next to impossible, we were fortunate to have a general quantum compiler freely available. It was also very helpful to receive support from the author to help us get up and running.

There were some unfortunate aspects about Qubiter that we did not like. The most significant negative aspect was that Qubiter did not do any optimizations or use more static quantum gates for its library. Instead, Qubiter used the same circuit structure for a given number of qubits and relied heavily on rotation and phase gates of arbitrary angles. While this is fine for the initial synthesis, it takes a considerable amount of work to map Hadamards, Y, X, and other gates



while still maintaining some level of fidelity. Because of Qubiter's limitations, we were unable to do any sort of quantum circuit structural comparisons or investigations.

Overall, we were excited to synthesize our evolved operators. However, we were dissatisfied that Qubiter's limitations did not allow further investigations. We hope that as newer quantum compilers emerge with more advanced techniques, features, and optimizations that this investigation can be tried again.

## Chapter 8

### Summary and Major Findings

Our research started out with a proposal to evolve quantum operators using an ES and a method of randomly generating unitary matrices (section 3.2) with the hope of being able to learn more about creating better quantum unitary operators through their eigenanalysis and synthesis. It was quite surprising that our ES was able to perform so well as indicated by the graph in figure 5.32 where valid solutions were found for even 5 qubit cases in under 30 generations for single problem instances. However, this did not come for free. It took a tremendous amount of effort in both implementing the ES as well as conducting numerous (as in tens-of-thousands) studies to tune the ES parameters as detailed in section 5.2. The result was, for the first time, an algorithm which designed quantum unitary operators represented using real-valued vectors and was published in [4]. Although the method has some limitations, it was used to evolve known single qubit quantum gates (section 5.3.2), the Hadamard gate which actually resulted in the pseudo-Hadamard instead (section 5.3.1). We also used it design a sub-circuit by evolving the 2 qubit oracle for Deutsch's Problem in section 5.3.3. We used the ES to quite successfully evolve quantum operators which represented the solution to problem instances of 3 and 4 node independent set problems (ISP) in section 5.4. We even went as far as to evolve

a general quantum operator for solving any 3-node ISP in section 5.5 albeit with partial success.

The eigenanalysis in section 6 was not as fruitful as we had hoped. The eigenvalue plots detailed in section 6.1 did not provide any real discernible patterns or information. This was later justified by the behavior of our random unitary matrix method. The same was also true of the eigenvectors. Although we showed how one could map the eigenvectors to the standard computational basis in section 6.3, we did not use this in our analysis. We were curious if pairs of eigenvalues close in value might suggest some degree of degeneracy. Therefore, it was like a silver lining when we found that spectral decomposition could be used to adjust eigenvalues and in our one application we found benefit by forcing degeneracy where it seemed suggested. This method, which was described in section 6.4, was discovered late in our research so it should be given more consideration in future work.

As we concluded in chapter 7 on our quantum logic synthesis results, we were hindered by the capabilities of our quantum compiler. At the time, it was the only mature quantum compiler publicly available. However, because of its lack of optimizations, heavy reliance on gates with arbitrary angles, and pre-determined circuit structure, very little useful data could be gleaned from the synthesis results. Also, the inability to port its code to linux or Windows prevented us from running on much more powerful machines and limited the number of parallel studies.

We had breakthrough in the middle of the research with the invention of our quantum algebraic method which was described in chapter 4 and published in [5]. It absorbed a tremendous amount of time to construct, derive its canonical

representation, as well as research its potential applications (e.g., the primary focus was on quantum logic synthesis where we attempt to form a circuit directly from the algebraic representation). Unfortunately, the majority of the material was not mature enough to be added to this dissertation. However, for the first time, a quantum circuit could be easily represented using a simple mathematical expression using well-known operations of matrix algebra and thus opens a whole new area of research using a wealth of previous knowledge. We used this method to prove known quantum circuit identities and as we mention in chapter 9, it has many more important potential applications.

## Chapter 9

### Future Work

#### 9.1 Further Evolving Unitary Matrices

One of the fundamental developments from this research which was detailed in chapter 5 was an EA based on an Evolution Strategy (ES) paired with a random unitary matrix method to evolve unitary operators that try to solve a particular problem. However, there are definitely improvements which can be made and further areas of research. We review these areas in this section.

#### Better Random Unitary Matrix Models

In section 3.2 we defined a method of generating a random unitary matrix given a set of angle parameters. In all respects, it is a very good method especially when considering how a random matrix should behave. However, the number one problem with this method was that for an  $N \times N$  matrix (where  $N = 2^q$  and  $q$  is the number of qubits), we needed  $(N - 1)N/2$   $\phi$  angles,  $(N - 1)N/2$   $\psi$  angles, and  $N - 1$   $\chi$  angles giving us a total of  $N^2 - 1$  angles. That means we need almost as many angles as we do elements in our matrix. This has an impact on our ES which has a  $\sigma$  parameter associated with every object parameters. Therefore, our 5-qubit individual is encoded with

1024 object parameters and another 1024 strategy parameters. Thus, having so many angle parameters means a very large search space for the ES. Also, our chosen random matrix method is computationally expensive to implement. It requires  $O(\frac{2^q(2^q+1)}{2})$  matrix multiplications to produce the final unitary matrix. At 5-qubits, that's 528  $32 \times 32$  complex double-precision matrix calculations.

Smaller and more simple random matrix model which are constructed to cater to the particular problem(s) we are trying solve should be investigated. It doesn't need to be ultimately configurable and as we saw with our chosen method, the anti-symmetric limitation prevented us from evolving many known elementary gates. Let's take for example the simple 2-qubit random model represented by the circuit in figure 9.1.

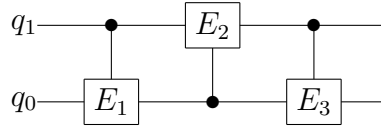


Figure 9.1: Simple Random Unitary Model Example.

The unitary gates  $E_1$ ,  $E_2$ , and  $E_3$  could be simple one-dimensional rotation gates of arbitrary angles like  $X(\theta)$ ,  $Y(\theta)$ , or adjusts phase like  $S(\theta)$ . Or, they could be more general Bloch-sphere rotations like we described in 2.2.2 using any combination of the rotation gates from the equations in 2.2.4. For example,  $E_1$  could be a combination of  $R_x(\theta_x) \times R_y(\theta_y)$  and have two angle parameters. Or, being more general, all three could use the general 1-qubit form defined in equation 2.2.5 and repeated below,

$$E_i = e^{-i\gamma_i} R_z(\theta_{zi}) R_y(\theta_{yi}) R_x(\theta_{xi})$$

However, this would create 12 angle parameters for the circuit in figure 9.1 which

is worse than what our chosen method required. However, maybe this model has qualities which lend itself to solving certain problems better? Perhaps it can be simplified to remove angle parameters?

There is potentially a lot of reward in investigating if we can intelligently construct the random unitary model using information about the problem(s) we intend to solve. For example, we typically know the input states and associated output states, and if we saw that a particular input qubit was always the same in the output state (for example,  $q_1 = |0\rangle$  before and after for all training pairs), then we can make an assumption that no gate needs to operate on qubit  $q_1$  and that it is probably only used as a control node. The circuit from figure 9.1 could be reduced down to the circuit in figure 9.2.

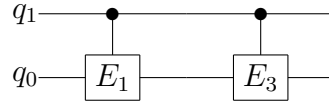


Figure 9.2: Reduced Simple Random Unitary Model.  $E_2$  appeared not to be needed as  $q_1$  always remained in the same state.

Having a potentially simpler model (especially one geared towards the problem at hand) could yield fewer angle parameters and — more importantly — operators that are more apt to find a higher fit solution and not hit limitations that would prevent it from being a better solution. Both of these traits would aid the ES greatly. It is also possible that custom random models could benefit in other areas such a generalized learning and general problem solving, both of which involve multiple input–output training pairs.

## Hybrid Algorithms and Optimizations

No one single algorithm is best for all cases, and sometimes it is better to have a mixture of algorithms to help cover a broader range of problems. This could also be true for our ES. Early in our research we investigated merging a greedy (and deterministic) algorithm into our ES. Its job was to select a small set of angles (like 2 or 3 of them) and run a greedy search to see what configuration of those angles resulted in a maximum fitness. The angle selection was purely random. What we found was that the greedy algorithm had a large impact to the ES runtime. In fact, it was better to not use the greedy algorithm since our ES alone could search faster. With that said, we shouldn't totally omit the possibility that a hybrid solution could be found.

In the area of ES optimizations, we also tried running deterministic routines that would give us a more fit initial population. And in all cases, the time spent on preparing the initial population was actually better spent on a totally random initial population but allowed to run for more generations. This could have been our weak attempt and more investigation into producing a healthier initial population should still be conducted.

However, one optimization that we did had a large impact on our ES performance: We limited the angles to be integer factors of  $\pi/12$ . We did this for the case where we were evolving an oracle to solve for Deutsch's problem in section 5.3.3. What normally would take hundreds of generations was now happening in only 26 generations. Part of the reason is that we knew the oracle was likely a permutation matrix<sup>1</sup> and this would help solve for those solutions by increasing the probability of producing strong 1's and 0's in the matrix.

---

<sup>1</sup>A *permutation matrix* is essentially an identity matrix but the columns have been arranged in some arbitrary order. An example is the Toffoli gate.



## Benefits of Recombination

While tuning the parameters for the ES in section 5.2, we investigated recombination. Earlier studies in our research indicated that adding recombination caused more disturbance than benefit. However, as the ES matured we attempted adding recombination a second time and actually found cases which outperformed pure mutation. However, it seemed to only be of value where our target percentage was lower. For the remainder of our studies we only considered pure mutation. Recombination is still a valuable aspect of ES although it is not the primary operation, and could use more investigation.

## Generalized Learning Behavior

One of the early topics we tried tackling in our research was that of a form of quantum machine learning. The idea was simple, have the ES evolve (or “train”) a unitary operator on a sub-set of input-output states and then see how the operator would perform when given an input it was not trained with. As a lofty example, we could train the unitary operator to solve for most of the solutions to any  $N$  node ISP, then test its solutions for input configurations we never trained it with. The first problem we ran into was that multiple input-output training was very difficult in terms of convergence. We saw this in section 5.5 with the 3 node general ISP. Earlier tests also confirmed that our evolved operators were not generalizing well. Part of the difficulty was our random unitary matrix scheme. With a different model, perhaps generalized training would be more interesting. However, we still feel this area warrants further investigation.

## 9.2 Applications of our Quantum

### Logic Algebra

One of the most important areas of future research deals with the quantum logic algebra we defined in chapter 4. Although some basic applications were described such as showing circuit identities and extensions to multi-valued logic, there are many more applications.

### Quantum Computing Simulation

In the area of quantum computing simulation, not only can our quantum logic algebra be used to represent a quantum circuit in a simple mathematical expression using basic matrix operations which has a wealth of research behind it, but it could potentially be used to reduce equations, thereby reducing the number of matrix calculations and thus increase simulation and time and perhaps reduced memory overhead.

As an example we look at the circuit in figure 9.3 which is a well known circuit original investigated by Barenco et al in [33] which shows a general 3-qubit double-controlled unitary gate (Barenco called this gate:  $\Lambda_2(U)$ ) being decomposed into 2-qubit operations.

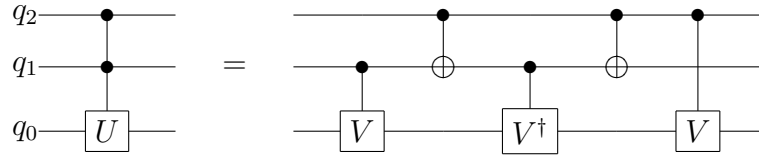


Figure 9.3: Barenco et al's decomposition of  $\Lambda_2(U)$  where  $V^2 = U$ .

In our algebra, the right side circuit has – at first – a lengthy description. The left side is simple. We use the notation that Kronecker products are implied

(that is,  $AB = A \otimes B$ ).

The left side equation is simply

$$S = D_0II + D_1D_0I + D_1D_1U \quad (9.2.1)$$

The right side has 5 slices from left-to-right

$$S_1 = ID_0I + ID_1V \quad (9.2.2)$$

$$S_2 = D_0II + D_1XI \quad (9.2.3)$$

$$S_3 = ID_0I + ID_1V^\dagger \quad (9.2.4)$$

$$S_4 = D_0II + D_1XI \quad (9.2.5)$$

$$S_5 = D_0II + D_1IV \quad (9.2.6)$$

When we write out  $S = S_5 \times S_4 \times S_3 \times S_2 \times S_1$ , the expansion can look painful and it is not apparent that this whole expression is equivalent to equation 9.2.1. However, while expanding we see a natural simplification take place. Namely, anytime we have a term where a  $D_0$  is multiplied by a  $D_1$  (or  $D_1$  multiplied by  $D_0$ ), then the term is dropped because  $D_0 \times D_1 = D_1 \times D_0 = 0$ . We wrote a small perl script which was able to symbolically multiply the right-side slice equations above. The result is shown below,

$$S' = D_0D_0I + D_0D_1(V^\dagger V) + D_1D_1(VV) + D_1D_0(VV^\dagger) \quad (9.2.7)$$

Since  $V$  is unitary,  $V^\dagger V = VV^\dagger = I$ . Also, since  $V^2 = U$ , the above equation simplifies to equation 9.2.1!

So, we discovered two things in this exercise: (1) we were able to prove

using our algebraic method that Barenco's decomposition is valid, and, (2) an otherwise complex circuit consisting of multiple slices can quickly be reduced into only a few terms which required far less matrix multiplies and additions than the original right-side equation. Therefore, future research should involve using our quantum logic algorithm to benefit quantum computing simulation.

## Quantum Logic Synthesis

The power of our quantum logic algebra is its ability to quickly transform a quantum circuit into an algebraic expression. With this expression we are able to use the wealth of knowledge to manipulate it with known mathematical rules, laws, and identities, and perhaps pave the way to forming new ones. However, one of the areas we did not cover although considerable time was spent in its initial investigation, was that of *forming a quantum circuit based on a quantum algebraic expression*.

This is a very difficult task and as of this writing, has no known method apart from considerable manipulation of the expression which might involve non-intuitive expansions.

