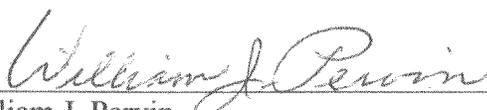


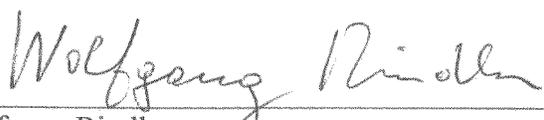
QUANTUM COMPUTATION USING GEOMETRIC ALGEBRA

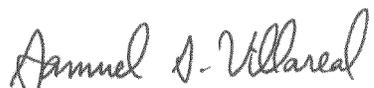
APPROVED BY SUPERVISORY COMMITTEE


Cyrus D. Cantrell III, Chair


Michael Manthey, Co-Chair


William J. Pervin


Wolfgang Rindler


Samuel S. Villareal

Copyright 2002

Douglas J. Matzke

All Rights Reserved

To all the great thinkers
who aspire to understand the
mysteries of quantum mechanics

QUANTUM COMPUTATION USING GEOMETRIC ALGEBRA

by

Douglas J. Matzke, BSEE, MSEE

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT DALLAS

May 2002

ACKNOWLEDGEMENTS

My first thank you goes to my committee chair Dr. Cyrus Cantrell, who gave me a chance three years ago when I approached him with a very vague idea about mapping Boolean logic into linear mathematics. Choosing this kind of subject (ultimately, quantum computing) is traditionally extremely mathematical and my approaching it with a very pragmatic engineering perspective must have caused him some amusement while requiring the patience of a saint. My most heartfelt thanks goes to my like-minded friend and committee co-chair Michael Manthey, whom I first met at the PhysComp94 conference, and who lives in the mountains of Crestone, Colorado. I visited Mike in Colorado for three days in the summer of 2000, where he taught me the fundamentals of geometric algebra by hand-cranking the equations. Based on that tedious manual experience, I returned to Dallas and a month later created the prototype `ga.pl` tool described in this dissertation. Once this and subsequent GA tools were available, the quantum representation practically revealed itself in another month. I am eternally indebted to Mike for enabling one of the most creative and fulfilling periods of my life. Another very big thank you also goes to Nick Lawrence of Lawrence Technologies who hired me in the summer of 2001, due to his interests in high-dimensional software technology, which enabled me to write the bulk of this dissertation in only two months. My other committee members William Pervin, Wolfgang Rindler, and Samuel Villareal were fabulous and very supportive. A very special thanks goes to my spouse Patricia, who kept our home running smoothly while I was enjoying this time-consuming, yet creative, endeavor.

QUANTUM COMPUTATION USING GEOMETRIC ALGEBRA

Publication No. _____

Douglas J. Matzke, Ph. D.

The University of Texas at Dallas, 2002

Supervising Professor: Cyrus Cantrell

This dissertation reports that arbitrary Boolean logic equations and operators can be represented in geometric algebra as linear equations composed entirely of orthonormal vectors using only addition and multiplication. Geometric algebra is a topologically based algebraic system that naturally incorporates the inner and anti-commutative outer products into a real-valued geometric product, yet does not rely on complex numbers or matrices. A series of custom tools was designed and built to simplify geometric algebra expressions into a standard sum of products form, and automate the anticommutative geometric product and operations. Using this infrastructure, quantum bits (qubits), quantum registers and EPR-bits (ebits) are expressed symmetrically as geometric algebra expressions. Many known quantum computing gates, measurement operators, and especially the Bell/magic operators are also expressed as geometric products. These results demonstrate that geometric algebra can naturally and faithfully represent the central concepts, objects, and operators necessary for quantum computing, and can facilitate the design and construction of quantum computing tools.

TABLE OF CONTENTS

ACKNOWLEDEMENTS	v
ABSTRACT.....	vi
LIST OF TABLES	x
LIST OF FIGURES	xiii
CHAPTER 1 INTRODUCTION TO QUANTUM COMPUTING	1
1.1 Problem Definition	1
1.1.1 Why Quantum Computing is Important	2
1.1.2 Understanding Quantum Computing is Difficult	3
1.1.3 Expanding the Quantum Computing Industry	4
1.2 Intended Audience	4
1.3 Improving Understanding and Building Tools	5
1.4 Summary of Results	6
CHAPTER 2 THE UNIVERSE IS A QUANTUM COMPUTER	7
2.1 Information is Primitive and Physical	7
2.2 High Dimensional Spaces	9
2.2.1 High Dimensional Metrics versus Intuition	11
2.2.2 Space and Time are Linked	12
2.2.3 Hilbert Spaces and Tensor Products	14
2.2.4 The Geometric Algebra Alternative	15
2.3 Classical Versus Quantum Computing	16
2.3.1 States and Vectors for Bits versus Qubits	17
2.3.2 Superposition and Entanglement	18
2.3.3 Classical versus Quantum Algorithms	20
2.3.4 Quantum Computers	21
2.4 Steps in Quantum Computing	21
CHAPTER 3 BOOLEAN LOGIC IN QUANTUM COMPUTING	23
3.1 Unitary Transforms and Rotations	23
3.2 Reversible Computing	24
3.3 Universal Logic and Quantum Gates	25
3.4 Toffoli and Fredkin Reversible Gates	26

3.5 Boolean Logic and Operators Using Linear Mathematics	28
3.5.1 Boolean Logic in Galois Fields	28
3.5.2 XOR Dominated Logic	30
3.5.3 Toffoli and Fredkin Gates in GF(2)	31
3.5.4 Boolean Logic is formally non-linear	33
 CHAPTER 4 GEOMETRIC ALGEBRA FOUNDATIONS	 35
4.1 Matrix versus Algebraic Notation	35
4.2 Co-Occurrence and Co-Exclusion	36
4.3 Geometric Algebra Principles	40
4.3.1 Scalars and Vectors	41
4.3.2 Bivectors and Anticommutative Multiplication	42
4.3.3 N-vectors and Mixed Rank Expressions	45
4.3.4 Number of Elements in a Geometric Algebra	47
4.3.5 Quaternions in Geometric Algebra	48
4.3.6 Inner Product Definition	50
4.3.7 Outer Product and Inner Product Examples.....	52
4.4 Geometric Algebra Tools for Boolean Logic	54
4.4.1 Boolean Logic in GA	54
4.4.2 GA Evaluator	57
4.4.3 Universal Logic Decode in GA	60
4.4.4 GA Generator	68
4.5 GA Expression Solver Tool	70
4.6 Cartesian Distance Metric	73
4.7 Computational Basis and Projectors	75
4.8 Determinants in Geometric Algebra	82
4.8 Summary of Logic in Geometric Algebra	84
 CHAPTER 5 SINGLE QUBIT REPRESENTED IN GEOMETRIC ALGEBRA.....	 86
5.1 Qubit as Co-Occurrence of Two States	86
5.2 Pseudoscalar is Spinor Operator	89
5.3 Hadamard Transform	89
5.4 Pauli Spin Matrix Transforms	90
5.5 Alternative Basis Sets	92
5.6 Qutrits for Spin-1 Particles	98
5.7 Qudit for \mathbf{H}_d Quantum Systems	99
5.8 Phase Shift Transform	99
 CHAPTER 6 MULTIPLE QUBITS REPRESENTED IN GEOMETRIC ALGEBRA.....	 102
6.1 Qubit Interaction in Quantum Register as Tensor Product.....	102
6.2 Propagation of Null States	106
6.3 Pauli Spin and Cross-Qubit Singlets.....	107
6.4 Sequential and Concurrent Application of Spinor Operators	108
6.5 Concurrent Hadamard Transform and Ebits	110
6.6 Concurrent Hadamard Transform and Alternative Bases.....	110

6.7 Entanglement Means Co-occurrence with Cross-Qubit Spinor	117
6.8 Bell Basis States are Irreversible in \mathcal{Q}_2	118
6.8.1 Discarded Phase Information in Bell States.....	119
6.8.2 No Multiplicative Inverse for $(\mathbf{S}_A + \mathbf{S}_B)$	120
6.8.3 Recursive Operator Erases Phase Information	120
CHAPTER 7 QUANTUM COMPUTING IN GEOMETRIC ALGEBRA.....	124
7.1 Single Qubit Operators	124
7.1.1 Operators as Computational Basis Vectors.....	126
7.1.2 Single qubit measurements	128
7.2 Two-Qubit Operators	131
7.2.1 Control-Not Gate for \mathcal{Q}_2	132
7.2.2 Control-Hadamard Gate for \mathcal{Q}_2	136
7.2.3 Two qubit measurement.....	138
7.2.4 Computational Basis for Two Qubits	138
7.3 Three-Qubit Operators	140
7.3.1 Toffoli Gate in \mathcal{Q}_3	140
7.3.2 Fredkin Gate in \mathcal{Q}_3	143
7.3.3 Computational Basis for Three Qubits	144
7.3.4 Bell Basis for Three Qubits.....	144
7.3.5 Three qubit measurement.....	146
7.3.6 Example Quantum Computation.....	146
CHAPTER 8 SUMMARY AND CONCLUSIONS	147
8.1 Summary of Contributions.....	147
8.2 Conclusions	154
8.3 Future Effort.....	155
APPENDIX A FREDKIN GATE USING GALOIS FIELD(2)	156
APPENDIX B PERL SOURCE CODE FOR GA.PL TOOL.....	162
APPENDIX C PERL SOURCE CODE FOR GANDG.PL TOOL	173
APPENDIX D PERL SOURCE CODE FOR GAG.PL TOOL.....	174
APPENDIX E PERL SOURCE CODE FOR GASOLVE.PL TOOL.....	177
APPENDIX F PERL SOURCE CODE FOR SHARED PERMUTATION.PL FILE	180
APPENDIX G PERL SOURCE CODE FOR SHARED GALIB.PL FILE	182
APPENDIX H SUMMARY OF NOTATION AND BASES	185
REFERENCES	187
VITA	