Why this would be valuable should be quite obvious. If we were given a quantum circuit which was potentially not optimal, then we could transform it to our quantum algebra, simplify it through expanding and collapsing terms (or other methods), then re-generate the circuit based on our new simplified algebraic expression. This is much easier said than done.

Where things get complicated is when we have multiple slices. Not all examples will collapse to a single slice expression like they did in figure 9.3. Let's take a simple example and say after simplifying my circuit, we ended up with

the following algebraic expression (note that matrix multiplies are explicitly written use the  $\cdot$  symbol rather than the  $\times$  symbol):

$$U = D_0 D_0 + (D_0 \cdot X) D_1 + D_1 (X \cdot D_0) + (D_1 \cdot X) (X \cdot D_1) \quad (9.2.8)$$

By looking at equation 9.2.8, it is not clear what circuit this expression implements. However, with some simple factoring, we can start to see known algebraic structures appear,

Factor out  $D_0 I$ .

$$U = (D_0 I) \cdot (ID_0 + XD_1) + D_1 (X \cdot D_0) + (D_1 \cdot X) (X \cdot D_1)$$

Factor out  $D_1 X$ .

$$U = (D_0 I) \cdot (ID_0 + XD_1) + (D_1 X) \cdot (ID_0 + XD_1)$$

Factor out common term  $ID_0 + XD_1$ .

$$U = (D_0 I + D_1 X) \cdot (ID_0 + XD_1)$$

This leaves us with two terms which are known circuit structures multiplied together. If we let  $S_1 = ID_0 + XD_1$  and  $S_2 = D_0 I + D_1 X$ , then we recognize these circuit structures as CNOT gates and thus can draw the quantum circuit, shown in figure 9.4.

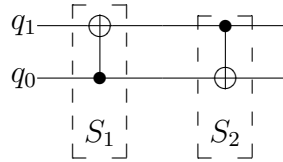


Figure 9.4: Reconstructed circuit from a quantum algebraic expression.

This example shows only single-controlled gates, the complexity of identi-

fying gates with many controls and how to factor their terms out is obviously much higher.

We can take the above simple concept one more step by developing rules to look for as we're manipulating the expressions. For example, the function,  $F$ , before a CNOT operation in a circuit is shown in figure 9.5.

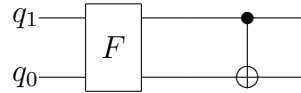


Figure 9.5: General circuit showing  $F$  followed by a CNOT.

The expression for such a circuit will have the form

$$U = (D_0I + D_1X) \cdot F = (D_0I) \cdot F + (D_1X) \cdot F \quad (9.2.9)$$

So, when we are manipulating our equations and we see a sub-expression like

$$(D_0I) \cdot F + (D_1X) \cdot F$$

then we can recognize the factoring and see this is the two necessary terms ( $D_0I$  and  $D_1X$ ) to form a CNOT followed by some expression,  $F$ . The idea here is that we are trying to extract out the circuit starting from the end (or right-most gate) to the beginning.

As an extension, we can look at the circuit in figure 9.6 which separates the function of the upper qubit from the lower qubit. There are instances this might help decompose the expression into a circuit.

The derived expression is

$$U = (D_0I + D_1X) \cdot (f_1f_2) = (D_0I) \cdot f_1 + (D_1X) \cdot f_2 \quad (9.2.10)$$

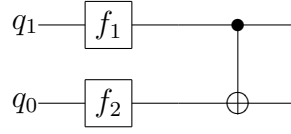


Figure 9.6: General circuit where the function of the upper circuit is separated by from the function of the lower qubit, followed by a CNOT.

Therefore, if during our equation manipulation we see a sub-expression which has the form:

$$(D_0I) \cdot f_1 + (D_1X) \cdot f_2$$

then we can see that it has the circuit form in figure 9.6.

These examples showed how a couple of forms could reveal a CNOT gate but these forms could be made even more general to expose any general gate. Also, we looked at right-most but nothing prevents us from looking for factoring to expose left-hand side circuits (e.g.,  $U = F \cdot (D_0I + D_1X)$ ).

Obviously an intimate understanding of manipulating equations which are a mixture of addition/subtraction, multiplication, and Kronecker products of matrices must be attained. As we experimented with these equations, we also investigated specialized *division* operations although the work is not stated here since it wasn't yet conclusive. However, division exposed some very interesting properties and is definitely a potentially fruitful area of research.

One can envision a quantum logic synthesizer that does a first-pass mapping of the input description of the quantum computer (e.g., a unitary matrix) into a quantum circuit. Then the circuit, either in whole or in part, is transformed into our algebraic notation. A series of algorithms using known identities and rules then work on the circuit to simplify the expression. Next, another set of algorithms manipulate the equations into a sequence of multiplied terms where

each term is a known circuit structure.

Overall, applying our quantum logic algebra to synthesis is clearly one of the areas that needs further research and would greatly increase the usefulness and power of the method.

## **Exploring Multi-Value Logic**

We briefly mention how our quantum logic algebra can encompass multi-value quantum logic and give a simple example in section 4.8. However, this was the only treatment we gave to multi-valued logic and clearly deserves more investigation. It is quite possible that our quantum logic algebra has an even more important role in multi-valued logic.

### **9.3 Expanding The Eigenanalysis**

The eigenanalysis we did in section 6 was largely focused on the eigenvalues. Although it was interesting to visualize the eigenvalues on the unit circle, their comparisons didn't provide us much insight. Individuals trained with a probabilistic target had more eigenvalue variation than complex target which was an intuitive finding. Most individuals tended to have similar eigenvalues. In the cases where we changed the problem instance and looked at how the eigenvalues were altered, there wasn't a clear reason — some eigenvalues didn't change and some did but not for an obvious reason. It was interesting to see how the eigenvalue compared for single versus multi-solution ISP instances. In some cases an averaging of eigenvalues was seen, but not in others. Overall, we pulled very little useful information from the plots. It is possible that more useful information can be extracted from the eigenvectors themselves. We did have some



highlights while examining the eigenanalysis but we feel a lot is still left on the table to be investigated. In this section, we highlight a few of those areas.

## Eigenvector Mapping and Eigenvalue Adjusting

In section 6.3, we discovered a way to map the eigenvalues and eigenvectors from the spectral decomposition format into the common outer-product format using equation 6.3.12 repeated here:

$$v_{ij} = \sum_{k=1}^N \lambda_k e_{ki} e_{kj}$$

The basis vectors were mapped from mostly non-intuitive eigenvectors to the standard computational basis (e.g.,  $|00\rangle$ ,  $|01\rangle$ , etc.). We showed how this transformation could be applied to the NOT gate. For example, by looking at the eigenvalues and eigenvectors, it wasn't clear if it was a NOT gate. However, this is as far as we went. With most mathematical transformations, it is possible that this method lends itself to simplifying or facilitating the solutions to other problems and warrants further research. For example, this might provide a method to adjusting eigenvectors for a given operator. Or, given a unitary operator, we could see how standard input vectors are mapped to output vectors (as a function standard basis vectors) which might tell us more about the function of the operator, highlighting dependencies or even entanglement.

The other highlight which was probably the most useful outcome of the eigenanalysis was described in section 6.4 which gave us a method to adjust the eigenvalues of a given operator without destroying its unitary property. But again, we only showed a few examples of its application and it should be investigated further.

Let's say we had a given unitary operator  $U$  which we knew was a good solution but needed enhancement. Section 6.4 allows us to adjust the eigenvalues to form a new unitary operator,  $U'$ , using:

$$\mathbf{U}'(\theta_1, \dots, \theta_n) = \sum_{i=1}^n e^{i\theta_i} |v_i\rangle \langle v_i|$$

What we never investigated is how to alter the eigenvalues in the above equation to enhance the fitness over the original  $U$ . This is an area for exploration. For example, one could use another ES that explored various values of  $e^{i\theta_i}$ . An ES lends itself well since we have  $N = 2^q$  eigenvalues which is a relatively small number (this is the square-root of the number of angle parameters our main ES had to deal with) and our individual could encode the  $N$  angles as a real-valued vector:  $\langle \theta_1, \theta_2, \dots, \theta_N \rangle$ . Or, even a deterministic greedy or gradient-descent type search could be used to hone the angles by small deltas, reconstruct the new  $U'$ , and determine its fitness and select the new angle configuration with the best fitness.

## Eigenvector Analysis

We had mentioned some eigenvector analysis techniques in section 3.3 but they were largely omitted because of the unitary nature of our operators. There really wasn't a concept of a dominant eigenvector because our eigenvalues all had unity modulus. Also, the eigenvectors formed the orthonormal basis vectors for our operator, which meant the eigenvector was a pure state of the operator which took away a lot of the mystery of the eigenvectors (it was this fact that made it possible to map the eigenvectors to the standard computational basis in section 6.3). Eigenvectors are harder to compare and view. However, there are

still some remaining analysis methods we did not try. Methods of comparing the eigenvectors of unitary operators could be investigated. Also, just as we developed a method to adjust eigenvalues and see a benefit, maybe a similar method can be done for eigenvectors. For example, possibly as long as the eigenvectors form an orthonormal basis, the spectral decomposition form can be used and eigenvector adjustments could be made which benefit the fitness of a unitary operator.

## 9.4 Better Logic Synthesis

All of the quantum logic synthesis we did was with Qubiter and the results are summed up in chapter 7. We noted in our conclusions in that chapter that Qubiter had some severe limitations that inhibited further investigations; Namely, no optimizations and the excessive reliance on gates with arbitrary angles. So, this begs the continuation of this research as newer and better quantum compilers are developed. For example, it would be interesting to see how Dr. Lukac’s GA compiler[40] described in section 3.1.1 — which implements configurable quantum gate library and various optimizations — would deal with our evolved unitary operators.

As part of the future work that can be done in logic synthesis, we largely re-state a lot of what we had planned to do but ran into limitations with the synthesizer. For example, if we had two operators related in some way, we could perform the following:

- Compare gate usage and overall count counts.
- Identify common sub-circuits and their frequency.

And more advanced techniques could be investigated which might involve how well the operators map to certain technologies such as NMR or ion trap.

It would also be interesting to see how operators of varying fitness compare in terms of synthesis results. For example, would more fit operators require more or less gates?

Since our ES creates random unitary matrices based on angles, there is an intuitive feeling that those arbitrary angles will need to be mapped to static gates in some way. Obviously very obscure angles which have many digits of accuracy will have some error associated with them, so, perhaps it would be interesting to see how (and if) an optimized operator which uses more integral-friendly angles would result in a cleaner synthesis result.

## References

- [1] F. Vatan and C. Williams. Optimal quantum circuits for general two-qubit gates. *Physical Review A*, 69(3):32315, 2004.
- [2] VV Shende, SS Bullock, and IL Markov. A Practical Top-down Approach to Quantum Circuit Synthesis. *Arxiv preprint quant-ph/0406176*, 2004.
- [3] R.R. Tucci. A Rudimentary Quantum Compiler (2cnd Ed.). *Arxiv preprint quant-ph/9902062*, 1999.
- [4] Steven R. Hutsell and Garrison W. Greenwood. Applying evolutionary techniques to quantum computing problems. *CEC 2007. IEEE Congress on Evolutionary Computation*, pages 4081–4085, 2007.
- [5] Steven R. Hutsell and Garry W. Greenwood. Efficient Algebraic Representation of Quantum Circuits. *the Journal of Discrete Mathematical Sciences and Cryptography*, Accepted for Publication, February 2009.
- [6] K. Życzkowski and M. Kuś. Random unitary matrices. In *Phys. A: Math Gen.*, volume 27, pages 4235–4245, 1994.
- [7] Lukac M. Quantum Logic Synthesis and Inductive Learning, Ph.D. dissertation. *Portland State University*, 2009.
- [8] P.M. Dirac. *Principles of quantum mechanics*. Oxford, 1948.

- [9] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, October 2000.
- [10] C. H. Bennett and P. W. Shor. Quantum information theory. *IEEE Transactions on Information Theory*, 44(6):2724–2742, 1998.
- [11] C.H. Bennett and Brassard G. Quantum cryptography: Public-key distribution and coin tossing. In *Proceedings of the IEEE International Conference on Computers, Systems and Signal Processing, Bangalore, India*, page 175, Los Alamitos, Calif., 1984. IEEE Computer Society Press.
- [12] T. Menneer. *Quantum Artificial Neural Networks*. PhD thesis, The University of Exeter, UK, May 1998.
- [13] J. Eisert, M. Wilkens, and M. Lewenstein. Quantum games and quantum strategies. *Physical Review Letters*, 83:3077, 1999.
- [14] E.W. Piotrowski and J. Sladkowski. Quantum-like approach to financial risk: quantum anthropic principle. *Acta Physica Polonica*, 32:3873, 2001.
- [15] S. Wake. *Quantum investing*. Texere Publishing, London, 2002.
- [16] S. Habib, K. Jacobs, and H. Mabuchi. Quantum feedback control - how can we control quantum systems without disturbing them? *Los Alamos Science*, 27:126, 2002.
- [17] A. C. Doherty, S. Habib, K. Jacobs, H. Mabuchi, and S. M. Tan. Quantum feedback control and classical control theory. *Physical Review A*, 62:012105, 2000.

- [18] P. Benioff. Quantum mechanical hamiltonian models of turing machines. *J. Stat. Phys.*, 29:515, 1982.
- [19] R. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21, No. 6/7:467–488, 1982.
- [20] R. Feynman. Quantum mechanical computers. *Foundations of Physics*, 16:507–531, 1982.
- [21] D. Deutsch. Quantum theory, the church–turing principle and the universal quantum computer. *Proc. Roy. Soc. Lond. Ser. A*, 400:96–117, 1985.
- [22] D. Deutsch. Quantum computational networks. *Roy. Soc. Lond. Ser. A*, 425:73–90, 1989.
- [23] D. Deutsch and R. Jozsa. Rapid solution of problems by quantum computation. *Proc. Roy. Soc. Lond. Ser. A*, 439:553–558, 1992.
- [24] A. Berthiaume and G. Brassard. Oracle quantum computing. In *Proc. Workshop on Physics and Computation*, pages 195–199. IEEE Computer Society Press, 1992.
- [25] D. Simon. On the power of quantum computation. In *Proc. 35th Ann. Symp. on Foundations of Computer Science*, pages 116–123. IEEE Computer Society Press, 1994.
- [26] E. Bernstein and U. Vazirani. Quantum complexity theory. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 11–20, 1993.