LIST OF TABLES

Table 3.1: Some important logic operators	25
Table 3.2: Logic Tables for Fredkin and Toffoli Gates	27
Table 3.3: Karnaugh Maps of Intrinsic Logic Operators in GF(2)	29
Table 3.4: GF(2) is Boolean Complete	29
Table 3.5: Logic Inclusive OR for {2, 3, 4} input states in GF(2)	31
Table 3.6: Fredkin Gate as Matrix Operator in GF(2)	32
Table 3.7: Toffoli Gate as Matrix Operator in GF(2)	32
Table 4.1: Product Summary	46
Table 4.2: Comparison of outer and inner products for \mathbf{G}_2	53
Table 4.3: Outer and Inner product pairs for \mathbf{G}_3	53
Table 4.4: Addition and Multiplication Tables for \mathbf{G}_2	54
Table 4.5: Binary Mapping Tables for Operators in \mathbf{G}_2	55
Table 4.6: Logic Map Reasoning Table for \mathbf{G}_2	56
Table 4.7: Boolean Logic Summary Table for \mathbf{G}_2	56
Table 4.8: Summary of Logic Operations for $\{\pm, 0\}$ in \mathbf{G}_2	62
Table 4.9: Summary of Logic Operations in \mathbf{G}_3	64
Table 4.10: Product terms $(1\pm\mathbf{a})(1\pm\mathbf{b})$ and $(1\pm\mathbf{b})(1\pm\mathbf{a})$ show phases for \mathbf{G}_2	72
Table 4.11: Difference between two Scalars for \mathbf{G}_I	73
Table 4.12: Ternary Number Label Notation for $\mathbf{G}_I = \text{span}\{\mathbf{a}\}$	74

Table 4.13: Eigenvector Summary from $E_k R_k = R_k$ for \mathbf{G}_2	78
Table 4.14: Eigenvectors E_K are major axes that form Dual Tetrahedrons in \mathbf{G}_2	79
Table 4.15: Computational Operators encoded as geometric products for \mathbf{G}_2	80
Table 4.16: Linear Independence of Decode States in Karnaugh Map for $\mathbf{G}_2 = \text{span}\{\mathbf{a}, \mathbf{b}\}$.	84
Table 5.1: Summary of Qubit State Meaning	88
Table 5.2: Summary of Spinor $\mathbf{S}_A = (\mathbf{a0} \mathbf{a1})$ Action on Qubit States in \mathbf{G}_2^-	89
Table 5.3: Alternative Reversible Basis Encodings for Qubit A in \mathbf{G}_2	94
Table 5.4: Computational Basis Measurement $(A)(1 \pm \mathbf{a0})(1 \pm \mathbf{a1})$ for \mathbf{G}_2	96
Table 5.5: Sparse Invariant States \mathbf{I}^\pm for \mathbf{G}_2	96
Table 5.6: Computation Basis Measurement Destroys Qubit Symmetry for \mathbf{G}_2	98
Table 6.1: Summary of Basis States using spinor singlets $\{\mathbf{S}_{00}, \mathbf{S}_{01}, \mathbf{S}_{10}, \mathbf{S}_{11}\}$ for \mathbf{Q}_2	115
Table 6.2: Summary of Bell/magic States times Recursive Operators for \mathbf{Q}_2	116
Table 6.3: Masked Operator States $(-1 - \mathbf{S}_A \mathbf{S}_B)(\mathbf{S}_A + \mathbf{S}_B) = 0$ for \mathbf{Q}_2	121
Table 7.1: Operator and Basis Summary for Single Qubit A in \mathbf{Q}_1	124
Table 7.2: Operator Summary for 41 out of 81 states for \mathbf{Q}_1	125
Table 7.3: Computational Products as Independent States for \mathbf{Q}_1	126
Table 7.4: Operators as Concurrent Vectors in \mathbf{Q}_1	127
Table 7.5: Reversible Basis Encoding Results in \mathbf{Q}_1	130
Table 7.6: Desired Control-Not Operator $X = +\mathbf{a1}$ where $(\mathbf{a1} \mathbf{b1}) X = -\mathbf{b1}$	132
Table 7.7: Summary of Operator $CNOT_{AB} = A_0$ for $\mathbf{Q}_2 = \{A, B\}$	134
Table 7.8: Row by Row Operator Solutions for $CNOT$ for $\mathbf{Q}_2 = \{A, B\}$	135
Table 7.9: Operator Summary for $\mathbf{Q}_2 = \{A, B\}$	137

Table 7.10: Valid State Rows for $A_0 B_0 C_0 TOF_{AB}$ in \mathcal{Q}_3 143

LIST OF FIGURES

Figure 3.1: Karnaugh Maps of Logic OR using XOR logic	30
Figure 4.1: Co-occurrence and Co-exclusion Concepts	38
Figure 4.2: Bivector Defines an Oriented Area (via right hand rule)	42
Figure 4.3: Spinor Plane	45
Figure 4.4: First seven rows of Pascal's Triangle	47
Figure 4.5: Quaternions as Bivectors in three dimensions	49
Figure 4.6: Sample Outputs from <i>ga.pl</i> Tool.....	58
Figure 4.7: Example Products and Table Controls for <i>ga.pl</i> Tool.....	59
Figure 4.8: First attempt for multivector $a + b + a b$ in \mathbf{G}_2	60
Figure 4.9: Second attempt produces two-input Logic AND in \mathbf{G}_2	60
Figure 4.10: Inversion of Logic AND Produces 2-input Logic NAND in \mathbf{G}_2	61
Figure 4.11: Two-input Logic OR and NOR in \mathbf{G}_2	61
Figure 4.12: Logic AND for $\{+, -\}$ in \mathbf{G}_3	63
Figure 4.13: Logic NAND for $\{-, 0\}$ in \mathbf{G}_3	63
Figure 4.14: Validation of logic AND for up to ten vectors using $\{+, 0\}$	65
Figure 4.15: Arbitrary Row Decode R_k for Vectors in \mathbf{G}_2	67
Figure 4.16: Addition and Subtraction of ROW 1 and ROW 3 for vectors in \mathbf{G}_2	68
Figure 4.17: GA Generator Tool Examples	69
Figure 4.18: Examples of <i>gasolve.pl</i> for \mathbf{G}_2	71

Figure 4.19: Solve for eigenvectors where $(\mathbf{e}_k)^2 = 1$	76
Figure 4.20: Solve for eigenvectors where $\mathbf{e}_k (1 + \mathbf{e}_k) = (1 + \mathbf{e}_k)$	76
Figure 4.21: Major diagonals form eigenvectors $E_{0..3}$ on left and duals $E_{7..4}$ on right for \mathbf{G}_2 ..	77
Figure 4.22: Sides of dual tetrahedrons form $P_{0..3}$ on left and duals $P_{7..4}$ on right for \mathbf{G}_2	79
Figure 5.1: Qubit as Co-occurrence of \mathbf{G}_2^- state vectors	87
Figure 5.2: Qubit State Difference decodes Classical States in \mathbf{G}_2^-	87
Figure 5.3: Illustration of Qubit States for 90° and 180° Rotations	90
Figure 5.4: On-axis and Off-axis Bases for \mathbf{G}_2^- on left and \mathbf{G}_2^+ on right.....	95
Figure 5.5: Spinor in Spin-down, spin-up, and superposition states (from IBM)	100
Figure 6.1: Non-Zero Qubit States for \mathbf{Q}_2 and \mathbf{Q}_3	103
Figure 6.2: Non-Zero Qubit States for \mathbf{Q}_4	104
Figure 6.3: Non-Zero Qubit States for \mathbf{Q}_5 and \mathbf{Q}_6	105
Figure 6.4: Pair-wise Cancellation of Null States in \mathbf{Q}_2	106
Figure 6.5: Concurrent Hadamard Transform for \mathbf{Q}_2	108
Figure 6.6: Concurrent Hadamard Transform for \mathbf{Q}_3 , \mathbf{Q}_4 and \mathbf{Q}_5 (truncated tables)	109
Figure 6.7: Square of Concurrent Hadamard $(\mathbf{S}_A + \mathbf{S}_B)^2 = (1 - \mathbf{S}_A \mathbf{S}_B)$ in \mathbf{Q}_2	112
Figure 6.8: Validation of Magic State Phase Relationship $\mathbf{M}_3 = \mathbf{B}_0 - \mathbf{S}_{00}$ in \mathbf{Q}_2	118
Figure 6.9: Summary of Bell and magic states using Pauli basis singlets	122
Figure 7.1: Full Matrix Decode of $CNOT_{AB}$ gate output for $\mathbf{Q}_2 = \{A, B\}$	134
Figure 7.2: Control-Hadamard Operator $CHAD_{AB}$ for $\mathbf{Q}_1 = \{A\}$ and $\mathbf{Q}_2 = \{A, B\}$	136
Figure 7.3: Fredkin Gate States for \mathbf{Q}_3 in Pauli Basis	143
Figure 7.4: Entangled Concurrent Hadamard States for \mathbf{Q}_3	145

CHAPTER 1

INTRODUCTION TO QUANTUM COMPUTING

The original motivation for this dissertation was to understand how the fundamental computational resources of space and time differ between classical and quantum computers. It was anticipated that if Boolean logic could be represented as a set of simultaneous linear matrix equations for both classical and quantum systems, then insightful distinctions would arise about the nature of information, space, time, concurrency, reversibility, and computation. The minimum outcome expected from this approach was that, by representing Boolean logic in a formal linear mathematical framework, it would be possible to create better logic tools that treat logic and registers in a unified way. This goal of representing Boolean logic in a linear mathematics is the conceptual glue that binds this dissertation while connecting the computer engineering perspective to the math and physics domains.