- [27] P. W. Shor. Algorithms for quantum computation: discrete log and factoring. In *Proceedings of the 35th Annual Symposium on the Foundations of Computer Science*, page 124. IEEE Computer Society Press, 1994.
- [28] L.K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings, 28th Annual ACM Symposium on the Theory of Computing*, page 212, 1996.
- [29] R.P. Feynman, R.B. Leighton, and M. Sands. The Feynman Lectures on Physics, Vol. III. *Quantum Mechanics (Addison-Wesley, Reading, 1965)*, 1965.
- [30] JA Jones, RH Hansen, and M. Mosca. Quantum logic gates and nuclear magnetic resonance pulse sequences. *Journal of Magnetic Resonance*, 135(2):353–360, 1998.
- [31] K. Dorai and A. Kumar. Implementation of a Deutsch-like quantum algorithm utilizing entanglement at the two-qubit level on an NMR quantum-information processor. *Physical Review A*, 63(3):34101, 2001.
- [32] J. Gruska. *Quantum computing*. McGraw-Hill Book Co Ltd, 1999.
- [33] A. Barenco, C.H. Bennett, R. Cleve, D.P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J.A. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *Physical Review A*, 52(5):3457–3467, 1995.
- [34] W.K. Wootters and W.H. Zurek. A single quantum cannot be cloned. *Nature*, 299:802, 1982.
- [35] R. Jozsa. Illustrating the concept of quantum information. *Arxiv preprint quant-ph/0305114*, 2003.



- [36] Dan C. Marinescu and Gabriela M. Marinescu. *Approaching quantum computing*. Pearson Prentice Hall, 2004.
- [37] L.M.K. Vandersypen, M. Steffen, G. Breyta, C.S. Yannoni, M.H. Sherwood, and I.L. Chuang. Experimental realization of Shor’s quantum factoring algorithm using nuclear magnetic resonance. *Arxiv preprint quant-ph/0112176*, 2001.
- [38] M. Boyer, G. Brassard, P. Høyer, and A. Tapp. Tight Bounds on Quantum Searching. *Fortschritte der Physik*, 46(4-5):493–505, 1999.
- [39] H.G. Beyer and H.P. Schwefel. Evolution strategies – A comprehensive introduction. *Natural Computing*, 1(1):3–52, 2002.
- [40] M. Lukac, M. Perkowski, H. Goi, M. Pivtoraiko, C. H. Yu, K. Chung, H. Jeech, B. Kim, and Y Kim. Evolutionary approach to quantum and reversible circuits synthesis. *Artif. Intell.*, 20 Issue 3-4:361–417, 2003.
- [41] P. Massey, J.A. Clark, and S. Stepney. Human-competitive evolution of quantum computing artefacts by genetic programming. *Evol Comput*, 14(1):21–40, 2006.
- [42] P. Massey, J. Clark, and S. Stepney. Evolution of a human-competitive quantum fourier transform algorithm using genetic programming. In *Proc. GECCO*, pages 1657–1663, 2005.
- [43] M. Lukac and M. Perkowski. Evolving quantum circuits using genetic algorithm. In *EH ’02: Proceedings of the 2002 NASA/DoD Conference on Evolvable Hardware (EH’02)*, pages 177–185, 2002.

- [44] YZ Ge, L.T. Watson, and EG Collins. Genetic algorithms for optimization on a quantum computer. *Unconventional Models of Computation*, pages 218–227, 1998.
- [45] BIP Rubinstein. Evolving quantum circuits using genetic programming. *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, 1, 2001.
- [46] T. Yabuki and H. Iba. Genetic algorithms for quantum circuit design-evolving a simpler teleportation circuit. In *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, pages 421–425. Morgan Kauffman Publishers, 2000.
- [47] T. Yabuki and H. Iba. Genetic algorithms for quantum circuit design - evolving a simpler teleportation circuit. In Darrell Whitley, editor, *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, pages 425–430, Las Vegas, Nevada, USA, 8 2000.
- [48] J.T. Alander. Optimization by quantum computing and genetic algorithms. *Quantum Computing, Physical Realizations, Report Series, Nro TKK-F-A810*, pages 73–82, 2001.
- [49] K.H. Han and J.H. Kim. Genetic quantum algorithm and its application to combinatorial optimization problem. *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, 2, 2000.
- [50] M.H.A. Khan and M. Perkowski. Genetic algorithm based synthesis of multi-output ternary functions using quantum cascade of generalized

- ternary gates. *Evolutionary Computation, 2004. CEC2004. Congress on*, 2, 2004.
- [51] M. Lukac, M. Pivtoraiko, A. Mishchenko, and M. Perkowski. Automated Synthesis of Generalized Reversible Cascades using Genetic Algorithms. *5th International Workshop on Boolean Problems*, pages 33–45, 2002.
  - [52] GX Zhang, WD Jin, and N. Li. An improved quantum genetic algorithm and its application. *Lecture Notes in Artificial Intelligence*, 2639:449–452, 2003.
  - [53] C. Ruican, M. Udrescu, L. Prodan, and M. Vladutiu. A Genetic Algorithm Framework Applied to Quantum Circuit Synthesis. *Intelligence (SCI)*, 129:419–429, 2008.
  - [54] L. Spector, H. Barnum, H. J. Bernstein, and N. Swamy. Quantum computing applications of genetic programming. *Advances in Genetic Programming*, 3:135–160, 1999.
  - [55] JR Koza, SH Al-Sakran, and LW Jones. Cross-Domain Features of Runs of Genetic Programming Used to Evolve Designs for Analog Circuits, Optical Lens Systems, Controllers, Antennas, Mechanical Systems, and Quantum Computing Circuits. *Evolvable Hardware, 2005. Proceedings. 2005 NASA/DoD Conference on*, pages 205–214, 2005.
  - [56] L. Spector. *Automatic Quantum Computer Programming: A Genetic Programming Approach*. Kluwer Academic Publishers, 2004.
  - [57] A. Leier and W. Banzhaf. Evolving Hogg’s quantum algorithm using linear-tree GP. *Lecture Notes in Computer Science*, pages 390–400, 2003.

- [58] M. Mitchell. *An introduction to genetic algorithms*. Bradford Books, 1996.
- [59] D.E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1989.
- [60] F. J. Dyson. Statistical theory of the energy levels of complex systems. I. *J. Math. Phys.*, 3(1):140–156, January/February 1962.
- [61] MA Nielsen and J. Kempe. Separable States Are More Disordered Globally than Locally. *Physical Review Letters*, 86(22):5184–5187, 2001.
- [62] H. Chen. Quantum entanglement without eigenvalue spectra. *Arxiv preprint quant-ph/0108093*, 2001.
- [63] V.V. Shende, S.S. Bullock, and I.L. Markov. Synthesis of Quantum Logic Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(6):1000–1010, 2006.
- [64] G. Cybenko. Reducing quantum computations to elementary unitary operations. *Computing in Science & Engineering [see also IEEE Computational Science and Engineering]*, 3(2):27–32, 2001.
- [65] J.J. Vartiainen, M. Möttönen, and M.M. Salomaa. Efficient Decomposition of Quantum Gates. *Physical Review Letters*, 92(17):177902, 2004.
- [66] D. Deutsch, A. Barenco, and A. Ekert. Universality in Quantum Computation. *Arxiv preprint quant-ph/9505018*, 1995.
- [67] E. Knill. Approximation by quantum circuits. *arXiv.org:quant-ph/9508006*, 1995.

- [68] M. Mottonen and J. J. Vartiainen. Decompositions of general quantum gates. *arXiv.org:quant-ph/0504100*, 2006.
- [69] C. C. Paige and M. Wei. History and generality of the cs decomposition. *Linear Algebra and Applications*, 208:303–326, 1994.
- [70] C. P. Williams and A. Gray. Automated design of quantum circuits. In *Lecture Notes in Computer Science*, volume 1509, pages 113–125. Springer-Verlag, 1998.
- [71] M. Perkowski, A. Al-Rabadi, and P. Kerntopf. Multiple-valued quantum logic synthesis. In *Proc. of 2002 International Symposium on New Paradigm VLSI Computing*, pages 41–47, 2002.
- [72] J. A. Jones, R. H. Hansen, and M. Mosca. Quantum logic gates and nuclear magnetic resonance pulse sequences. *Journal of Magnetic Resonance*, 135:353–360, 1998.
- [73] J. Behrems and J. Garcia. Computers and intractability: A guide to the theory of NP-completeness, 1979.
- [74] A.Y. Kitaev, A. Shen, and M.N. Vyalyi. *Classical and Quantum Computation*. American Mathematical Society, 2002.
- [75] R. Cleve, A. Ekert, L. Henderson, C. Macchiavello, and M. Mosca. On quantum algorithms. *Complexity*, 4:33, 1998.
- [76] A. Galindo and M.A. Martin-Delgado. A family of grover’s quantum searching algorithms. *arXiv.org:quant-ph/0009086*, 2000.

- [77] N. Shenvi, J. Kempe, and K. Birgitta Whaley. A quantum random walk search algorithm. *Physical Review A*, 67:052307, 2003.
- [78] A. Ambainis, J. Kempe, and A. Rivosh. Coins make quantum walks faster. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1099–1108, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.
- [79] M. Mosca and A. Ekert. The Hidden Subgroup Problem and Eigenvalue Estimation on a Quantum Computer. *Quantum Computing and Quantum Communications: First Nasa International Conference, Qcqs' 98, Palm Springs, California, Usa, February 17-20, 1998: Selected Papers*, 1999.
- [80] D.S. Abrams and S. Lloyd. Quantum Algorithm Providing Exponential Speed Increase for Finding Eigenvalues and Eigenvectors. *Physical Review Letters*, 83(24):5162–5165, 1999.
- [81] P. Jaksch and A. Papageorgiou. Eigenvector approximation leading to exponential speedup of quantum eigenvalue calculation. *arXiv.org:quant-ph/0308016*, 2003.

## Appendix A

### Compiling and Running the ES

The source code for the ES is written in C++ and composed of only four files. The main file (`es.cpp`) and three accompanying header files (`random.h`, `unitary.h`, `matrix.h`, and `parse_args.h`). A description of these files is as follows:

- *es.cpp* is the main C++ file which contains the `main()` function and the primary ES code.
- *matrix.h* is a basic header file which contains all of the basic matrix class and accompanying functions.
- *unitary.h* contains the function which implements the method described in section 3.2 to generate random unitary matrices.
- *random.h* is a header file which defines standard random functions useful for evolutionary algorithms in general.
- *parse\_args.h* is a header file which handles parsing command-line arguments.

## A.1 Compiling the ES

Compiling was done using a standard GNU C++ compiler like *gcc*. Since there is only one C++ file, compilation and linking is extremely simple. All that is needed are the five source files we described previously in the same directory and run the compiler command:

```
gcc es.cpp -o es.exe
```

This will produce an executable called *es.exe*.

## A.2 Running the ES

There is only one required argument for ES executable which is the ES file while we'll describe in more detail later. The other arguments are optional and are used to override settings in the ES, invoke special options, experimental settings, or for debug.

The command-line use of the *es.exe* has the following format:

```
es.exe <es_file> [-runs <uint>] [-gens <uint>] [-mu <uint>] [-lambda <uint>]
               [-mpl <1|0>] [-is <uint>] [-tprob <1|0>] [-tfit <uint>]
               [-fit <1|0>] [-train_mask <uint>] [-pm <uint>]
               [-recomb_mode <1|0>] [-recomb_parent <uint>] [-recomb_prob <uint>]
               [-d1] [-d2]
```

Descriptions:

=====

Required:

-----

<es_file>	# ES file
-----------	-----------

Optional:

-----

[-runs <uint>]	# number of ES runs
----------------	---------------------



```

[-gens <uint>]          # number of generations per run
[-mu <uint>]            # number of parents
[-lambda <uint>]        # number of children
[-mpl <1|0>]            # 1=(mu+lambda), 0=(mu,lambda)
[-is <uint>]            # initial/reset sigma (x 100)
[-tprob <1|0>]          # target mode: 1=probability, 0=complex
[-tfit <uint>]          # target fitness (x 100)
[-fit <1|0>]            # fitness mode (experimental)
[-train_mask <uint>]    # mask out certain input/output pairs from training
[-pm <uint>]            # probability of mutation
[-recomb_mode <1|0>]    # recombination mode: 1=intermediate, 0=discrete
[-recomb_parent <uint>] # number of parents used for recombination
[-recomb_prob <uint>]   # probability of apply recombination to an allele
[-d1]                  # debug level 1 messages (high level debug)
[-d2]                  # debug level 2 messages (low level debug)

```

### A.3 The ES File Format

The ES file is a text file which describes the configuration of the quantum circuit, the input and output states, and various parameters for the ES. The format for the ES file is as follows:

```

qubit=<uint>            # The number of qubits
runs=<uint>              # The number of runs for the ES
gens=<uint>              # The maximum number of generations per run
mu=<uint>                # The number of parents
lambda=<uint>            # The number of children
mpl=<0|1>                # 1=(mu+lambda), 0=(mu,lambda)
io_pairs=<uint>          # number of input/output states for training
init_sigma=<float>       # The value for the initial/reset sigma
target_prob=<0|1>        # target state mode: 1=probability, 0=complex
target_fitness=<float>   # Target fitness. 1=perfect. 0.75=75% fitness
# Define Input-Output states pairs (there will be <io_pairs> of these)
# ...