1.1 Problem Definition

Quantum computing promises the delivery of unbelievable performance compared to classical computers. This promise was encouraged by Peter Shor's 1994 discovery of a quantum-computing algorithm [34] that would efficiently factor the product of two large prime numbers. The difficulty of factoring the product of two large prime numbers is the basis for many computer encryption techniques because, independent of technology, it is computationally "difficult" to factor using any classical computer. Quantum computers

running Shor's algorithm, in contrast, can solve this type of problem relatively easily with *linear* time increase in effort for a linear growth of the problem size. This ability defines a new computational complexity class called Quantum Polynomial Time. In spite of this immense computational speedup, very few people can comprehend or apply this result because the existing mathematical notation for expressing quantum computing is derived from, and conceptually indebted to, traditional quantum mechanics which, lacking any notion of concrete mechanism is opaque. This dissertation therefore proposes an alternative mathematical representation for quantum computing.

1.1.1 Why Quantum Computing is Important

Historically, quantum computing was interesting because of the fundamental relationship between information and quantum mechanics. Visionaries such as Richard Feynman [11] and Rolf Landauer [19] worked on the problem over 40 years ago. During the last 20 years, researchers in this field justified working on this problem as preparing solutions for the predicted nanometer scaling limits of classical computers (cf. nanoelectronics). As a result of Shor's discovery, this quest was accelerated by the desire for a drastic increase in computing performance, as well as a growing interest in such uniquely quantum mechanical applications as quantum cryptography [4] and quantum teleportation [5].

My own interest in this field originates with the understanding that classical computers are intrinsically limited, not by technology or energy concerns, but rather because information encoded as energy or matter is *primarily segregated and limited* by the nature of *space* and *time* [27]. Information encoded using quantum states somehow bypasses these classical information encoding limits and creates a non-local and a-temporal *information wholism*, as

required by large scale quantum consistency. Even though information encoded as quantum states is not synonymous with energy, it must still be *physical* in order to be consistent with the second law of thermodynamics and black hole mechanics [31]. Quantum computing is the ideal place to study the core relationship between information and the classical computing resources of space and time. True computational concurrency is not possible except in the quantum domain because causal spacetime segregates classically encoded information.

1.1.2 Understanding Quantum Computing is Difficult

Classically trained computer engineers and programmers seem ill equipped to participate in the quantum computing revolution unless they have been educated in the specialized math and physics of quantum computing. The quantum computation rules are so different from classical computing that currently this field is inhabited predominantly by physicists and mathematicians. Unfortunately, many of these researchers lack the sophisticated and well developed programming skills to build the development environment tools needed to program complex and unintuitive quantum systems. This is a classic skills-mix problem and the goal is to provide the tools needed by designers of quantum computing systems.

The quantum rules are so bizarre and therefore counterintuitive that most traditional classical programming techniques do not apply because they depend on implicit assumptions about spacetime, information representation, energy, and causality that are inappropriate in the quantum domain. Additionally, many new concepts such as reversibility, superposition, entanglement, high dimensional spaces, and true concurrency add additional confusion to the already unfamiliar mathematical terrain [25].

1.1.3 Expanding the Quantum Computing Industry

Since quantum computing is both fundamental and important, the computing industry needs more engineers and programmers to participate in its understanding and development. The traditional computer industry has an inverted developers' pyramid within which a relatively small percentage of designers build hardware while many more people build tools such as editors, assemblers, linkers, libraries, compilers, debuggers, operating systems, networks, and other infrastructure utilities. The next larger group of people builds application-specific software on top of the hardware and software infrastructure. Finally, a very large group of customers use these applications and are supported by sales and service personnel.

This same development progression must occur in order to make quantum computing a success, which of course assumes it is sufficiently general in purpose to support important applications or that it acts as the transition technology when conventional electronics scales into the nanoelectronic region. The diverse occupations listed above will not be primarily staffed by physicists and mathematicians because of an inappropriate skills mix. Engineers and programmers must somehow get involved in the quantum tools and programming effort and this can only be accomplished if a mathematical bridge [21] exists to facilitate the building of the required software tools and applications. Such an approach to quantum computing is proposed in this dissertation, using an already defined algebra.

1.2 Intended Audience

This dissertation is primarily intended for classically trained computer engineers and programmers without previous exposure to quantum computing or its mathematical underpinnings (usually using Hilbert space matrix notation). The goal is to gently define,

using geometric algebra, the concepts and mathematical representations needed for understanding quantum computing without any prior background except training in conventional computer design and programming. Despite this approach and targeted audience, another goal is to be faithful to physics, reproducing the usual quantum physics descriptions of devices and phenomena in a new language. The ultimate goal is to provide a common mathematical language and tool framework that acts as a bridge between engineers and physicists.

1.3 Improving Understanding and Building Tools

Quantum mechanics in Hilbert space is traditionally described using Dirac's "bra-ket" notation $\langle | \rangle$, where the "ket" $| \rangle$ is a column vector and its transpose the "bra" $\langle |$ is a row vector. This matrix notation is compact and terse, and represents a language and conceptual barrier to engineers and programmers in the same way that physicists and mathematicians may not comprehend the latest computer definitions, programming languages, or concepts. Furthermore, the Hilbert spaces usually employed in quantum mechanics rely on complex numbers, which also creates a representational mismatch because of the semantic gap between complex numbers and the binary 1's and 0's of computers.

Geometric algebra [16] provides a mathematical alternative that is isomorphic to Hilbert spaces in expressing the high-dimensional spaces that inevitably occur, but using only real (rather than complex) numbers. Additionally, geometric algebra uses an algebraic rather than a matrix notation, similar to Boolean algebra, thereby making it more palatable to engineers and programmers. Real numbers more closely match the traditional Boolean logic representation and mindset of classical programmers, thereby facilitating understanding and

encouraging the creation of customized tools for quantum computing. Since geometric algebra is considered to be a universal mathematical language for physics and engineering [21], it should also be useful for exploring quantum computing. The geometric algebra required for this research is described in Chapter Four.

1.4 Summary of Results

This dissertation demonstrates that arbitrary Boolean logic equations and operators can be represented in geometric algebra as linear equations composed entirely of orthonormal vectors using only addition and multiplication. A set of tools is defined, built, and described that generate, simplify, evaluate, print, and solve Boolean logic equations written as geometric algebra expressions. Quantum bits (qubits) and quantum registers are expressed using this representation and the new tools are used to discover, define, and explore the states, operators, and intrinsic properties that are important to quantum computing. These results demonstrate that geometric algebra appears to faithfully capture central quantum computing concepts, enabling deeper insight into the mechanisms behind quantum computing as well as the direct creation of efficient quantum computing development tools.

CHAPTER 2

THE UNIVERSE IS A QUANTUM COMPUTER

The universe is really a very large extended quantum simulation [12] whose outcome is the classical world view we experience around us. This fact is accepted because the inverse scenario is impossible, since classical computers cannot efficiently simulate quantum mechanical systems. Even empty space and black holes are filled by invisible, nonphysical quantum states (or zero-point energy), which represent a very high-dimensional quantum foam or ether that can only be observed as a small projection into our 4D spacetime.

This accepted understanding means that quantum states are *protophysical* [28], since they are more fundamental than classical entities such as space, time, energy, or matter. This idea agrees with the “big bang” theory of cosmic evolution in which quantum states existed before classical features appeared. Since primitive quantum states encode (or *are*) information, information is primitive and the start of the universe represents a quantum “bit bang” [23].

2.1 Information is Primitive and Physical

IBM Fellow Rolf Landauer lectured for years that “information is physical” [20]. In this he opposed the prevailing thought during most of this century that computation took a minimum theoretical amount of energy proportional to the thermal noise of a molecule at room temperature [26]. Landauer argued that the irreducible cost of computing was not the

computation itself, but rather was due to *erasing* information, which releases a small amount of energy consistent with thermodynamics. With the efforts of others, Landauer's Principle ultimately led to the field of reversible computation [6][13].

Landauer's Principle requires that information be conserved in order to be consistent with the 2nd Law of Thermodynamics, even when the information is encoded as quantum phase states. Bekenstein and Schiffer ultimately showed [2][32][33] this physical reality of quantum-encoded information by demonstrating that, when a quantum state is thrown into a black hole, the surface area size increases by the minimum discrete amount of one bit, which is approximately Planck's area = h^2 , where Planck's constant is $h = 6.626 \times 10^{-34}$ Js.

Black holes are therefore *bit buckets*, due to the combined laws of thermodynamics, quantum mechanics, gravity, and information theory. The surface area of the black hole is known as its event horizon (or entropy, measured in bits), and represents the boundary between our physical 4D spacetime and the quantum dimensions inside (that form the singularity). Since time is infinitely dilated and space infinitely contracted within a black hole, the entire perspective inside is experienced as a single spacetime point, exactly like the observer frame of a photon. This understanding intimately ties the physicality of information, encoded as high-dimensional spaces, to physical observables such as space, time, energy, and gravity.