```

The last part of the ES file defines the input/output state pairs. These are based on the standard basis states. To support superposition, we can define the

input state using multiple pure states. The format of a single input/output pair is

$N$   $c_1|I_1\rangle$   $c_2|I_2\rangle$  ...  $c_N|I_N\rangle$   $M$   $d_1|O_1\rangle$   $d_2|O_2\rangle$  ...  $d_M|O_M\rangle$

Where  $N$  represents the number of basis state terms needed for the input state. The output state is made up of a superposition of  $M$  basis states. Therefore, using the form above, the input state,  $|\psi_i\rangle$ , will be:

$$|\psi_i\rangle = c_1 |I_1\rangle + \dots + c_N |I_N\rangle = \sum_{i=1}^N c_i |I_i\rangle \quad (\text{A.3.1})$$

And the output (or target) state,  $|\psi_t\rangle$ , will be:

$$|\psi_i\rangle = d_1 |O_1\rangle + \dots + d_N |O_M\rangle = \sum_{j=1}^M d_j |O_j\rangle \quad (\text{A.3.2})$$

An example ES file which trains a unitary operator to solve for general 3-node ISP problem is as follows:

```
qubits=3
runs=1
gens=1000
mu=15
lambda=500
mpl=1
io_pairs=8
init_sigma=0.1
target_prob=1
target_fitness=0.99
1 1.0|000> 1 1.0|111>
1 1.0|001> 2 0.7071067811865476|101> 0.7071067811865476|110>
1 1.0|010> 2 0.7071067811865476|011> 0.7071067811865476|110>
1 1.0|011> 1 1.0|110>
1 1.0|100> 2 0.7071067811865476|011> 0.7071067811865476|101>
1 1.0|101> 1 1.0|101>
```

```
1 1.0|110> 1 1.0|011>
1 1.0|111> 1 1.0|000>
```

## A.4 The ES Output

The ES executable does not generate any output files. All output is sent to the standard-output and was typically re-directed to a file. The output is verbose mostly for debugging purposes. The output has the following basic stages:

1. Parsing the ES file.
2. Constructing the Input/Output states.
3. Parsing the command-line arguments.
4. Showing the final parameters to be used by the ES.
5. Initializing the population.
6. Iterate through the runs and generation (main ES loops)
7. The best individual is displayed (per run and generation and overall)

Below is an example output taken from an ES executable for the 3-node ISP:

```
-D- Parsing isp.es
-D- Qubits=3, N=8, IO_PAIRS=8
-D- Read Input 0, term 0 = 000> with coefficient 1
-I- Converted 000> to 0
-D- Input State 0 with 1 terms
-D- Read Output 0, term 0 = 111> with coefficient 1
-I- Converted 111> to 7
-D- Output State 0 with 1 terms
-D- Input Vector = (8 x 1)[[1.0000000000000000e+00+0.0000000000000000...
```

```

-D- Output Vector = (8 x 1)[[0.0000000000000000e+00+0.0000000000000000...
-D- Read Input 1, term 0 = 001> with coefficient 1
-I- Converted 001> to 1
-D- Input State 1 with 1 terms
-D- Read Output 1, term 0 = 101> with coefficient 0.707107
-I- Converted 101> to 5
-D- Read Output 1, term 1 = 110> with coefficient 0.707107
-I- Converted 110> to 6
-D- Output State 1 with 2 terms
-D- Input Vector = (8 x 1)[[0.0000000000000000e+00+0.0000000000000000...
-D- Output Vector = (8 x 1)[[0.0000000000000000e+00+0.0000000000000000...
...
-D- Found -mu = 15
-D- Found -lambda = 500
-D- Found -is = 0.1
-D- Found -tfits = 0.98
-I- Info:
-I- Qubits = 3
-I- RUNS = 1, MAX Generations = 1000
-I- MU = 15, LAMBDA = 500
-I- MU_PLUS_LAMBDA = 1
-I- TARGET_PROB = 1
-I- TARGET_FITNESS = 0.98
-I- IO_PAIRS = 8
-I- INIT_SIGMA = 0.1
-D- Size of an individual = 394264 bytes
-D- Size of population = 203045960 bytes
-D- Setting poplist[0] to point to population[0]
-D- Setting poplist[1] to point to population[1]
-D- Setting poplist[2] to point to population[2]
-D- Setting poplist[3] to point to population[3]
-D- Setting poplist[4] to point to population[4]
-D- Setting poplist[5] to point to population[5]
...
-I- Generation: 0
-I- Run 0: Gen=0: Best=0.934777, Prob.Dist=(8 x 1)[[3.80492644562836...
-I- Generation: 1
-I- Run 0: Gen=1: Best=0.940681, Prob.Dist=(8 x 1)[[1.28243517050193...

```

```

-I- Generation: 2
-I- Run 0: Gen=2: Best=0.947328, Prob.Dist=(8 x 1)[[1.06678743852447...
-I- Generation: 3
-I- Run 0: Gen=3: Best=0.947328, Prob.Dist=(8 x 1)[[1.06678743852447...
-I- Generation: 4
-I- Run 0: Gen=4: Best=0.949542, Prob.Dist=(8 x 1)[[2.71768436890629...
-I- Generation: 5
-I- Run 0: Gen=5: Best=0.953403, Prob.Dist=(8 x 1)[[2.08142988475689...
...
-I- Best of Run 0 = 0.965463, with Prob.Dist=(8 x 1)[[2.575499149816...
-I- Best U = (8 x 8)[[2.6614338399183344e-02+1.5826139292238883e-01i...
-I- Final Parent Population and Fitness:
-I- Parent[0] = 0.965463, Prob.Dist=(8 x 1)[[2.5754991498160990e-02+...
-I- Parent[1] = 0.965463, Prob.Dist=(8 x 1)[[2.4917651369587426e-02+...
-I- Parent[2] = 0.965463, Prob.Dist=(8 x 1)[[2.4740887611428956e-02+...
-I- Parent[3] = 0.965463, Prob.Dist=(8 x 1)[[2.6511076756574343e-02+...
-I- Parent[4] = 0.965463, Prob.Dist=(8 x 1)[[2.5360371481615053e-02+...
-I- Parent[5] = 0.965462, Prob.Dist=(8 x 1)[[2.5849219920282794e-02+...
...
-I- VeryBest=0.965463, Err=7.739611, U=(8 x 8)[[2.6614338399183344e-...
-I- Probability Distribution for input 0 = (8 x 1)[[2.57549914981609...
-I- Probability Distribution for input 1 = (8 x 1)[[1.07400676397577...
-I- Probability Distribution for input 2 = (8 x 1)[[1.65591433105776...
-I- Probability Distribution for input 3 = (8 x 1)[[9.33277319285629...
-I- Probability Distribution for input 4 = (8 x 1)[[1.29769749481549...
-I- Probability Distribution for input 5 = (8 x 1)[[2.32171743177962...
-I- Probability Distribution for input 6 = (8 x 1)[[1.23535896048615...
-I- Probability Distribution for input 7 = (8 x 1)[[1.22447778361793...
...
-I- psi[1][2] = 1.60048 (sig_psi=0.00069657)
-I- phi[1][2] = 0.565421 (sig_phi=0.000177666)
-I- chi[2] = 0.49849 (sig_chi=0.00604527)
-I- psi[2][3] = 3.77186 (sig_psi=0.000184942)
-I- phi[2][3] = 0.00466546 (sig_phi=0.000373294)
-I- psi[1][3] = 4.31741 (sig_psi=0.0130396)
-I- phi[1][3] = 1.15522 (sig_phi=0.000436296)
...
-I- Showing the top 10 (or MU if MU < 10) in the final population:

```

```

-I- Parent[0] = 0.965463, U=(8 x 8) [[2.6614338399183344e-02+1.582613...
-I- Parent[1] = 0.965463, U=(8 x 8) [[5.0585520272807036e-02+1.495284...
-I- Parent[2] = 0.965463, U=(8 x 8) [[5.4524651803217361e-02+1.475396...
-I- Parent[3] = 0.965463, U=(8 x 8) [[3.9887442645027665e-02+1.578609...
-I- Parent[4] = 0.965463, U=(8 x 8) [[4.7408133526010823e-02+1.520290...
-I- Parent[5] = 0.965462, U=(8 x 8) [[4.4688607567537897e-02+1.544414...
-I- Parent[6] = 0.965462, U=(8 x 8) [[4.7483150090648414e-02+1.511969...
-I- Parent[7] = 0.965462, U=(8 x 8) [[4.1307960560509074e-02+1.587297...
-I- Parent[8] = 0.965462, U=(8 x 8) [[5.3721833473884963e-02+1.488251...
-I- Parent[9] = 0.965462, U=(8 x 8) [[5.8040128974821789e-02+1.459133...
-I- Done

```

## Appendix B

### ES Source Code

#### B.1 The Main ES Code: ES.CPP

```
#include "unitary.h"
#include "random.h"
#include <stdio.h>
#include <iostream.h>
#include "parse_args.h"          // my arg parser

#define MAX_POPSIZE    1500    // Max Population Size
#define MAX_IO_PAIRS   1024    // Max IO Pairs

// global
matrix INPUT_VECTOR[MAX_IO_PAIRS], TARGET_VECTOR[MAX_IO_PAIRS];

// basic globals
int RUNS = 0;                    // Max number of runs
int GENS = 0;                    // Max generations
int QUBITS = 0;                  // Qubits
int N = 0;                       //  $2^{\text{(qubits)}}$ , number of states
int PM = 100;                    // Prob. of mutation, usually 100%
int MU = 0;                       // Number of parents
int LAMBDA = 0;                  // Number of children
int POPSIZE = 0;                 // MU+LAMBDA
int MU_PLUS_LAMBDA = 0;          // =1 for (mu+lambda), =0 for (mu,lambda)
int DEBUG1 = 0, DEBUG2 = 0;      // Dumb level1 and/or level2 debug information
int IO_PAIRS = 0;                // Number of IO pairs to train with
double TARGET_FITNESS = 1.0;     // Target Fitness to quit training
```

```

double INIT_SIGMA = 0.1;          // Value to initially set sigmas to and also reset when we hit
                                   // boundaries

int TARGET_PROB = 1;              // =1 to use the target vector as a probability or =0 if it's
                                   // treated as a complex (pre-measure) value

int FITNESS = 0;                  // =0 for SUM(1/(err+1)) (slightly better)
                                   // =1 for 1/(SUM(err)+1)

unsigned long TRAIN_MASK = 0xFFFFFFFF; // determines which I/O pairs we should use for training,
                                   // upto 32 I/O pairs (5 qubits)

double ROUND_ANGLES = PI / 12.0;   // 0.261799387799149; if >0, then
                                   // we round the angles after mutation to the closest
                                   // integral of ROUND_ANGLES

int RECOMB_MODE = 1;               // 1=intermediate(average), 0=discrete (pick one)

int RECOMB_PARENTS = 2;            // how many parents to chose for recombination

int RECOMB_PROB = 10;              // 10% chance that we'll actually recombine an angle/sig_angle

double FN = (double) N;            // For real-number equations

float tau = 0.0,                   // use FN-1 = N-1 = number of angle parameters
      tau_prime = 0.0;

typedef unsigned long uint32;

int ParseFile (char *);            // This loads our cfigs from a file, qubits, flags, input/output
                                   // pairs, etc.

matrix ConvertState (char *, int); // Convert from a state string (e.g, |0110>) to a vector

// create the structure which represents an individual
class individual
{
public:
    // First, our object parameters which are the angles which define the unitary matrix
    double psi[MAX_ANGLES][MAX_ANGLES], phi[MAX_ANGLES][MAX_ANGLES], chi[MAX_ANGLES];
    // Second, are our strategy parameters
    double sig_psi[MAX_ANGLES][MAX_ANGLES], sig_phi[MAX_ANGLES][MAX_ANGLES], sig_chi[MAX_ANGLES];
    // fitness value
    double fitness, err;
    // unitary matrix
    matrix U;

```



```

// functions:

individual ()
{
    // random initialization
    // initialize();
}

void initialize ()
{
    int i, j;
    // random angles
    for (int j = 2; j <= N; j++)
    {
        for (int i = j - 1; i > 0; i--)
        {
            psi[i][j] = 2.0 * PI * uniform (&idum);
            phi[i][j] = PI / 2.0 * uniform (&idum);
            sig_psi[i][j] = INIT_SIGMA;    // 1.0;
            sig_phi[i][j] = INIT_SIGMA;    // 1.0;
        }
        chi[j] = 2.0 * PI * uniform (&idum);
        sig_chi[j] = INIT_SIGMA;    // 1.0;
    }
    U = create_unitary_matrix (N, psi, phi, chi);
    // calc_fitness(INPUT_VECTOR, TARGET_VECTOR); // need to fix eventually
    calc_fitness ();    // need to fix eventually
}

void update_matrix ()
{
    // run this when we've updated the angle parameters
    if (DEBUG2)
        cout << "\n-D- Updating Matrix!";
    U = create_unitary_matrix (N, psi, phi, chi);
}

double calc_fitness (matrix * I, matrix * T)
{
    // given column vectors I (input)
    // and desired output, T, determine fitness

    int i;

```

```

matrix R;                // (N,1); // result vector ~ Nx1 = column vector
err = 0.0;
fitness = 0.0;
if (DEBUG2)
{
    printf ("\n-D- calc_fitness U=");
    U.show ();
}
// Here, we assume T is always complex, so if we're in TARGET_PROB mode, it means we use
// |T|^2 instead!
if (TARGET_PROB)
{
    R = (U * (*I)).modulus_squared () - T->modulus_squared ();
    if (DEBUG2)
    {
        printf ("\n-D- R=|U*I|^2 - |T|^2 = ");
        R.show ();
    }
}
else
{
    R = (U * (*I)) - (*T);
    if (DEBUG2)
    {
        printf ("\n-D- R=(U*I) - T = ");
        R.show ();
    }
}
// At this point, R is our error vector (either complex or probability error)
// The fitness function is : SUM(i=0 to N-1)(1/(((^2+epsilon)))
// Either way, we need absolute positive error, so we use
// do |R|^2 or |R|.
// R = R.modulus_squared(); // mod_squared is faster than modulus...
R = R.modulus ();        // mod_squared is faster than modulus...
if (DEBUG2)
{
    printf ("\n-D- |R| = ");
    R.show ();
}

```

```

    }
    for (i = 0; i < N; i++)
    {
        if (FITNESS)
        {
            // = 1/(SUM(err)+1)
            err += R.M[i][0].r;    // sum the absolute errors
        }
        else
        {
            // = SUM(1/(err+1)) - should be better
            err += R.M[i][0].r;    // sum the absolute errors
            fitness += 1.0 / (R.M[i][0].r + 1.0);
        }
    }
}

if (FITNESS)
{
    // = 1/(SUM(err)+1) - previously, this yielded bad results
    fitness = (1.0 / (err + 1.0));    // perfect fit is when err=0, fitness = 1
}
else
{
    // = SUM(1/(err+1))
    // perfect fit when err = 0, fitness = N
    // fitness = fitness/N; // so that perfect fitness = 1
}

if (DEBUG1)
    printf ("\n-D- Err=%lf, Fitness=%lf", err, fitness);
return fitness;    // return fitness
}

double calc_fitness ()
{
    // uses all available IO PAIRS
    double myfit = 0.0, my_err = 0.0;

    int i;
    int cnt = 0;

    for (i = 0; i < IO_PAIRS; i++)
    {
        // test across all IO pairs
        if ((unsigned long) pow (2, i) & TRAIN_MASK)
        {
            // are we supposed to train with this io-pair?
            if (DEBUG2)

```

```

        cout << "\n-D- Calculating fitness for input " << i;
        calc_fitness (&INPUT_VECTOR[i], &TARGET_VECTOR[i]);
        myfit += fitness;
        my_err += err;
        cnt++;
    }
    else
    {
        if (DEBUG2)
            cout << "\n-D- skipping io pair number " << i;
    }
}

// fitness = myfit/IO_PAIRS; // Avg. fitness of all pairs
// New method, take cumulative err and recalculate fitness
if (FITNESS)
{
    // = 1/(SUM(err)+1)
    fitness = 1.0 / (my_err + 1.0);    // perfect fit = 1 when my_err = 0
}
else
{
    // = SUM(1/(err+1))
    fitness = myfit;    // perfect fit = N*IO_PAIRS or IO_PAIRS if already divided by N
    fitness = myfit / (cnt * N);
}

if (DEBUG2)
    cout << "\n-D- Total err was " << my_err << " and fitness was " << fitness;
return fitness;
}

void show_probabilities (matrix * I)
{
    // this function does a mod_squared of the result to give the
    // probability distribution

    matrix P;
    P = (U * (*I)).modulus_squared ();
    // show P
    // printf("\n-I- Probability Distribution = "); P.show();
    P.show ();
}

void show_probabilities ()
{
    // show probability distribution (O=U*I) mod squared

```

```

// for all INPUT vectors

matrix P;

int i;
for (i = 0; i < IO_PAIRS; i++)
{
    P = (U * INPUT_VECTOR[i]).modulus_squared ();
    printf ("\n-I- Probability Distribution for input %d = ", i);
    P.show ();
}
}

void show_angles ()
{
    // shows the angles

    int i, j;
    for (int j = 2; j <= N; j++)
    {
        for (int i = j - 1; i > 0; i--)
        {
            cout << "\n-I- psi[" << i << "]" << j << " = " << psi[i][j] << " (sig_psi=" <<
                sig_psi[i][j] << ")";
            cout << "\n-I- phi[" << i << "]" << j << " = " << phi[i][j] << " (sig_phi=" <<
                sig_phi[i][j] << ")";
        }
        cout << "\n-I- chi[" << j << "]" << " = " << chi[j] << " (sig_chi=" << sig_chi[j] << ")";
    }
}

}; // class individual

void
copy_individual (individual * A, individual * B)
{
    // copy A to B

    // method 1 - this won't work if it only copies pointers
    B->U = A->U;
    B->fitness = A->fitness;
    B->err = A->err;
    for (int i = 0; i < MAX_ANGLES; i++)
    {
        for (int j = 0; j < MAX_ANGLES; j++)
        {

```

```

        B->psi[i][j] = A->psi[i][j];
        B->phi[i][j] = A->phi[i][j];
        B->sig_psi[i][j] = A->sig_psi[i][j];
        B->sig_phi[i][j] = A->sig_phi[i][j];
    }
    B->chi[i] = A->chi[i];
    B->sig_chi[i] = A->sig_chi[i];
}

} // copy_individual

void
mutate (individual * p)
{
    // mutate individual p
    // each angle has a strategy parameter (sigma) such as sig_psi, sig_phi, and sig_chi
    // we update the angles according to:
    // beta = tau_prime*RAND_NORMAL;
    // sig_psi = sig_psi*exp(beta+(tau*RAND_NORMAL)
    // ...etc..
    // then update angles:
    // psi = psi + (sig_psi*RAND_NORMAL)

    int i, j;
    float beta = tau_prime * normal (&idum);

    if (DEBUG2)
        cout << "\n-D- MUTATE: beta = " << beta << ", tau = " << tau;
    for (j = 2; j <= N; j++)
    {
        for (i = j - 1; i > 0; i--)
        {
            if (DEBUG2)
                cout << "\n-D- MUTATE BEFORE: sig_psi[" << i << "][" << j << "] = " << p->
                    sig_psi[i][j];

            if (DEBUG2)
                cout << "\n-D- MUTATE BEFORE: psi[" << i << "][" << j << "] = " << p->psi[i][j];

            p->sig_psi[i][j] *= exp (beta + tau * normal (&idum));
            p->sig_phi[i][j] *= exp (beta + tau * normal (&idum));
            // reset strategy parameters

```

```

    if (p->sig_psi[i][j] < 0.0001 || p->sig_psi[i][j] > 2.0)
        p->sig_psi[i][j] = INIT_SIGMA; // 1.0;
    if (p->sig_phi[i][j] < 0.0001 || p->sig_phi[i][j] > 2.0)
        p->sig_phi[i][j] = INIT_SIGMA; // 1.0;

    p->psi[i][j] += p->sig_psi[i][j] * normal (&idum);
    p->phi[i][j] += p->sig_phi[i][j] * normal (&idum);

    // limit checks, phi is between 0 and pi/2, psi and chi are 0 to 2*pi
    if (p->psi[i][j] < 0.0)
        p->psi[i][j] = 0.0;
    if (p->phi[i][j] < 0.0)
        p->phi[i][j] = 0.0;
    if (p->psi[i][j] > 2 * PI)
        p->psi[i][j] = 2 * PI;
    if (p->phi[i][j] > PI / 2)
        p->phi[i][j] = PI / 2.0;
    if (DEBUG2)
        cout << "\n-D- MUTATE AFTER: sig_psi[" << i << "][" << j << "] = " << p->
            sig_psi[i][j];
    if (DEBUG2)
        cout << "\n-D- MUTATE AFTER: psi[" << i << "][" << j << "] = " << p->psi[i][j];
}

if (DEBUG2)
    cout << "\n-D- MUTATE BEFORE: sig_chi[" << j << "] = " << p->sig_chi[j];
if (DEBUG2)
    cout << "\n-D- MUTATE BEFORE: chi[" << j << "] = " << p->chi[j];
p->sig_chi[j] *= exp (beta + tau * normal (&idum));
if (p->sig_chi[j] < 0.0001 || p->sig_chi[j] > 2.0)
    p->sig_chi[j] = INIT_SIGMA; // 1.0;

p->chi[j] += p->sig_chi[j] * normal (&idum);
if (p->chi[j] < 0.0)
    p->chi[j] = 0.0;
if (p->chi[j] > 2 * PI)
    p->chi[j] = 2 * PI;
if (DEBUG2)
    cout << "\n-D- MUTATE AFTER: sig_chi[" << j << "] = " << p->sig_chi[j];

```

```

        if (DEBUG2)
            cout << "\n-D- MUTATE AFTER: chi[" << j << "] = " << p->chi[j];
    }
    // mutate parameters, update the matrix
    p->update_matrix ();
}
// mutate

void
round_angles (individual * p, double angle)
{
    // rounds the matrix angles using the supplied granularity

    int n;
    double r, f;
    int i, j;

    if (DEBUG1)
        cout << "\n-D- round_angles: angle=" << angle;

    // round the angles
    // ? what to do with the sigmas?
    for (int j = 2; j <= N; j++)
    {
        for (int i = j - 1; i > 0; i--)
        {
            // psi
            r = p->psi[i][j] / angle;
            n = int (r);
            f = r - n;
            if (f >= 0.5)
            {
                n++;
            }
            if (DEBUG1)
                cout << "\n-D- round_angles: changing psi[" << i << "][" << j << "] from " << p->
                    psi[i][j] << " to " << n * angle;
            p->psi[i][j] = n * angle;
            // phi
            r = p->phi[i][j] / angle;

```



```

        n = int (r);
        f = r - n;
        if (f >= 0.5)
        {
            n++;
        }
        if (DEBUG1)
            cout << "\n-D- round_angles: changing phi[" << i << "]"[" << j << "]" from " << p->
                phi[i][j] << " to " << n * angle;
            p->phi[i][j] = n * angle;
        }
        // chi
        r = p->chi[j] / angle;
        n = int (r);
        f = r - n;
        if (f >= 0.5)
        {
            n++;
        }
        if (DEBUG1)
            cout << "\n-D- round_angles: changing chi[" << j << "]" from " << p->
                chi[j] << " to " << n * angle;
            p->chi[j] = n * angle;
        }
    }

    // round_angles

//
// *** MAIN ***
//
// FORMAT: <exe> <cfg_file>
int
main (int argc, char *argv[])
{
    int run, gen, i, rv, j, bi;
    individual *population;    // actual population in no-sorted order
    individual *poplist[MAX_POPSIZE];    // this is the sorted list of population
    individual best, verybest, *temp;

```

```

int plist[MAX_POPSIZE];    // tracks a list of parents for recomb, maybe other things

// Parse cfg file
ParseFile (argv[1]);

// Allow args to override certain settings, not qubits
if (GetIntOpt (MU, "-mu", argc, argv))
{
    cout << "\nD- Found -mu = " << MU;
}
if (GetIntOpt (LAMBDA, "-lambda", argc, argv))
{
    cout << "\nD- Found -lambda = " << LAMBDA;
}
GetIntOpt (MU_PLUS_LAMBDA, "-mpl", argc, argv);
int tmpvar = 0;
if (GetIntOpt (tmpvar, "-is", argc, argv))
{
    // init sigma is divided by 100. so, for 0.1, give -is 10
    INIT_SIGMA = (double) tmpvar / 100.0;
    cout << "\nD- Found -is = " << INIT_SIGMA;
}
GetIntOpt (TARGET_PROB, "-tprob", argc, argv);
if (GetIntOpt (tmpvar, "-tfit", argc, argv))
{
    // target fitness is divided by 100. so, for 0.99, give -is 99
    TARGET_FITNESS = (double) tmpvar / 100.0;
    cout << "\nD- Found -tfit = " << TARGET_FITNESS;
}
if (GetIntOpt (tmpvar, "-train_mask", argc, argv))
{
    TRAIN_MASK = (unsigned long) tmpvar;
    cout << "\nD- Found -train_mask = " << TRAIN_MASK;
}

// Mutation/Recomb probability
GetIntOpt (PM, "-pm", argc, argv);

```

```

// Recombination knobs
GetIntOpt (RECOMB_MODE, "-recomb_mode", argc, argv);
GetIntOpt (RECOMB_PARENTS, "-recomb_parents", argc, argv);
GetIntOpt (RECOMB_PROB, "-recomb_prob", argc, argv);

// Get RUN and GEN overrides
GetIntOpt (RUNS, "-runs", argc, argv);
GetIntOpt (GENS, "-gens", argc, argv);

// debug
DEBUG1 = GetBoolOpt ("-d1", argc, argv);
DEBUG2 = GetBoolOpt ("-d2", argc, argv);

// fitness level
GetIntOpt (FITNESS, "-fit", argc, argv);

// recalc again just in case we changed MU or LAMBDA
POPSIZE = MU + LAMBDA;

tau = 1.0 / sqrt (2.0 * sqrt (FN - 1.0)); // use FN-1 = N-1 = number of angle parameters
tau_prime = 1.0 / sqrt (2.0 * (FN - 1.0));

population = new individual[POPSIZE];

cout << "\n-I- Info:";
cout << "\n-I- Qubits = " << QUBITS;
cout << "\n-I- RUNS = " << RUNS << ", MAX Generations = " << GENS;
cout << "\n-I- MU = " << MU << ", LAMBDA = " << LAMBDA;
cout << "\n-I- MU_PLUS_LAMBDA = " << MU_PLUS_LAMBDA;
cout << "\n-I- TARGET_PROB = " << TARGET_PROB;
cout << "\n-I- TARGET_FITNESS = " << TARGET_FITNESS;
cout << "\n-I- IO_PAIRS = " << IO_PAIRS;
cout << "\n-I- INIT_SIGMA = " << INIT_SIGMA;
cout << "\n-I- TRAINING_MASK = " << TRAIN_MASK;
cout << "\n-I- ROUND_ANGLES = " << ROUND_ANGLES;

```

```

cout << "\n-I- PROBABILITY OF MUTATION vs RECOMBINATION = " << PM;
cout << "\n-I- RECOMB: MODE=" << RECOMB_MODE << ", PROB=" << RECOMB_PROB << ", PARENTS=" <<
    RECOMB_PARENTS;

// size check
printf ("\n-D- Size of an individual = %d bytes", sizeof (individual));
printf ("\n-D- Size of population    = %d bytes", sizeof (individual) * POPSIZE);

// simple check
if (MU + LAMBDA != POPSIZE)
{
    cout << "\n-E- MU(" << MU << ") + LAMBDA("
        << LAMBDA << ") != POPSIZE(" << POPSIZE << ")!\n";
    exit (1);
}

// initialize random
srand ((unsigned) time (NULL));
// idum = -rand(); // only rand in the code
idum = -1193804919; // hard-coded seed for now

best.fitness = verybest.fitness = -999999.9;

// matrix H = HADAMARD();
// matrix H = PHASE();

// set initial INPUT, TARGET.
// INPUT is all 1's times 1/sqrt(N)
// INPUT_VECTOR[0] = ONEHOT_COLUMN_VECTOR(1,N);
// INPUT_VECTOR[1] = ONEHOT_COLUMN_VECTOR(2,N);
// Determine output vectors
/* Examples:
    // TARGET_VECTOR[0] = ONEHOT_COLUMN_VECTOR(1,N);
    // TARGET_VECTOR[1] = (H*INPUT_VECTOR[1]);
*/
for (i = 0; i < IO_PAIRS; i++)
{
    INPUT_VECTOR[i] = matrix (N, 1);