This relationship between the physicality of information and physical observables can be examined from a computational complexity perspective rather than the above classical energy viewpoint. For example, take an NP-complete computer algorithm and scale the problem so that solving it using today's most advanced fixed size computer technology

would take longer than the age of the universe. This spatially limited algorithm thus has unlimited temporal extent. Now temporally restrict this solution to a relatively small time T and compute the number of processors N working in parallel to solve it in that time T . The N processors each of mass m communicate using light traveling at the speed of light c , thereby roughly defining a sphere of diameter $d \approx T * c$. The total mass $M = N * m$ of this computer must fit into the volume of this sphere, and as the problem size increases, the mass grows exponentially faster than its volume and exceeds the black hole mass density limit. Even though a classical computational solution therefore *cannot* be built, this class of problem *can* be solved using quantum computers. Therefore, even though quantum states are physical, they must have a completely different relationship to space, time, and information than computers made with mass, information encoded as energy, and computation responsive to classical causality rules.

2.2 High Dimensional Spaces

Computation is *smarter* the greater the number of independent degrees of freedom because more dimensions mean more information is *local* simultaneously. The unit distance locality metric (number of neighbors at unit distance away from any grid point) in a discrete cellular grid is simply $2n$, where n is the number of orthogonal dimensions. As will be formally shown later in this chapter, every qubit represents two separate degrees of freedom (i.e. dimensions), so large quantum systems represent many more degrees of freedom than are possible in a 3D physical space.

When information is non-adjacent, it must be moved if it is to be used in a later decision.

Based on this thinking and assuming scaling is at the maximum, it is *impossible* to simulate

any number of dimensions in fewer than that number of dimensions without the appearance of some kind of anisotropic behaviors, because the locality metric is no longer uniform. Quantum systems can be *smarter* than classically constrained computers because they possess a locality metric proportional to the number of degrees of freedom and this space-like locality represents a primitive computational resource. This is also the primary reason why 3D classical computers cannot efficiently simulate high-dimensional quantum systems.

The key to understanding quantum computing is to learn about the physical reality of quantum state information encoded in high-dimensional spaces. Mathematically speaking, quantum bits (qubits) are represented as two orthogonal vectors, so each dimension can change independently of the other, thus allowing all possible combinations of these vector states, and hence capable of expressing both classical and superposition phase states. Many qubits entangled in a quantum register thus create an exponentially larger number of linearly independent dimensions, due to the tensor product operator combining the orthogonal vector states of every qubit. As operators evolve the overall system state, the independent states become constrained due to the erasure of information but still represent the same large number of dimensions. In contrast, the state space size does not exponentially expand for classical systems.

The concept that mathematically ideal quantum states actually encode information is foreign to most computer scientists and engineers, since historically they have been taught the now obsolete idea that *energy* is synonymous with information encoding (except in classical communications systems). These mathematically ideal dimensions must however be physical

in order to have a tangible effect on the 4D universe. The quantum states and their associated information metric constitute the bookkeeping method for tracking bits, just as the energy metric is the bookkeeping method for the 2nd Law of Thermodynamics. Quantum systems represent a particle/wave duality and also an energy/information duality with separate ledgers for energy and state information. Erasing information represents a transfer $I \rightarrow E$ between these two ledgers, whereas adding information means the transfer $E \rightarrow I$.

2.2.1 High Dimensional Metrics versus Intuition

High dimensional spaces are difficult to understand because the intuition we develop for one, two and three dimensions does not scale to n dimensions where say $n \gg 20$. For example, metrics based on the Cartesian distance (square root of the sum of the squares) between two points in an n -dimensional *unit hypercube* have the following properties: as n becomes large, the *expected mean or standard distance* between any two random points grows as $d = \sqrt{n/6}$ (as one might expect), but the *standard deviation* of a random collection of points at the distance d approaches the constant $s = \sqrt{7/120} = .2415$ [22]. For example for $n = 3$, then $d = .707$ and *most points lie at distance* $.707 \pm .24$ but for $n = 30$ then $d = 2.23$ so *most points lie at distance* $2.23 \pm .24$.

This experimentally verified result is unexpected, and therefore unintuitive, because it means that two randomly chosen points are *likely to be the standard distance apart* from each other and it is statistically *highly unlikely* to find randomly chosen points closer or farther away.

This property makes high dimensional spaces useful as content addressable or error correcting memories [17]. Another unintuitive property is that the volume of an n -dimensional hypercube lies near its surface, similar to the event horizon surface area of a

black hole. For these reasons, much of the work in high-dimensional spaces is done using purely mathematical techniques. Few engineers or programmers have much experience or insight regarding the meaning or practical uses of these spaces.

2.2.2 Space and Time are Linked

Einstein showed that space and time are linked together in an integrated 4D spacetime framework. This classical spacetime must emerge from the high-dimensional framework of quantum mechanics, as described in Wheeler's "It from Bit" paper [36]. Therefore the quantum domain must have a different kind of primitive temporal framework, if for no other reason than it exists in a completely different spatial environment. Conceptually, this proto-temporal framework is identical to the synchronization tokens and mechanisms used by asynchronous logic [29] and the work of Manthey [23] (not to mention that of C.A. Petri in the 1960s and 70s). The understanding of quantum time is very important to computation because time is a major computational resource related to how a computation *synchronizes* and *changes* state.

This idea can be easily envisioned by adapting a thought experiment devised to illustrate relativistic time dilation. In a variation of that famous thought experiment, we postulate twin astronauts, one of whom stays on earth while the other takes a ride in a spaceship for 20 years at 0.995 the speed of light. Each astronaut took one of two identical computers, both of which are tasked with solving the same set of 200 problems, each of which is estimated to take 10% of a year to complete. For the humans involved, the twin staying on earth ages 20 years. Due to time dilation, his brother ages only 2 years during the 20 earth years and arrives home 18 years younger than his twin brother. Slowing down human aging is generally

labeled as *good*. For the twin computers, the outcome is interpreted somewhat differently. The computer on earth finishes all 200 problems just as the spaceship returns. The onboard computer however has finished only 20 problems. Slow computation is considered to be *bad*. Time dilation therefore is considered to be *good* for humans but *bad* for computers because the key computational resource of classical time slows down.

Quantum superposition exhibits a completely different temporal abstraction than time dilation because it represents an *ideal concurrency*, effectively outside of conventional time. This concurrency is a direct result of the independent spatial degrees of freedom, since a countable but practically infinite locality metric exists for high-dimensional spaces. If all independent states can simultaneously interact due to superposition, then no delay is required to complete a piecewise decision process. Since these states also have no mass, they also need no spatial separation, and are similar to a small singularity as found in a black hole.

Ideal concurrency due to superposition *must* be valid for quantum states, otherwise placing a detector *after* one of the slits in the twin-slit experiment could not retroactively create the *self-consistent* effect of the single photon passing only through the other slit. Quantum energy eigenstates act like a memory constraint mechanism because they exist outside classical time (or persist through time) and all possible interaction sequences must be self consistent. True superposition concurrency with unlimited local spatial extent (a la Einstein-Podolsky-Rosen - EPR) [3] is possible only for quantum-encoded states and is impossible for classically encoded ones.

Quantum-encoded states directly address both primary computation resources (i.e. space and time) by allowing an unlimited locality metric, thus eliminating the need for the sequential integration of partial results. Quantum systems are always self-consistent because they act as a *whole*, which again is only possible with the unlimited spatial locality of a high-dimensional space, thereby enabling true concurrency (cf. co-occurrence) [23]. Interaction of disjoint finite sets of dimensions (operators and resulting co-exclusions [23]) creates a synchronization-based proto-time from which classical time and energy metrics ultimately emerge. Co-occurrence and co-exclusion will be discussed further in Chapter Four.

2.2.3 Hilbert Spaces and Tensor Products

Matrix notation is a compact way of expressing simultaneous linear systems, and has been extensively used. The original goal of this dissertation was to map the Boolean logic of traditional computing into matrix notation in the hope of gaining mathematical insight and formalism for classical computing and parallelism. Since quantum computing is also represented this way, though as reversible linear matrices and unitary operators in high-dimensional Hilbert space, this effort was quietly extended to include the quantum computing notion of true concurrency.

Traditionally, qubits and quantum systems have been described [15] using complex-valued n -dimensional Hilbert spaces, denoted as \mathbf{H}_n . A single qubit, denoted as \mathbf{H}_2 , is the sum of two complex-valued basis variables $|\mathbf{y}\rangle = \mathbf{a}|0\rangle + \mathbf{b}|1\rangle$ using Dirac's bra-ket notation, where

$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, $\{\mathbf{a}, \mathbf{b}\} \in \mathbf{C}$ and $|\mathbf{a}|^2 + |\mathbf{b}|^2 = 1$. The complex constants $\{\mathbf{a}, \mathbf{b}\}$ are called

the probability amplitudes and their square is the probability of that state's occurring.

Operators are square matrices that allow linear combinations of these basis vectors.

Quantum registers combine $q = n/2$ qubits to form larger Hilbert spaces $\mathbf{H}_{2^{n=2q}}$ using the

tensor product (\otimes) operator to form $|\mathbf{y}\rangle = |\mathbf{y}_1\rangle \otimes |\mathbf{y}_2\rangle \otimes \dots \otimes |\mathbf{y}_q\rangle = \sum_{i=0}^{2^q-1} \mathbf{a}_i |i\rangle$ with

orthonormal binary basis vectors $B = \{|i_{base2}\rangle \mid i \in 0 \leq i < 2^q\}$ and complex constants

$\{\mathbf{a}_i\} \in \mathbf{C}$ with the unitarity constraint $\sum_{i=0}^{2^q-1} |\mathbf{a}_i|^2 = 1$. The number of states grows as the power

2^q of the number of qubits due to the tensor product operator. While the above definitions

are pleasantly concise to mathematicians, they are terse and enigmatic to engineers and

programmers. Complex-valued states, bra-ket notation, and the tensor product are all

notational ideas that are much simpler to describe and understand using geometric algebra.

The focus of this dissertation is therefore to provide an alternative and more palatable

mathematical language for understanding, representing, and exploring quantum computing.