```

```

        INPUT_VECTOR[i].fill (1.0 / sqrt (FN));
        // TARGET_VECTOR[i] = ONEHOT_COLUMN_VECTOR(1,N);
        printf ("\n-I- INPUT_VECTOR[%d]=", i);
        INPUT_VECTOR[i].show ();
        printf ("\n-I- TARGET_VECTOR[%d]=", i);
        TARGET_VECTOR[i].show ();
    }

    // set initial poplist to be 1-to-1 with population
    for (i = 0; i < POPSIZE; i++)
    {
        printf ("\n-D- Setting poplist[%d] to point to population[%d]", i, i);
        poplist[i] = &population[i];
    }

    // check random individual
    j = rOn (POPSIZE);
    if (DEBUG1)
        printf ("\n-D- Checking individual %d", j);
    if (DEBUG1)
        printf ("\n-D- addr = %d, poplist[%d]= %d", &population[j], j, poplist[j]);

    // OPTIMIZATION IDEA:
    // Might be easier to just have population[] and always keep the parents at the top.
    // When sorting, for (mu,lambda), we only sort using the bottom lambda
    // individuals but fill from the top.
    // Whereas, with (mu+lambda) we sort the whole population.

    if (DEBUG1)
        printf ("\n-D- Starting Runs Loop\n");
    for (run = 0; run < RUNS; run++)
    {
        // intialize parents which are the top MU of the poplist
        for (i = 0; i < MU; i++)
        {
            if (DEBUG1)
                printf ("\n-D- Init Parent %d", i);
            poplist[i]->initialize ();
            if (DEBUG1)

```

```

    {
        printf (" - Fitness = %lf", poplist[i]->fitness);
        poplist[i]->U.show ();
    }
}

// generation loop
for (gen = 0; gen < GENS; gen++)
{
    cout << "\n-I- Generation: " << gen;
    for (i = 0; i < LAMBDA; i++)
    {
        // create the LAMBDA children
        rv = rOn (100) + 1;
        if (rv <= PM)
        {
            // do mutation
            j = rOn (MU);      // pick random parent
            if (DEBUG1)
                cout << "\n-D- MUTATE: Picking random parent: " << j
                    << ". Copying to " << i+ MU;
            // copy parent to child portion of pool (spot i+mu)
            copy_individual (poplist[j], poplist[i + MU]);
            mutate (poplist[i + MU]);
        }
        else
        {
            // recombination
            j = rOn (MU);      // pick random parent
            copy_individual (poplist[j], poplist[i + MU]);
            if (DEBUG1)
                cout << "\n-D- RECOMB: Picking random parent: " << j
                    << ". Copying to " << i+ MU;
            if (DEBUG2)
                cout << "\n-D- Recombination, mode = " << RECOMB_MODE
                    << ", parents = " << RECOMB_PARENTS;

            // select k random parents
            for (int ii = 0; ii < RECOMB_PARENTS; ii++)
            {
                plist[ii] = rOn (MU);    // re-selection is fine
                if (DEBUG2)

```

```

        cout << "\n-D- Selected parent " << plist[ii]
            << " for recombination";
    }
    // go through each allele
    for (int jj = 2; jj <= N; jj++)
    {
        for (int ii = jj - 1; ii > 0; ii--)
        {
            // PHI
            if ((r0n (100) + 1) <= RECOMB_PROB)
            { // recombine?
                if (RECOMB_MODE)
                { // intermediate
                    double tmpa = 0.0, tmps = 0.0;
                    for (int pp = 0; pp < RECOMB_PARENTS; pp++)
                    {
                        tmpa += poplist[pp]->psi[ii][jj]; // angle
                        tmps += poplist[pp]->sig_psi[ii][jj]; // sigma
                    }
                    tmpa = tmpa / double (RECOMB_PARENTS);
                    tmps = tmps / double (RECOMB_PARENTS);
                    // update angle
                    poplist[i + MU]->psi[ii][jj] = tmpa;
                    poplist[i + MU]->sig_psi[ii][jj] = tmps;
                    if (DEBUG2)
                        cout << "\n-D- RECOMB INTERMEDIATE: psi"
                            << ii << " " << jj
                            << " updated to value of " << tmpa
                            << ", psi_sig to " <<
                                tmps;
                }
            }
            else
            { // discrete
                int pp = r0n (RECOMB_PARENTS);
                poplist[i + MU]->psi[ii][jj] =
                    poplist[pp]->psi[ii][jj];
                poplist[i + MU]->sig_psi[ii][jj] =
                    poplist[pp]->sig_psi[ii][jj];
            }
        }
    }
}

```

```

        if (DEBUG2)
            cout << "\n-D- RECOMB DISCRETE: psi "
                << ii << " " << jj
                << " updated from parent " << pp
                << " to " << poplist[pp]->
                    psi[ii][jj] << ", psi_sig to "
                << poplist[pp]->sig_psi[ii][jj];
    }
}

// PHI
if ((r0n (100) + 1) <= RECOMB_PROB)
{
    // recombine?
    if (RECOMB_MODE)
    {
        // intermediate
        double tmpa = 0.0, tmps = 0.0;
        for (int pp = 0; pp < RECOMB_PARENTS; pp++)
        {
            tmpa += poplist[pp]->phi[ii][jj];      // angle
            tmps += poplist[pp]->sig_phi[ii][jj];  // sigma
        }
        tmpa = tmpa / double (RECOMB_PARENTS);
        tmps = tmps / double (RECOMB_PARENTS);
        // update angle
        poplist[i + MU]->phi[ii][jj] = tmpa;
        poplist[i + MU]->sig_phi[ii][jj] = tmps;
    }
    else
    {
        // discrete
        int pp = r0n (RECOMB_PARENTS);
        poplist[i + MU]->phi[ii][jj] =
            poplist[pp]->phi[ii][jj];
        poplist[i + MU]->sig_phi[ii][jj] =
            poplist[pp]->sig_phi[ii][jj];
    }
}

}

// CHI
if ((r0n (100) + 1) <= RECOMB_PROB)

```



```

{      // recombine?
    if (RECOMB_MODE)
    { // intermediate
        double tmpa = 0.0, tmps = 0.0;
        for (int pp = 0; pp < RECOMB_PARENTS; pp++)
        {
            tmpa += poplist[pp]->chi[jj];      // angle
            tmps += poplist[pp]->sig_chi[jj];  // sigma
        }
        tmpa = tmpa / double (RECOMB_PARENTS);
        tmps = tmps / double (RECOMB_PARENTS);
        // update angle
        poplist[i + MU]->chi[jj] = tmpa;
        poplist[i + MU]->sig_chi[jj] = tmps;
    }
    else
    { // discrete
        int pp = r0n (RECOMB_PARENTS);
        poplist[i + MU]->chi[jj] = poplist[pp]->chi[jj];
        poplist[i + MU]->sig_chi[jj] = poplist[pp]->sig_chi[jj];
    }
}

// we've changed the params, recreate the unitary matrix
poplist[i + MU]->update_matrix ();

// Round the angles if desired
if (ROUND_ANGLES)
{
    round_angles (poplist[i + MU], ROUND_ANGLES);
    // this modifies the angles, need to update the matrix
    poplist[i + MU]->update_matrix ();
}

// compute fitness of new child
// poplist[i+MU]->calc_fitness(INPUT_VECTOR, TARGET_VECTOR);
poplist[i + MU]->calc_fitness ();
}

// sort_population(poplist,k); TO-DO, sort the bottom k individuals to the top

```

```

// (for Mu,Lambda, k = LAMBDA, for Mu+Lambda, k = POPSIZE)
if (DEBUG1)
    cout << "\n-D- Sorting!";
for (i = MU_PLUS_LAMBDA ? 0 : MU; i < POPSIZE - 1; i++)
{
    // FOR (Mu,Lambda), start i at MU rather than 0
    bi = i;          // initial best
    for (j = i + 1; j < POPSIZE; j++)
    {
        // compare best to others
        if (poplist[bi]->fitness < poplist[j]->fitness)
        {
            // update best-index
            bi = j;
        }
    }
    if (DEBUG2)
        cout << "\n-D- Swapping " << i << "(" << poplist[i]
            << ") = " << poplist[i]->fitness << " with "
            << bi << "(" << poplist[bi] << ") = " << poplist[bi]->fitness;
    temp = poplist[i];
    if (MU_PLUS_LAMBDA)
    {
        poplist[i] = poplist[bi];
    }
    else
    {
        poplist[i - MU] = poplist[bi];    // for (mu,lambda), copy the children to
                                           // the parent space of the poplist
    }
    poplist[bi] = temp;
    if (DEBUG2)
        cout << "\n-D- Check Swap: " << i << "(" << poplist[i]
            << ") = " << poplist[i]->fitness << " , " << bi
            << "(" << poplist[bi] << ") = " << poplist[bi]->fitness;
}
if (best.fitness < poplist[0]->fitness)
{
    if (DEBUG1)
        printf ("\n-D- Updating fitness from %lf to %lf", best.fitness,
            poplist[0]->fitness);
}

```

```

        copy_individual (poplist[0], &best);    // NOTE, best is not a pointer. Need to
                                                // OVERLOAD copy_individual function?
    }
    // output generation report (best individual fitness)
    printf ("\n-I- Run %d: Gen=%d: Best=%lf, Prob.Dist=", run, gen, best.fitness);
    best.show_probabilities (&INPUT_VECTOR[0]);

    // did we meet target fitness!
    // if(best.fitness >= TARGET_FITNESS) {
    double foo = (best.U * INPUT_VECTOR[0]).M[0][0].modulus_squared ();
    if (foo > TARGET_FITNESS)
    {
        printf ("\n-I- Target fitness of %lf met by %lf in
                generation %d!", TARGET_FITNESS, best.fitness, gen);
        // break all loops
        run = RUNS;
        gen = GENS;
    }
}

printf ("\n-I- Best of Run %d = %lf, with Prob.Dist=", run, best.fitness);
best.show_probabilities (&INPUT_VECTOR[0]);
printf ("\n-I- Best U = ");
best.U.show ();
if (verybest.fitness < best.fitness)
{
    copy_individual (&best, &verybest); // keep track of overall very best
}
best.fitness = -999999.9;    // reset best
}

// display final parent population
printf ("\n-I- Final Parent Population and Fitness:");
for (i = 0; i < MU; i++)
{
    printf ("\n-I- Parent[%d] = %lf, Prob.Dist=", i, poplist[i]->fitness);
    poplist[i]->show_probabilities (&INPUT_VECTOR[0]);
}
// check

```

```

// For the final calculations, allow all I/O PAIRs for fitness computation
printf ("\n-I- VeryBest=%lf, Err=%lf, U=", verybest.fitness, verybest.err);
verybest.U.show ();
printf ("\n-I- VeryBest Generalization Test on all I/O Pairs");
TRAIN_MASK = 0xFFFF;
verybest.calc_fitness ();
printf ("\n-I- VeryBest All IO Fitness=%lf, Err=%lf, U=", verybest.fitness, verybest.err);
verybest.U.show ();
if (DEBUG1)
{
    printf ("\n-I- CHECK: VeryBest=%lf, Err=%lf, U=", verybest.fitness, verybest.err);
    verybest.U.show ();
}
verybest.show_probabilities ();
verybest.show_angles ();
// Show top (?) in population
printf ("\n-I- Showing the top 10 (or MU if MU < 10) in the final population:");
for (i = 0; i < 10 && i < MU; i++)
{
    poplist[i]->calc_fitness ();
    printf ("\n-I- Parent[%d] = %lf, U=", i, poplist[i]->fitness);
    poplist[i]->U.show ();
}
printf ("\n-I- Done\n");
delete population;
}

int
ParseFile (char *fn)
{
    // File format is defined as follows:
    // qubits=<int>
    // runs=<int>
    // gens=<int>
    // mu=<int>
    // lambda=<int>
    // mpl=<int>
    // io_pairs=<int>

```

```

// init_sigma=<float>
// target_prob=<int>
// target_fitness=<double>
// train_mask=<ulong>
// |<input_state_0>> <terms> <float_coeff0>|output0> <float_coeff1>|output1> ...
// ...

int i, j, n;
FILE *fh = fopen (fn, "rb");
char str[80];
double coef;
matrix input, output;

cout << "\n-D- Parsing " << fn;

fscanf (fh, "qubits=%d\n", &QUBITS);
fscanf (fh, "runs=%d\n", &RUNS);
fscanf (fh, "gens=%d\n", &GENS);
fscanf (fh, "mu=%d\n", &MU);
fscanf (fh, "lambda=%d\n", &LAMBDA);
fscanf (fh, "mpl=%d\n", &MU_PLUS_LAMBDA);
fscanf (fh, "io_pairs=%d\n", &IO_PAIRS);
fscanf (fh, "init_sigma=%lf\n", &INIT_SIGMA);
fscanf (fh, "target_prob=%d\n", &TARGET_PROB);
fscanf (fh, "target_fitness=%lf\n", &TARGET_FITNESS);
fscanf (fh, "train_mask=0x%lx\n", &TRAIN_MASK);
fscanf (fh, "round_angles=%lf\n", &ROUND_ANGLES);

N = (int) pow (2, QUBITS);
FN = (double) N;
POPSIZE = MU + LAMBDA;

// sanity check
cout << "\n-D- Qubits=" << QUBITS << ", N=" << N << ", IO_PAIRS=" << IO_PAIRS;
cout << "\n-D- Training Mask = " << TRAIN_MASK;

// read i/o pairs which will have dimension matrix(N,1)
for (i = 0; i < IO_PAIRS; i++)

```

```

{
    input.zero (N, 1);      // init input vector
    fscanf (fh, "%d", &n); // read number of input terms
    for (j = 0; j < n; j++)
    {
        fscanf (fh, "%lf|%s", &coef, str);
        cout << "\n-D- Read Input " << i << ", term " << j << " = " << str <<
            " with coefficient " << coef;
        input += (ConvertState (str, QUBITS) * coef);
    }
    cout << "\n-D- Input State " << i << " with " << n << " terms";
    output.zero (N, 1);    // init output vector
    fscanf (fh, "%d", &n); // read number of output terms
    for (j = 0; j < n; j++)
    {
        if (j == n - 1)
        {
            fscanf (fh, "%lf|%s\n", &coef, str);
        }
        else
        {
            fscanf (fh, "%lf|%s", &coef, str);
        }
        cout << "\n-D- Read Output " << i << ", term " << j << " = " << str <<
            " with coefficient " << coef;
        output += (ConvertState (str, QUBITS) * coef);
    }
    cout << "\n-D- Output State " << i << " with " << n << " terms";
    // define the input/output
    INPUT_VECTOR[i] = input;
    TARGET_VECTOR[i] = output;
    cout << "\n-D- Input Vector = ";
    input.show ();
    cout << "\n-D- Output Vector = ";
    output.show ();
}
fclose (fh);

```

```

        return 1;
    }

    matrix
    ConvertState (char *str, int q)
    {
        // Convert from a state string (e.g, |0110>) to a vector
        matrix v;
        int i, index = 0;
        int rows = (int) pow (2, q);

        for (i = 0; i < q; i++)
        {
            if (str[q - i - 1] == '1')
            {
                index += (int) pow (2, i);
            }
        }

        cout << "\n-I- Converted " << str << " to " << index;
        v.zero (rows, 1);
        v (index, 0) = 1.0;

        return v;
    }

```

## B.2 The Header Files

### B.2.1 MATRIX.H

```
// MATRIX library by Steven R. Hutsell
// June 25th, 2006
// Currently performs very simple complex matrix and vector operations
// such as multiply, add and subtract.
// Vector ops like inner (dot) and outer product are
// performed using single column and/or single row matrices
//
// To be extended is the ability to do more complex operations
// such as determinants, inverse, adjoints, congruence, transpose, etc.
//
// More advanced linear operations will likely not be added but who
// knows. For example, RREF and eigen-analysis to name a couple.
//
// RULES:
// =====
// When overloading operators, we set certain LEFT-RIGHT precedences:
//
// Rule 1: multiply/dividing a complex with a scalar:
// Always have the scalar to the RIGHT of the complex:
// E.g., z*k or z/k, never k*z or k/z
// to get 1/z then create a complex 1 = (1+i0) then do complex_1/z instead.
//
// Rule 2: multiply/dividing a matrix by a complex or scalar
// Always have the matrix to the LEFT and the complex or scalar to the RIGHT
// E.g., M*k, M/k, M*z, M/z. Never k*M or z*M or k/M

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>

#define MAX_ROWS    64          // 6 qubit
#define MAX_COLUMNS 64

#define STRING_WIDTH 48         // String width of a complex number
```



```

#define PI 3.141592653589793238462643383279502884197169399375105820974944592307

// Print Format: 0=MATLAB=[[a+bi]], 1=DERIVE=[[a+#ib]]
// #define PRINT_FORMAT 0      // MATLAB
enum printformats
{
    MATLAB,
    DERIVE
};
enum precisionformats
{
    HIGH,
    LOW
};

// by default, set PRINT_FORMAT to MATLAB
printformats PRINT_FORMAT = MATLAB;    // DERIVE;
// by default, set precision to LOW
precisionformats PRECISION_FORMAT = HIGH;    // LOW;

// Complex class
class complex
{
public:
    // vars
    double r, i;           // the real and imaginary components
    char s[STRING_WIDTH];  // string of the complex number

    // functions
    complex ()
    {
        r = 0.0;
        i = 0.0;
    };
    // default constructor
    complex (double a, double b)
    {
        // init with two doubles

```

```

        r = a;
        i = b;
    }
    complex (double a)
    {
        // init with only real
        r = a;
        i = 0.0;
    }
    complex (const complex & z)
    {
        // init with another complex
        r = z.r;
        i = z.i;
    }
    complex operator + (const complex & z)
    {
        // addition
        complex a;
        a.r = r + z.r;
        a.i = i + z.i;
        return a;
    }
    complex operator - (const complex & z)
    {
        // subtraction
        complex a;
        a.r = r - z.r;
        a.i = i - z.i;
        return a;
    }
    complex operator * (const complex & z)
    {
        // multiplication
        complex a;
        a.r = (r * z.r) - (i * z.i);
        a.i = (r * z.i) + (i * z.r);
        return a;
    }
    complex operator / (const complex & z)
    {
        // divide
        complex a;
        a.r = (r * z.r + i * z.i) / (z.r * z.r + z.i * z.i);
    }

```

```

        a.i = (i * z.r - r * z.i) / (z.r * z.r + z.i * z.i);
        return a;
    }

    complex operator *= (const complex & z)
    {
        // multiplication

        complex a;
        a.r = (r * z.r) - (i * z.i);
        a.i = (r * z.i) + (i * z.r);
        r = a.r;
        i = a.i;
        return a;
    }

    complex operator /= (const complex & z)
    {
        // divide

        complex a;
        a.r = (r * z.r + i * z.i) / (z.r * z.r + z.i * z.i);
        a.i = (i * z.r - r * z.i) / (z.r * z.r + z.i * z.i);
        r = a.r;
        i = a.i;
        return a;
    }

    complex & operator = (const complex & z)
    {
        // equal

        r = z.r;
        i = z.i;
        return *this;
    }

    complex & operator = (const double &x)
    {
        // equal

        r = x;
        i = 0.0;
        return *this;
    }

    int operator == (const complex & z)
    {
        // equality check

        return (r == z.r && i == z.i) ? 1 : 0;
    }

    int operator != (const complex & z)

```

```

{
    // inequality check
    return (r == z.r && i == z.i) ? 0 : 1;
}

complex operator * (double k)
{
    // multiply by constant
    complex a;
    a.r = k * r;
    a.i = k * i;
    return a;
}

complex operator / (double k)
{
    // divide by constant
    complex a;
    a.r = k / r;
    a.i = k / i;
    return a;
}

complex & operator += (const complex & z)
{
    // +=
    r += z.r;
    i += z.i;
    return *this;
}

complex & operator -= (const complex & z)
{
    // -=
    r -= z.r;
    i -= z.i;
    return *this;
}

char *p ()
{
    // return a string of the result
    return ~(*this);
}

char *operator ~ ()
{
    // another print function
    if (PRINT_FORMAT == MATLAB)
    {
        // MATLAB
        if (PRECISION_FORMAT == HIGH)

```

```

        {
            sprintf (s, "%.16e+%.16ei", r, i);
        }
        else
        {
            sprintf (s, "%lf+%lfi", r, i);
        }
    }
    else if (PRINT_FORMAT == DERIVE)
    {
        // DERIVE
        if (PRECISION_FORMAT == HIGH)
        {
            sprintf (s, "%.16e+#i%.16e", r, i);
        }
        else
        {
            sprintf (s, "%lf+#i%lf", r, i);
        }
    }
    else
    {
        printf ("\n-E- Unknown PRINT_FORMAT = %d\n", PRINT_FORMAT);
        exit (1);
    }
    return s;
}

complex operator ! ()
{
    // conjugate
    complex a;
    a.r = r;
    a.i = -i;
    return a;
}

complex operator ^ (int pow)
{
    // z^pow
    complex a, b;
    int n;
    if (pow == 0)

```

```

{
    a.r = 1.0;
    a.i = 0.0;
    return a;
}
else if (pow > 0)
{
    a.r = r;
    a.i = i;
    b.r = r;
    b.i = i;
    for (n = 1; n < pow; n++)
    {
        b.r = (r * a.r) - (i * a.i);
        b.i = (r * a.i) + (i * a.r);
        a.r = b.r;
        a.i = b.i;
    }
    return a;
}
else
{
    // pow is negative
    pow = -pow;
    a.r = r;
    a.i = i;
    b.r = r;
    b.i = i;
    for (n = 1; n < pow; n++)
    {
        b.r = (r * a.r) - (i * a.i);
        b.i = (r * a.i) + (i * a.r);
        a.r = b.r;
        a.i = b.i;
    }
    b.r = a.r;
    b.i = a.i;
    a.r = b.r / (b.i * b.i + b.r * b.r);
    a.i = -b.i / (b.i * b.i + b.r * b.r);
}

```

```

        return a;
    }
}

double modulus ()
{
    // returns sqrt(r^2+i^2)
    return sqrt (r * r + i * i);
}

double modulus_squared ()
{
    // square modulus returns r^2+i^2
    return r * r + i * i;
}
};

// MATRIX CLASS
// The matrix class will be kept as simple as possible
// a by default will be complex.
class matrix
{
public:
    // vars
    complex M[MAX_ROWS][MAX_COLUMNS]; // matrix data
    int rows, columns;

    // functions
    matrix ()
    {
    };
    // default constructor
    matrix (int n, int m)
    {
        zero (n, m);
    };
    // default constructor, n rows, m columns
    matrix & operator = (const matrix & X)
    {
        // equal
        int r, c;
        rows = X.rows;
        columns = X.columns;
        for (r = 0; r < rows; r++)
        {

```

```

        for (c = 0; c < columns; c++)
        {
            M[r][c] = X.M[r][c];
        }
    }
    return *this;
}

complex operator  () (int r, int c) const
{
    // matrix index (row,col)
    return M[r][c];
}

complex & operator  () (int r, int c)
{
    // matrix index (row,col)
    return M[r][c];
}

void eye (int n)
{
    // initializes to the identity matrix (nxn)
    int r, c;
    for (r = 0; r < n; r++)
    {
        for (c = 0; c < n; c++)
        {
            M[r][c].r = (r == c) ? 1.0 : 0.0;
            M[r][c].i = 0.0;
        }
    }
    rows = n;
    columns = n;
}

void zero (int n, int m)
{
    // initializes a zero matrix (n rows, m columns)
    int r, c;
    for (r = 0; r < n; r++)
    {
        for (c = 0; c < m; c++)
        {
            M[r][c].r = 0.0;
            M[r][c].i = 0.0;
        }
    }
}

```



```

    }

}

rows = n;
columns = m;
}

void one (int n, int m)
{
    // initializes a matrix with all 1's (n rows, m columns)

    int r, c;
    for (r = 0; r < n; r++)
    {
        for (c = 0; c < m; c++)
        {
            M[r][c].r = 1.0;
            M[r][c].i = 0.0;
        }
    }

    rows = n;
    columns = m;
}

void show ()
{
    // print the matrix

    int r, c;
    printf ("%d x %d)", rows, columns);
    for (r = 0; r < rows; r++)
    {
        printf ("[";
        for (c = 0; c < columns; c++)
        {
            printf ("%s", ~M[r][c]);
            if (c != columns - 1)
            {
                if (PRINT_FORMAT == MATLAB)
                {
                    printf (" ");
                }
                else if (PRINT_FORMAT == DERIVE)
                {
                    printf (" ");
                }
            }
        }
    }
}

```

```

        }
        else
        {
            printf ("\n-E- Invalid PRINT_FORMAT = %d\n", PRINT_FORMAT);
            exit (1);
        }
    }
}
printf ("]");
if (r != rows - 1)
{
    if (PRINT_FORMAT == MATLAB)
    {
        printf ("; ");
    }
    else if (PRINT_FORMAT == DERIVE)
    {
        printf (" ");
    }
    else
    {
        printf ("\n-E- Invalid PRINT_FORMAT = %d\n", PRINT_FORMAT);
        exit (1);
    }
}
}
// printf("]\n");
printf ("]");
}

matrix operator + (const matrix & X)
{
    // addition

    int r, c;
    matrix A;
    if (rows != X.rows || columns != X.columns)
    {
        printf ("\n-E- Matrix size mismatch for addition!\n");
        exit (1);
    }
}

```

```

        for (r = 0; r < rows; r++)
        {
            for (c = 0; c < columns; c++)
            {
                A.M[r][c] = M[r][c] + X.M[r][c];
            }
        }

        A.rows = rows;
        A.columns = columns;
        return A;
    }

matrix & operator += (const matrix & X)
{
    // addition
    int r, c;
    for (r = 0; r < rows; r++)
    {
        for (c = 0; c < columns; c++)
        {
            M[r][c] += X.M[r][c];
        }
    }
    return *this;
}

matrix operator - (const matrix & X)
{
    // addition
    int r, c;
    matrix A;
    if (rows != X.rows || columns != X.columns)
    {
        printf ("\nE- Matrix size mismatch for addition!\n");
        exit (1);
    }
    for (r = 0; r < rows; r++)
    {
        for (c = 0; c < columns; c++)
        {
            A.M[r][c] = M[r][c] - X.M[r][c];
        }
    }
}

```

```

    }

    A.rows = rows;
    A.columns = columns;
    return A;
}

matrix & operator -= (const matrix & X)
{
    // addition
    int r, c;
    for (r = 0; r < rows; r++)
    {
        for (c = 0; c < columns; c++)
        {
            M[r][c] = M[r][c] - X.M[r][c];
        }
    }
    return *this;
}

matrix operator * (const matrix & X)
{
    // multiplication
    int r, c, i;
    matrix A (rows, X.columns);

    if (columns != X.rows)
    {
        printf ("\n-E- Matrix size mismatch for multiplication!
        LHS=(%d x %d) RHS=(%d x %d)\n", rows, columns, X.rows, X.columns);
        exit (1);
    }

    for (r = 0; r < rows; r++)
    {
        for (c = 0; c < X.columns; c++)
        {
            for (i = 0; i < columns; i++)
            {
                A.M[r][c] += M[r][i] * X.M[i][c];
            }
        }
    }
}

```

```

    }
    return A;
}

matrix & operator *= (const matrix & X)
{
    // multiplication (M = M * X.M)
    int r, c, i;
    matrix A (rows, X.columns);

    if (columns != X.rows || rows != X.columns)
    {
        printf ("\n-E- Matrix size mismatch for multiplication!\n");
        exit (1);
    }

    for (r = 0; r < rows; r++)
    {
        for (c = 0; c < X.columns; c++)
        {
            for (i = 0; i < columns; i++)
            {
                A.M[r][c] += M[r][i] * X.M[i][c];
            }
        }
        for (c = 0; c < X.columns; c++)
        {
            M[r][c] = A.M[r][c];
        }
    }
    return *this;
}

matrix operator ! ()
{
    // conjugate matrix
    int r, c;
    matrix A;

    for (r = 0; r < rows; r++)
    {
        for (c = 0; c < columns; c++)