2.2.4 The Geometric Algebra Alternative

Geometric algebra relies on real numbers rather than complex numbers. This is topologically

easy to understand, because a complex number is really a point in a two-dimensional plane,

and can be directly represented in geometric algebra using two orthogonal real-valued

vectors and some scalars.

Matrix algebra is non-commutative and so is geometric algebra. The big difference is that

geometric algebra uses an algebraic rather than a matrix notation. Non-commutativity means

that multiplication order is important, since right side multiplication produces a different result from left side, such that $\mathbf{a} * \mathbf{b} \neq \mathbf{b} * \mathbf{a}$. This is obvious for matrix operators that are not square. Some matrices A have no multiplicative inverses A^{-1} defined for them and this is also true for some expressions in geometric algebra. In both algebras where A^{-1} is undefined, the determinant, which is used in the denominator for computing the multiplicative inverse, is zero; that is $\det(A) = 0$ so $1/\det(A) = \infty$, i.e. is undefined.

The high dimensional geometric algebra is implicitly non-commutative without using matrices because its geometric product is the sum of the inner (or dot) and outer (or wedge) products [16]. The *outer product* is non-commutative, such that $\mathbf{a} \wedge \mathbf{b} = -\mathbf{b} \wedge \mathbf{a}$ for orthogonal vectors $\{\mathbf{a}, \mathbf{b}\}$. The *geometric product* will be shown to be equivalent to the tensor product when using the correct representation for qubits, complex numbers are replaced by the geometric product of vectors, and algebraic notation replaces matrices and “ket”s. All of these simplifications make a geometric algebra approach to quantum computing rather more palatable to engineers, but nevertheless equivalent (and perhaps superior) to the Hilbert space approach. Chapter Four is devoted to geometric algebra fundamentals.

2.3 Classical Versus Quantum Computing

Several definitions used in quantum computing have a slightly different contextual meaning than their classical counterparts. Also, quantum computing contains ideas that are not present in classical computing. This section is an introduction and guide to these differences.

2.3.1 States and Vectors for Bits versus Qubits

A *bit* is a representation of two possible *binary states* denoted as $\{-, +\}$ or $\{\text{false}, \text{true}\}$ but is often expressed as the implementation-specific binary values $\{0, 1\}$. In the following chapters, please know that the $\{-, +\}$ binary representation in particular *does not* imply the values $\{0, 1\}$. A classical bit can be mathematically expressed as a *vector* because it represents a single independent degree of freedom. The primary property of the binary states of a classical bit is they are *mutually exclusive* and inverting either state produces the other. These are the only possibilities in a classical world.

Multiple bits can be represented as multiple, orthogonal bit-vectors “concatenated” together, which means that their bit values can change independently of each other, resulting in 2^n possible state combinations. The state thus produced by concatenating multiple bit-values can be thought of as a numeric binary address representing one of the corners of an n -dimensional unit hypercube. Since each vector’s binary states are mutually exclusive, the concatenated states also exclude all others, so effectively the vector address means only a single corner of the hypercube can be represented at a given instant. Inverting this address (i.e. composite state) is equivalent to addressing the diagonally opposite corner of the hypercube. Classical bits and states are clearly separate but closely related concepts.

Quantum bits or qubits exhibit a slightly different relationship between vectors and states. A classical bit uses a *single* vector to represent its two mutually exclusive states. In contrast, a qubit must additionally handle the physically observed cases when the two states are simultaneously *both on* or *both off* so the representation requires two independent binary-

valued vectors to encode those four possibilities. The simplest way to encode this is to assign each quantum state its own unique vector, which makes the *states independent of each other*.

As a result, vectors represent *bits* in the classical case but *states* in the quantum case.

Under these conditions, an individual *state vector* has the mutually exclusive modes of either *on* = “+”, or when inverted, *off* = “-” = *not on*. Each state vector can be independently placed in the mutually exclusive states of either *on* or *off*. In order not to confuse the use of the word “state,” when classical states are mapped to individual vectors in qubits, the word *state* will refer to one of the unique combinations of a set of orthogonal *vectors*, and this terminology is independent of whether the states are mutually exclusive or not.

2.3.2 Superposition and Entanglement

In any case, the qubit representation allows a qubit to represent the possibility of being in the classical state “+” and classical state “-” *simultaneously*, which is impossible for classically encoded bits. The qubit state to simultaneously express two mutually exclusive classical states is called *superposition*. Superposition is initially difficult to understand because it simultaneously represents both True *and* False, both Catholic *and* Protestant, or both Republican *and* Democrat. This idea is very hard to grasp coming from a traditional computing perspective where these states are historically regarded as obviously mutually exclusive. Superposition violates the Aristotelian law of the excluded middle.

The best way to think about such non-mutually exclusive states is to envision *sets of states*, where each state is represented by its own vector. The traditional classical bit-encoding can then be represented when the two vectors have opposite semi-redundant values such as

{Catholic=on, Protestant=off} which means the same as {Catholic, not Protestant}.

Additionally its inversion can be expressed as {not Catholic, Protestant}. Alternatively, the superposition states are represented when both values are the same, resulting in the set {Catholic, Protestant} or its inversion {not Catholic, not Protestant}. A qubit can be initialized to a particular starting state using standard quantum operators (Inversion and Hadamard).

The tensor product operator (\otimes) is now easy to understand using this *state set* model because combining the two qubits {C=Catholic, P=Protestant} \otimes {R=Republican, D=Democrat} produces all combinations of these sets, which is {C R, C D, P R, P D}. Quantum operators applied to this product set would simultaneously work on each combination. The orthogonal vectors that represent these sets of concurrent states for each qubit are *not concatenated* together like classical bits but written as the linear *sum* of vectors (C + P) and (R + D). Vector-state addition represents true concurrency (cf. Chapter 4) and enables superposition of qubit states, including combination generation using the tensor product. These ideas will be expressed in a formal and obvious manner in the geometric algebra notation. The wholistic properties (or coherence) of superposition and entanglement are easily upset due to noise interaction from the environment.

Some sets of states can be *separated* or factored back to the original constituent sets. For example, combining qubit states {red} \otimes {car, truck} produces the *entangled* set of states {red car, red truck} which is *separable* back into the sets {red} and {car, truck}. But the *entangled* set {red car, black dog} is *not separable* [15] because no sets of smaller dimension

exist that, when combined using the tensor product combination operator, can produce that result. Inseparable entanglement will be precisely shown later to be caused by information erasure. Inseparable entanglement does not exist in classical computation systems because it represents a specialized simultaneity constraint with information erasure. This subject will be discussed in detail in Chapter Six on multiple qubits.

2.3.3 Classical versus Quantum Algorithms

Classical computation and state machines are built on the causal state-to-state transition model known from finite state machines (FSM). FSM implementations assume that states are mutually exclusive, *measurable*, and most use a many-to-one state-to-state mapping, making them *irreversible*. Quantum algorithms work with *sets* of states at one time and must only allow *unitary* or one-to-one mappings in order to be *reversible* [24]. All measurements destroy information (i.e. a projection operator) so are also not reversible. This dissertation shows that GA projection operators without multiplicative inverses are not reversible and therefore cause information erasure. All of these new concepts will be defined and discussed in more detail in Chapter Three.

Two major classes of quantum algorithm exist. The first class is the implementation of traditional classical algorithms using quantum implementations of reversible logic gates to be discussed later, such as *Fredkin* and *Toffoli* gates [24]. Quantum Finite State Automata (QFSA) also fall into this category. These algorithms may be useful for implementing traditional logic when the nanoelectronics scaling limit is reached; they only use classical states and offer no computation gain due to superposition or entanglement.

The second class of quantum algorithm outperforms its classical counterpart by using superposition and entangled states. Shor's factoring algorithm [34] (which uses the *Quantum Fourier Transform* (QFT)) and Grover's quantum search algorithm [14] are the primary known algorithms in this class. A very useful outcome of the present research would be to create programming tools to aid in designing both classes of algorithms.

2.3.4 Quantum Computers

A physical implementation of the currently largest (seven qubit) quantum computer was reported by the Los Alamos National Labs (LANL) in March 2000 [18], only 18 months after the first three qubit machine was reported (using Nuclear Magnetic Resonance (NMR) and customized molecules). Early in 2002, IBM announced Shor's algorithm factored the number 15 into factors three and five, running on a seven qubit NMR quantum computer. Nanodots (single 3D-confined quantum states) and nanomolecules (4-5 coupled nanodots) are the long-term hope for quantum computers because they represent the continuation of contemporary semiconductor-based scaling methodology into the quantum computing domain.

2.4 Steps in Quantum Computing

The earlier sections of this chapter tried to show how fundamental quantum information principles are to the structure of the universe. Effectively, the universe is an extremely large quantum simulation from which classical entities emerge. This section describes the steps required to harness this innate ability to produce useful computation and answers.

The coherence of a quantum state is easily affected by noise from the environment, so special care must be taken to prepare and maintain the state to be used for computation. Each

quantum computation follows an algorithm or recipe of operations, starting with initializing the qubits to known states. At the end of evolving the system state using a programmed sequence of operators, the final evolved state must be *measured* to produce the answer, which has the side effect of destroying the coherent information represented in the evolved state. This measurement process is called a *projection*, because the high dimensional states are reduced to a single classical bit vector result inside the measurement apparatus.

Anywhere during this process, noise can also modify the system state, acting like an unwanted operator or measurement. For these reasons, redundancy of state with error correction techniques has been successfully applied to the evolution of quantum states. Designing quantum computers and algorithms is still in its infancy, but significant engineering progress has been made over the last ten years.

Quantum theory is very strong due to its very precise mathematical models. Unfortunately, Manthey and I believe the overall understanding and intuition about quantum computing is far less mature than generally thought. It is also not very widespread due to the various complexities involved. The dream is to make quantum computing a natural extension of the knowledge, techniques, and tools used in classical computing, resulting in a much larger participation from the traditional engineering and programming communities.

This concludes the gentle introduction to quantum computing concepts.

CHAPTER 3

BOOLEAN LOGIC IN QUANTUM COMPUTING

This chapter introduces the ideas that information, reversible computing, non-erasure, and unitary transforms are equivalent concepts, due to thermodynamic principles. Rolf Landauer first demonstrated in the 1960s that any information erasure leads to a logical irreversibility, which must ultimately result in a physical irreversibility and thus ultimately impacts the amount of heat generated by a computation. Unitary transforms are defined as a set of one-to-one state mappings that *are* reversible [24], whereas the many-to-one state mappings used by conventional logic OR and logic AND gates are logically irreversible, and most *classical state machines* have multiple ways of reaching a state (many-to-one) so are also not reversible. All these topics are very important to realizing logic via quantum mechanics.

3.1 Unitary Transforms and Rotations

The only useful operations on the quantum states of an isolated system are unitary transforms (expressed as unitary matrices in Hilbert space). Since unitary transforms preserve the norm of the inner product of quantum states, they effectively are performing rotations on those states. In fact, the only unitary operation possible on quantum states is to “rotate” them.

Rotation is inherently unitary because it uniformly transforms *all* states, which is equivalent to a one-to-one mapping, or equivalently a change in observer perspective. Since any rotation is unitary, it can be reversed and thus implies that none of these states are truly destroyed or

erased. Rotations are easily expressed using linear rotation groups [7]. Unfortunately, the formal mathematics for describing unitary transforms in complex-valued Hilbert spaces is quite involved. Therefore a formal and rigorous explanation will be provided in Chapter Four using the much simpler rotational formalism of geometrical algebra.

3.2 Reversible Computing

The concept of mapping states in a one-to-one and reversible fashion is identical to the unitary transforms used in quantum mechanics. This equivalence necessitates the exploration of a more concrete definition of reversible logic, expressed in the mathematical context of a linear system. Once some linear system is in place, the two well known three-input reversible gates, Fredkin and Toffoli, can be expressed. *Note that some one-to-one transforms erase information and are thus irreversible.*

Again, the usual mathematics of complex Hilbert spaces is too complex to make this simple point regarding the relationship between reversible Boolean logic and linear systems.

Therefore, the remainder of this chapter covers the main concepts of reversible logic and quantum gates *without* relying on Hilbert space mathematics. In order to demonstrate how linear systems must adapt to encompass logic operations (both reversible and irreversible), real vector linear algebras called Galois Fields (GF) will be introduced and used to express Boolean logic operators. The forms of these Galois Field expressions are similar to those developed later in geometric algebra and constitute an introduction to the key issues.

3.3 Universal Logic and Quantum Gates

A set of operators is “universal” or “Boolean complete” if any Boolean logic function can be represented using combinations of only those operators. The traditional two-input NAND logic gate (with one output) is universal even though it is not reversible. Likewise the two-input NOR gate is also universal, but its near-cousin the eXclusive-OR (XOR) gate is not universal by itself. Logic inclusive OR and logic AND gates are also Boolean *incomplete* by themselves unless inversion is available using logic NOT or logic XOR gates.

Table 3.1: Some important logic operators

NAND	false	true
false	true	true
true	true	false
Is Boolean Complete		

NOR	false	true
false	true	false
true	false	false
Is Boolean Complete		

XOR	false	true
false	false	true
true	true	false
Not Boolean Complete		

A universal one-input gate clearly does not exist, and likewise no two-input gates exist that are both universal and reversible. A minimum of three inputs and three outputs are required to define a classical universal reversible gate, cf. Section 3.4.

The rules regarding universality and reversibility change for quantum gates because all quantum systems are unitary and thus are inherently reversible. The challenge is to use this kind of system to produce classical Boolean logic and other useful computation. The smallest quantum system is a single qubit containing two interlocked orthogonal state vectors. This quantum *phase gate* is *q-universal* [1] because it can place the qubit in an arbitrary phase state. The *Hadamard gate* is equivalent to a phase gate with a preset 90° angle. Consequently

the Hadamard gate is not q-universal. Hadamard gates switch qubits back and forth between classical and superposition phases. A single qubit cannot represent any classical Boolean logic operation except inversion.

For systems with two qubits, conditional operators can be defined where, when the *control* qubit is “True”, the operation on the other *data* qubit is conditionally performed, else nothing happens. The naming convention for conditional gates is *control-<operation_name>*, where the most common is the *control-not* gate. The *control-not gate* conditionally inverts the data qubit when the control qubit is “True,” and so is equivalent to an asymmetric XOR logic operator. If the control is not in a classical state, the data bit is placed into a superposition state. The control-not gate is not logic universal. Similarly, the phase gate equivalent for two qubits is called the *control-phase gate*, which is q-universal for setting any quantum state. The control-V [10] gate is equivalent to a conditional *control-Hadamard* gate and is not logic universal. No two-qubit system (or logic operator) is Boolean complete, even though the two-input classical NAND and NOR gates are universal.

3.4 Toffoli and Fredkin Reversible Gates

The first Boolean complete classical logic operators arise using three classical bits or three qubits. The reversible Toffoli and Fredkin gates are the only two universal classical logic gates possible [24] and require three inputs and three outputs. Table 3.2 shows the logic tables for Fredkin and Toffoli gates with classical inputs {a, b, c} and outputs {A, B, C}.

Table 3.2: Logic Tables for Fredkin and Toffoli Gates

c	b	a	C	B	A
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	1	1	1
Fredkin Gate c=control					

c	b	a	C	B	A
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0
Toffoli Gate c=b=control					

The Toffoli Gate is called a *control-control-not gate*, where the two control inputs must both be “True” to conditionally invert the third data qubit. The input values nevertheless pass through to the outputs so nothing is ultimately erased. Other useful gates, defined as the *control-control-phase* and *control-control-Hadamard*, are also possible and will be defined later using geometric algebra. Thus, control gates can be defined with any number of inputs.

The *Fredkin gate* is similar to a small two-pole relay with a single control line and two data lines, which is functionally equivalent to a 2x2 crossbar switch. If the control line is “True,” the switch is in the “bar” state and the data values just pass thru. If the control line is “False,” the switch is in the “cross” state and the data values are routed to the opposite data output. All input values pass thru to an output, so this gate conserves the number of 0s and 1s between input and output pins. Using ballistic computing ideas where a logic “True” value is represented by the presence of a ball, the Fredkin gate is thermodynamically the most passive reversible gate since no balls need be created or destroyed. Assuming a steady supply of spare balls (source) as needed then destroying a ball (sink) is the same as erasing information.

The Toffoli and Fredkin operators are important because they link universal classical logic to unitary systems, and as well, demonstrate reversibility in classical computing systems. These bridge operators will be used in Section 3.5 below to illustrate how Boolean logic operators cannot be directly represented in linear systems composed exclusively from the input vector set. This concept is necessary in order to explain why unitary quantum computation requires inflated linear spaces (using the tensor product) to represent reversible logic operators.

3.5 Boolean Logic and Operators Using Linear Mathematics

Much of physics models can be expressed using simultaneous linear equations traditionally represented as matrices. Therefore, the first step in integrating computation and physics requires expressing Boolean logic functions as operators in a system of linear mathematics. This approach permits the same robust mathematical techniques from physics to be applied to computational science. Since any linear algebraic system that is Boolean complete should work, the relatively simple linear algebraic system called Galois Fields [30] is chosen, which is classical logic universal. This approach produces the unexpected result that Boolean logic operators can only be expressed when embedded into an expanded linear space.

3.5.1 Boolean Logic in Galois Fields

Galois Fields are finite fields that can be extended to any size $GF(n)$ but we use only the one-bit fields, i.e. $GF(2)$. Table 3.3 defines the multiplication and modulo 2 addition operators for inputs $\{a, b\}$ in $GF(2)$ and their respective equivalence to logic AND and logic XOR. The standard input values $\{1, 0\}$ correspond to the logic meanings $\{\text{True}, \text{False}\}$.

Table 3.3: Karnaugh Maps of Intrinsic Logic Operators in GF(2)

Multiply	b = 0	b = 1
a = 0	0	0
a = 1	0	1

$a * b \rightarrow a \text{ AND } b$

Add	b = 0	b = 1
a = 0	0	1
a = 1	1	0

$a + b \rightarrow a \text{ XOR } b$

Table 3.4 illustrates that GF(2) is Boolean complete since it intrinsically contains the logic AND operator and the inversion operator via logic XOR. Therefore any Boolean logic function can be written as a linear equation in GF(2) and simultaneous equations can be expressed as matrices.

Table 3.4: GF(2) is Boolean Complete

Boolean Logic Notation for {a, b}		
Operator	Symbol	using GF(2)
Identity a	a	$a * 1 = a + 0$
NOT a	$\neg a$	$a + 1$
a AND b	$a \wedge b$	$a * b = a b$
a XOR b	$a \otimes b$	$a + b$
a OR b	$a \vee b$	$a + b + a b$

This derivation is straightforward and can be expanded to any number of input states, but the two highlighted points warrant further comment. First, the NOT operator is implemented with XOR by adding a constant value of “1.” Second, the size of the OR expressions are longer in this algebra, since GF(2) intrinsically provides only logic XOR and logic AND. Subsection 3.5.2 demonstrates that logic OR gates for n inputs require $2^n - 1$ terms in GF(2).