```

```

        {
            A.M[r][c] = !M[r][c];
        }
    }

    A.rows = rows;
    A.columns = columns;
    return A;
}

matrix operator * (const complex & z)
{
    // multiply by complex number, e.g., M=X*z

    int r, c;
    matrix A;

    for (r = 0; r < rows; r++)
    {
        for (c = 0; c < columns; c++)
        {
            A.M[r][c] = M[r][c] * z;
        }
    }

    A.rows = rows;
    A.columns = columns;
    return A;
}

matrix operator * (const double &k)
{
    // multiply by real number, e.g., M=X*k

    int r, c;
    matrix A;

    for (r = 0; r < rows; r++)
    {
        for (c = 0; c < columns; c++)
        {
            A.M[r][c] = M[r][c] * k;
        }
    }

    A.rows = rows;
    A.columns = columns;

```

```

        return A;
    }

    matrix operator / (const complex & z)
    {
        // divide by complex number, e.g., M=X/z

        int r, c;
        matrix A;

        for (r = 0; r < rows; r++)
        {
            for (c = 0; c < columns; c++)
            {
                A.M[r][c] = M[r][c] / z;
            }
        }

        A.rows = rows;
        A.columns = columns;
        return A;
    }

    matrix operator ^ (const matrix & X)
    {
        // tensor product (kronecker) X (x) Y

        matrix A;
        A.rows = rows * X.rows;
        A.columns = columns * X.columns;

        int r1, r2, c1, c2;

        for (r1 = 0; r1 < rows; r1++)
        {
            for (c1 = 0; c1 < columns; c1++)
            {
                for (r2 = 0; r2 < X.rows; r2++)
                {
                    for (c2 = 0; c2 < X.columns; c2++)
                    {
                        A.M[r1 * X.rows + r2][c1 * X.columns + c2] = M[r1][c1] * X.M[r2][c2];
                    }
                }
            }
        }
    }

```

```

    }
    return A;
}

matrix transpose ()
{
    // returns the transpose version of this matrix
    matrix A;
    A.rows = columns;
    A.columns = rows;
    int r, c;
    for (r = 0; r < rows; r++)
    {
        for (c = 0; c < columns; c++)
        {
            A.M[c][r] = M[r][c];
        }
    }
    return A;
}

matrix adjoint ()
{
    // return conjugate transpose (aka dual)
    // could do: return !transpose();
    matrix A;
    A.rows = columns;
    A.columns = rows;
    int r, c;
    for (r = 0; r < rows; r++)
    {
        for (c = 0; c < columns; c++)
        {
            A.M[c][r] = !M[r][c];
        }
    }
    return A;
}

matrix dual ()
{
    // same as adjoint
    return adjoint ();
}

```



```

complex trace ()
{
    // sum of the diagonal of a matrix
    complex tr = 0.0;
    if (rows != columns)
    {
        printf ("\n-E- Cannot perform trace on a non-square matrix!\n");
        exit (1);
    }
    for (int i = 0; i < rows; i++)
    {
        tr += M[i][i];
    }
    return tr;
}

matrix modulus ()
{
    // returns the modulus by taking performing  $\sqrt{a^2+b^2}$  on each
    // element

    int r, c;
    matrix A;
    A.rows = rows;
    A.columns = columns;
    for (r = 0; r < rows; r++)
    {
        for (c = 0; c < columns; c++)
        {
            A.M[r][c] = M[r][c].modulus ();
        }
    }
    return A;
}

matrix modulus_squared ()
{
    // returns the modulus by taking performing  $(a^2+b^2)$  on each
    // element

    int r, c;
    matrix A;
    A.rows = rows;
    A.columns = columns;
    for (r = 0; r < rows; r++)

```

```

    {
        for (c = 0; c < columns; c++)
        {
            A.M[r][c] = M[r][c].modulus_squared ();
        }
    }
    return A;
}

complex unitary_error ()
{
    // returns an error value which is the sum of
    // square-difference of each element from the
    // equivalent unity matrix

    complex err = 0.0;
    int r, c;
    if (rows != columns)
    {
        printf ("\n-E- Cannot check unity matrix error since
                this is not a square matrix!\n");
        exit (1);
    }
    for (r = 0; r < rows; r++)
    {
        for (c = 0; c < columns; c++)
        {
            if (r == c)
            {
                // diagonal should be 1.0
                err += (M[r][c] - 1.0) ^ 2;
            }
            else
            {
                // non-diagonal elements should be 0.0
                err += (M[r][c] - 0.0) ^ 2;
            }
        }
    }
    return err;
}

void fill (complex z)
{
    // make each element in our matrix = z

```

```

        int r, c;
        for (r = 0; r < rows; r++)
        {
            for (c = 0; c < columns; c++)
            {
                M[r][c] = z;
            }
        }
    }

    // TO DO
    // modulus and modulus_squared (per element)
    // ADVANCED:
    // determinant
    // others?
};

matrix
HADAMARD ()
{
    matrix H (2, 2);
    H (0, 0) = 1.0;
    H (0, 1) = 1.0;
    H (1, 0) = 1.0;
    H (1, 1) = -1.0;
    H = H * (1.0 / sqrt (2.0));
    return H;
}

matrix
NOT ()
{
    // Also known as PauliX
    matrix N (2, 2);
    N (0, 1) = 1.0;
    N (1, 0) = 1.0;
    return N;
}

```

```

matrix
PHASE ()
{
    // AKA. S-Gate
    matrix P (2, 2);
    P (0, 0) = 1.0;
    P (1, 1) = complex (0.0, 1.0);
    return P;
}

matrix
V ()
{
    // SQRT(NOT) gate
    matrix V (2, 2);
    V (0, 0) = complex (1, 1);
    V (0, 1) = complex (1, -1);
    V (1, 0) = complex (1, -1);
    V (1, 1) = complex (1, 1);
    V = V * 0.5;
    return V;
}

matrix
PAULIY ()
{
    matrix P (2, 2);
    P (0, 1) = complex (0, -1.0);
    P (1, 0) = complex (0, 1.0);
    return P;
}

matrix
PAULIZ ()
{
    matrix P (2, 2);
    P (0, 0) = 1.0;
    P (1, 1) = -1.0;
    return P;
}

```

```

matrix
ONEHOT_COLUMN_VECTOR (int pos, int length)
{
    matrix X (length, 1);
    X (pos - 1, 0) = 1.0;
    return X;
}

matrix
ONEHOT_ROW_VECTOR (int pos, int length)
{
    matrix X (1, length);
    X (0, pos - 1) = 1.0;
    return X;
}

matrix
Toffoli ()
{
    // 3 qubit
    matrix T (8, 8);
    T.eye (8);
    // last two 1's on the diagonal are swizzled.
    T (7, 7) = 0.0;
    T (8, 8) = 0.0;
    T (8, 7) = 1.0;
    T (7, 8) = 1.0;
    return T;
}

//
// END
//

```

## B.2.2 UNITARY.H

```
// UNITARY.H
// Steven R. Hutsell
// June 30th, 2006
// -----
// Contains the functions and structures to create
// unitary matrices

#include "matrix.h"

#define MAX_ANGLES 64          // rule-of-thumb, for NxN matrix, MAX_ANGLES = N

// CREATE_UNITARY_MATRIX
// Returns: NxN unitary matrix
// Params:
//      N = NxN matrix
//      psi pointer to psi angle array (same for phi and chi)
matrix
create_unitary_matrix (int N, double psi[MAX_ANGLES][MAX_ANGLES],
                      double phi[MAX_ANGLES][MAX_ANGLES], double chi[MAX_ANGLES])
{
    matrix U, E_i_j;
    int i, j, ii, jj;

    U.eye (N);                // set U to NxN identity matrix

    // calculate E1*E2*...*E(N-1) by
    // calculating (E12)*(E23*E13)*(E34*E24*E14)*... the E_i_j's
    for (j = 2; j <= N; j++)
    {
        // the j param for E_i_j
        for (i = j - 1; i > 0; i--)
        {
            // the i param for E_i_j
            ii = i - 1;
            jj = j - 1;        // for indexing matrices
            // U *= E_i_j(N,i,j+1);
            E_i_j.eye (N);
            // RULES
            // (i,i)
            E_i_j (ii, ii).r = cos (phi[i][j]) * cos (psi[i][j]);
        }
    }
}
```

```

E_i_j (ii, ii).i = cos (phi[i][j]) * sin (psi[i][j]);
// (i,j)
if (i == 1)
{
    // chi only exists when i==1
    E_i_j (ii, jj).r = sin (phi[i][j]) * cos (chi[j]);
    E_i_j (ii, jj).i = sin (phi[i][j]) * sin (chi[j]);
}
else
{
    E_i_j (ii, jj).r = sin (phi[i][j]);
    E_i_j (ii, jj).i = 0.0;
}
// (j,i)
if (i == 1)
{
    // chi only exists when i==1
    E_i_j (jj, ii).r = -sin (phi[i][j]) * cos (chi[j]);
    E_i_j (jj, ii).i = sin (phi[i][j]) * sin (chi[j]);
}
else
{
    E_i_j (jj, ii).r = -sin (phi[i][j]);
    E_i_j (jj, ii).i = 0.0;
}
// (j,j)
E_i_j (jj, jj).r = cos (phi[i][j]) * cos (psi[i][j]);
E_i_j (jj, jj).i = -cos (phi[i][j]) * sin (psi[i][j]);

// debug
// printf("-D- (i,j)=%d,%d, psi=%lf, phi=%lf, chi=%lf, E_i_j=
// ",i,j,psi[i][j],phi[i][j],chi[j]); E_i_j.show();

// Now we have E_i_j
U *= E_i_j;
}
}
// U *= e^(i*alpha); // NOT UNITARY!??

return U;

```

}



## B.2.3 PARSE\_ARGS.H

```
// srhutsel
// nov 20th, 2007
// parses args

#include <string>

int
GetBoolOpt (char *str, int argc, char **argv)
{
    int i;
    int found = 0;
    for (i = 1; i < argc; i++)
    {
        if (!strcmp (str, argv[i]))
        {
            found = 1;
        }
    }
    return found;
}

int
GetIntOpt (int &res, char *str, int argc, char **argv)
{
    int i;
    int found = 0;
    for (i = 1; i < argc; i++)
    {
        // cout << "\n-D- arg " << i << " = " << argv[i];
        if (!strcmp (str, argv[i]))
        {
            res = atoi (argv[i + 1]);
            found = 1;
        }
    }
    return found;
}
```

```

int
GetStringOpt (char *res, char *str, int argc, char **argv)
{
    int i;
    int found = 0;
    for (i = 1; i < argc; i++)
    {
        if (!strcmp (str, argv[i]))
        {
            res = argv[i + 1];
            found = 1;
            cout << "\nD- ParseArgs: found " << str << " = " << res;
        }
    }
    return found;
}

```

## B.2.4 RANDOM.H

```
/*
***** RANDOM.H *****

This include file contains routines for getting a uniformly distributed
random variable in the interval [0,1], a gaussian
distributed random variable with zero mean and unity variance,
an exponential distributed random variable, and a
and a cauchy distributed random variable with

To use them (x & y declared as floats),

x = uniform(&idum);
y = normal(&idum);
z = cauchy(&idum);
u = expdev(&idum);

Notice that a pointer to "idum" is used.

The file also contains routines for getting uniformly
generated integers over specified intervals.

Before you use any of these routines, you must initialize
the long variable "idum" (declared in this file). To do this
you must include in your main program the following:

#include <time.h>

int main(void )
{
    .
    .
    .

    srand((unsigned) time(NULL));
    idum = -rand();
```

```

}

*/

#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV (1+(IM-1)/NTAB)
#define EPSILON 1.2e-7
#define RNMX (1.0-EPSILON)

long idum;

float
uniform (long *idum)
{
    int j;
    long k;
    static long iy = 0;
    static long iv[NTAB];
    float temp;

    if (*idum <= 0 || !iy)
    {
        if (-(*idum) < 1)
            *idum = 1;
        else
            *idum = -(*idum);
        for (j = NTAB + 7; j >= 0; j--)
        {

```

```

        k = (*idum) / IQ;
        *idum = IA * (*idum - k * IQ) - IR * k;
        if (*idum < 0)
            *idum += IM;
        if (j < NTAB)
            iv[j] = *idum;
    }
    iy = iv[0];
}

```

```

k = (*idum) / IQ;
*idum = IA * (*idum - k * IQ) - IR * k;
if (*idum < 0)
    *idum += IM;
j = (int) iy / NDIV;
iy = iv[j];
iv[j] = *idum;
if ((temp = AM * iy) > RNMX)
    return RNMX;
else
    return temp;
}

```

```

float
normal (long *idum)
{
    static int iset = 0;
    static float gset;
    float fac, rsq, v1, v2;

    if (iset == 0)
    {
        do
        {
            v1 = 2.0 * uniform (idum) - 1.0;
            v2 = 2.0 * uniform (idum) - 1.0;
            rsq = v1 * v1 + v2 * v2;

```

```

    }

    while (rsq >= 1.0 || rsq == 0.0);

    fac = sqrt (-2.0 * log (rsq) / rsq);

    gset = v1 * fac;

    iset = 1;

    return v2 * fac;

}

else
{
    iset = 0;

    return gset;

}

}

```

```

float
expdev (long *idum)
{
    float dum;

    do
        dum = uniform (idum);

    while (dum == 0.0);

    return -log (dum);

}

```

```

double
cauchy (long *idum)
{
    /* gives cacy r.v. centered about 0 with pdf
    f(x) = 1/(pi*(1+x*x)). */

    double z1, z2;

    z1 = normal (idum);
    z2 = normal (idum);
    if (z2 != 0.0)

```

```

        return (z1 / z2);
    else
        return 0.0;
}

```

```

int
r0n (int n)
/* returns a random integer in the interval [0,n-1] */
{
    return (int) floor ((float) n * uniform (&idum));
}

```

```

int
rnm (int n, int m)
/* returns a random integer in the interval [n,m] */
{
    int k;

    do
    {
        k = r0n (m + 1);
    }
    while (k < n || k > m);
    return k;
}

```