3.5.2 XOR Dominated Logic

Logic AND operations can be expressed compactly using GF(2)'s intrinsic multiply operator. Conversely, the logic OR operator is not intrinsic and must rely on XOR, which is the mutual exclusion operator. Since XOR logic is unintuitive, some useful design insight will be presented. Traditional logic designers use Karnaugh maps to help define logic equations by circling groups of quadrants and writing down the *linearly* independent expression using logic inclusive OR, sometimes writing + to signify the logic OR (symbol \vee). This approach also works for Karnaugh maps using GF(2) *only if the coverage terms are non-overlapping or odd*. The Karnaugh maps and equations in Figure 3.1 depict this idea for logic OR.

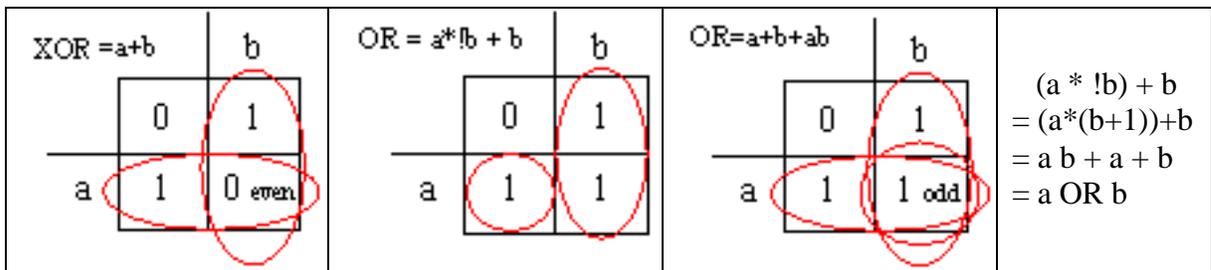


Figure 3.1: Karnaugh Maps of Logic OR using XOR logic

The linear independence difference between inclusive and exclusive OR is simply that XOR is odd parity: where each square in an XOR oriented Karnaugh map covered by an *odd number of terms* produces a value of 1. By definition, disjoint sets produce an odd number of covered squares for XOR logic and an even number of covered squares always produces a value of 0. Because of the symmetry of XOR, expressed as $(1 + \text{vector})$, the final number of terms in sum-of-products form grows large. A similar result will occur later in geometric algebra.

Table 3.5: Logic Inclusive OR for {2, 3, 4} input states in GF(2)

Input logic equation	Sum of all combinations in GF(2)	Number of terms
a OR b	$a + b + a b$	$2 + 1 = 3 = 2^2 - 1$
a OR b OR c	$a + b + c + a b + a c + b c + a b c$	$3 + 3 + 1 = 7$
a OR b OR c OR d	$a + b + c + d +$ $+ a b + a c + a d + b c + b d + c d$ $+ a b c + a b d + a c d + b c d$ $+ a b c d$	$4 + 6 + 4 + 1 = 15$

Applying this same approach for n inputs shows that inclusive OR in GF(2) requires every product combination of n inputs, or $2^n - 1$ terms (which is an odd number). Table 3.5 illustrates the logic OR expressions for $n \in \{2, 3, 4\}$ input states.

This result is significant for two reasons. First, since GF(2) is logic AND predominant, logic OR expressions require an exponential number of terms due to the influence of XOR. Second and more importantly, each product term used to define an operator in a GF(2) sum is *linearly independent* of all other terms. This directly implies that, given a matrix for orthogonal basis states $\{a, b\}$, then the basis set must be inflated (adding columns/rows) for each product term $\{a b\}$ in order to express Boolean logic operators in the linear matrix form. The next subsection demonstrates this result using the Fredkin and Toffoli gates.

3.5.3 Toffoli and Fredkin Gates in GF(2)

The universal reversible Fredkin gate has three inputs $[c b a]$ and three outputs $[C B A]$. The three inputs form a basis vector set using the values $\{1, 0\}$. The Fredkin linear operator is a function F dependent on control “c” that satisfies equation $[C B A] = [c b a] * F(c)$.

The summary of this result is shown in Table 3.6. See Appendix A for the full derivation of this GF(2) result.

Table 3.6: Fredkin Gate as Matrix Operator in GF(2)

$F_{c=1} = \text{Identity}$	$F_{c=0} = \text{Swap}$	$F(c) = F_0 + F_1$	Fredkin Matrix $[C B A] =$
$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & c & !c \\ 0 & !c & c \end{bmatrix}$	$[c \ b \ a \ \mathbf{bc} \ \mathbf{ac}]^* \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & \mathbf{1} & \mathbf{1} \\ 0 & \mathbf{1} & \mathbf{1} \end{bmatrix}$
$A = a, B = b, C = c$	$A = b, B = a, C = c$	$A = \mathbf{b + bc + ac}, B = \mathbf{a + bc + ac}, C = c$	

As expected, the Fredkin gate can be expressed as simultaneous equations in universal GF(2) where $A = b + \mathbf{bc} + \mathbf{ac}$, $B = a + \mathbf{bc} + \mathbf{ac}$, and C always equals c. The unexpected result is the conditional matrix representation of $F(c)$ can only be described in a larger linear vector space, inflated by the product of the control signal with each data line (see the bold text in table). Similarly, as Table 3.7 illustrates, the Toffoli gate with controls c and b should also be expressible as a matrix in GF(2) of the form $[C \ B \ A] = [c \ b \ a]^* T(c, b)$. However the vector space must again be inflated to express this operator.

Table 3.7: Toffoli Gate as Matrix Operator in GF(2)

c b a	C B A	Observable?	Toffoli Matrix $[C B A] =$
0 0 0	0 0 0	same	$[c \ b \ a \ \mathbf{bc}]^* \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & \mathbf{1} \end{bmatrix}$
0 0 1	0 1 0	same	
0 1 0	0 0 1	same	
0 1 1	0 1 1	same	
1 0 0	1 0 0	same	
1 0 1	1 0 1	same	
$\mathbf{1 \ 1 \ 0}$	$\mathbf{1 \ 1 \ 1}$	visible	$A = \mathbf{a + bc}, B = b, C = c$
$\mathbf{1 \ 1 \ 1}$	$\mathbf{1 \ 1 \ 0}$	visible	

3.5.4 Boolean Logic is formally non-linear

Based on universality and reversibility requirements in the context of finite linear systems, it is clear that conditional logic operators such as $F(c)$ and $T(c, b)$ can not be expressed solely in the linear space defined only by the input state space of $\{a, b, c\}$. The size of the linear space must be inflated by all the conditional terms, expressed as products, required to represent the logic operators in a linear algebra such as $GF(2)$. Once expressed in this expanded linear matrix format, the entire system defined by both the state and the operators is subject to full matrix techniques. Full reversibility is possible if all input and output states are maintained (i.e. square matrices) and no information is gratuitously erased.

The independent choice of including the product term $(a \cdot b)$ inside some function $f(a, b)$ is solely dependent on its desired behavior (compare the behaviors of $a + b$ versus $a + b + a \cdot b$) and independent of the values of either a or b . Therefore, a closed linear space which is to be capable of representing both the functional operators and their state must be inflated by all the products used by the functional behaviors. This is very important since the *states and operators* of many algebras are the same elements (cf. they are group elements), and the ability to distinguish between them is based on their use.

The size of the inflated linear space depends on both the number of input vectors and the complexity of the operations performed. The reason logic OR creates the maximum space inflation is that its output is reached from almost any combination of input states, which represents a many-to-one mapping. The reversible linear space must therefore be big enough to hold and reverse the state changes made by operators that perform logic OR.

In conclusion, any reversible linear system is Boolean complete when it reaches a certain minimum size. To represent both the states and the operators of arbitrary Boolean functions, the size of the closed linear system must be further expanded. The overall size of the closed reversible linear system is dependent on the amount of state plus the overall complexity of the operations performed, but is bounded by 2^n elements.

CHAPTER 4

GEOMETRIC ALGEBRA FOUNDATIONS

4.1 Matrix versus Algebraic Notation

This chapter summarizes the principles of geometric algebra (GA) required for this dissertation. A significant body of GA development has occurred, notably at Arizona State University and the University of Cambridge. GA unifies much of the existing mathematics in the specialized areas of vector algebra, quaternions, spinor algebra, matrix, and tensor algebra into a topologically based framework pioneered by William Clifford (1845-1879), who united the inner product with Grassmann's outer product. David Hestenes of ASU resurrected GA into the modern era.

Most of modern physics is based on vector and matrix algebra, plus the proliferation of other novel algebraic systems listed above that were created as they were needed. These specialized systems are naturally included as part of the topological approach to geometric algebra, including such broad application areas as mechanics, quantum mechanics, and general relativity. GA accomplishes this by using a high-dimensional algebraic notation that relies on neither complex numbers nor matrix notation.

As will be shown, GA's use of high-dimensional algebraic notation is particularly useful for efforts in quantum computing, because computer engineers and scientists have different skills

from physicists and mathematicians, thus making cross-disciplinary efforts in quantum computing difficult. Relying on real-valued, mixed-rank, algebraic notation, geometric algebra removes this difficulty and serves as a unifying mathematical language. As a result, GA makes quantum computing more understandable and palatable to engineers, and therefore it is more accessible to a broader group of researchers and developers.

4.2 Co-Occurrence and Co-Exclusion

Before introducing GA principles, it is important to introduce Manthey's [23] framework for the *meaning* of addition and multiplication operators. My research relies heavily on Manthey's interpretation and so it is introduced early in the discussion. I was originally attracted to his framework because of the allure of a mathematical formalism based on only addition and multiplication that bootstraps respectively spatial and temporal concepts for logic. I found much more by adopting his approach and interpretation, which works naturally with the highly symmetric geometric algebra. Manthey's conceptual primitives are Co-Occurrence and Co-Exclusion.

An important premise of Manthey's work is that a simple state and its additive inverse *exclude* each other (written as $\mathbf{a} \leftrightarrow \bar{\mathbf{a}}$), which defines the *mutually* exclusive states of a classical bit. Simple states are binary, where $+\mathbf{a} = \mathbf{a} = \text{"ON"}$ and $(-\mathbf{a}) = \bar{\mathbf{a}} = \text{"NOT ON"}$ and denoted by simple vectors \mathbf{a} , \mathbf{b} , \mathbf{c} , etc. that are herein generally considered to be orthonormal. These two states are classical and their exclusion is expressed as $\mathbf{a} + \bar{\mathbf{a}} = 0$. *Exclusion* is the constraint that two given states of a vector CANNOT logically or meaningfully said to occur at the same instant in time.

Co-occurrence is defined as two or more states “occurring” simultaneously and is expressed as the *sum* of these states. For states **a**, **b**, **c**, etc. (which will later be taken to be vector elements of GA), if state **a** and state **b** can occur at *exactly* the same instant in time (assuming Einstein locality), the relationship is denoted as $\pm\mathbf{a} \pm \mathbf{b}$ or $\pm\mathbf{b} \pm \mathbf{a}$, in that commutative properties of addition reflects the lack of temporal ordering characteristic of simultaneity. Since subtraction is *not* commutative, always interpret $-\mathbf{a} - \mathbf{b}$ as $\bar{\mathbf{a}} + \bar{\mathbf{b}}$. The interpretation of simultaneous states as *co-occurring* is used heavily in this research to give consistent and insightful meaning when combining multiple states using addition.

Co-exclusion requires two co-occurrences (wherein each component of the constituent co-occurrences can change independently) that exclude each other. Initially the state of the co-occurrence is say $(\mathbf{a} + \mathbf{b})$ and later the state is observed as $(\bar{\mathbf{a}} + \bar{\mathbf{b}})$. Since this later state is the complement of the earlier one, this transition implies that some operator interacted (using multiplication) with the system. The existence of such an operator implicitly invokes a hierarchical abstraction: *two* 1-bit-of-state processes (**a** and **b**) have been treated (by this operator) as *one* process with *two* bits of state. This operator can be determined, and in fact is expressed in geometric algebra as the product $(\mathbf{a} \mathbf{b})$. For states **a**, **b**, **c**, etc the co-exclusion can be denoted in either of the following two ways:

$$(\mathbf{b} + \mathbf{c}) \leftrightarrow (\bar{\mathbf{b}} + \bar{\mathbf{c}}) \quad \text{or} \quad (\mathbf{b} + \mathbf{c}) | (\bar{\mathbf{b}} + \bar{\mathbf{c}}) \quad (4.1)$$

Since co-exclusion infers an operator for change and implies temporal sequence, it is considered to be the temporal primitive. Conversely, co-occurrence is labeled the spatial primitive since it is static. Also, according to Einstein and Feynman, if anything is conserved it must be conserved locally, so token synchronization/conservation can only occur locally.

Mutual exclusion (or mutex), which controls execution or access order for shared computational resources, is historically the motivation for synchronization tokens. So Manthey's synchronization-based concepts of co-occurrence, exclusion and co-exclusion form the foundation for the computational interpretation of GA used in this dissertation. No other temporal or causal ideas are defined or implied for GA addition and multiplication. See Figure 4.1 for a graphical summary of these relationships.

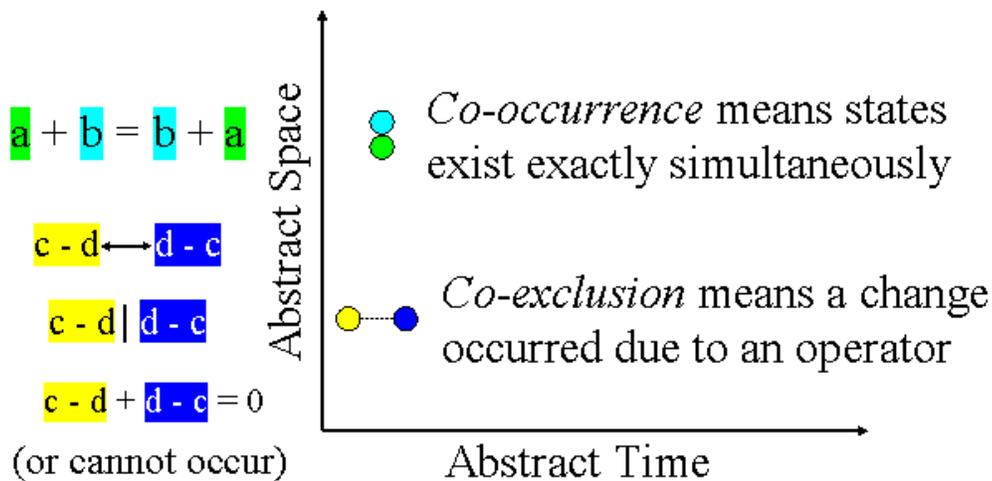


Figure 4.1: Co-occurrence and Co-exclusion Concepts

According to Manthey, a Turing machine cannot express the notion of true simultaneity since no co-occurrence primitive exists for it. His coin demonstration shown below illustrates this point about true simultaneity by relying on the premise that the coins are *formally* identical tokens, which are indistinguishable in every respect (cf. synchronization tokens). You could also use electrons or photons as tokens if you prefer.

The Coin Demonstration [23]:

Act I: *A man stands in front of you with both hands behind his back. He shows you one hand containing a coin and then returns the hand and coin behind his back. After a brief pause, he again shows you the same hand with what appears to be an identical coin. He again hides it and asks, “How many coins do I have?”*

The best answer at this point is “*at least one,*” which represents one bit of information with two possible states, state1 = “one coin” or state2 = “more than one coin.”

Act II: *The man now extends the same hand and it contains two identical coins.*

We now know that there are two coins, that is, *we have received one bit of information, in that the ambiguity is resolved.* We have now arrived at the final step in this demonstration.

Act III: *The man now asks, “Where did that bit of information come from??”*

This bit originates in the *simultaneous presence of the two coins.* Thus true concurrency *cannot* be simulated with a sequential Turing machine no matter how fast the individual tokens are sequentially presented. Likewise, even though modern general-purpose computers are Turing equivalent, the missing atomic “test and set” locking primitive must usually be added to the architecture in order to support the mutual exclusion synchronization tokens needed by the operating systems. Manthey argues that computing only with tokens is equivalent to quantum mechanics and he pioneered the use of discrete GA applied to quantum mechanics used in this dissertation. Addition is used to express the simultaneity of co-occurrence while multiplication is the mechanism for state change operators. Co-exclusion implies an operator must interact with the system state to allow a new state to emerge.

Distinct labels for “states” and “operators” become blurred when working with the mathematical elements defined by GA because both uses are merely conventions applied to the same expressions in the algebra. Also, these states are NOT assumed to be classical. Multiplication just means expressions in the algebra (states and operators) interact in a self-consistent manner without any notion of classical time or causality. According to Manthey, a simple state \mathbf{s} and its inverse [= not \mathbf{s}] cannot logically occur at the same instant in time. This is true for both individual simple states and complex sums of states. This postulate is expressed in equation (4.2) as a co-occurrence expression that sums to zero, and thus reserves the special meaning of “cannot occur” for 0.

$$\mathbf{a} + \bar{\mathbf{a}} = \mathbf{a} + (-\mathbf{a}) = 0 \quad \text{or} \quad (\mathbf{b} + \bar{\mathbf{c}}) + (\bar{\mathbf{b}} + \mathbf{c}) = 0 \quad (4.2)$$

This interpretation means that the traditional mathematical technique of equating an expression to zero and solving for the roots has a different meaning because the solutions found *cannot occur* and actually represent the *non-solutions*. This idea will acquire more meaning later when we look at destructive interference for sets of quantum states.

4.3 Geometric Algebra Principles

Geometric algebra is considered by many to be a universal, topologically-based, mathematical language for mathematics, physics, and engineering, [21] because GA’s real-valued geometric product (consisting of the sum of inner and outer products) replaces complex numbers. The original approach behind GA was developed by William Clifford in 1878 and has these benefits:

- 1) scales to arbitrary number of dimensions (i.e. scalars, vectors, bivectors, trivectors, etc)
- 2) unit imaginary is replaced by topologically derived pseudoscalar or n -vector

- 3) geometric product is invertible, which is the multiplicative inverse denoted as $1/A=A^{-1}$
- 4) encompasses anticommutative multiplication, quaternions, Pauli and Dirac spins, etc
- 5) generalized rotors work for both quantum mechanics and dilation in relativity.

The remainder of this section includes the basic definitions of geometric algebra, organized by the grade (or arity) of the elements. These elements form a finite algebra that is closed under addition, subtraction, multiplication and inversion, but not division (so is *not* a division algebra). Other books [16] carefully derive the properties of geometric algebra, so this section introduces only the concepts and rules used elsewhere in this dissertation.

4.3.1 Scalars and Vectors

Scalars and vectors have the familiar definitions from general mathematics. Scalars are just a real valued magnitude without any orientation or direction. A vector adds the notion of orientation where the sign is significant. The notation convention used here is to denote scalars (only real values $-1, 0,$ and $+1$) in normal font Greek characters $\{\mathbf{a}, \mathbf{b}, \mathbf{l}, \mathbf{m}, \dots\}$, bold vector names usually start in lower case alphabetic characters $\{\mathbf{a}, \mathbf{a1}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \dots\}$ and bivectors in bold uppercase alphabetic characters $\{\mathbf{A}, \mathbf{B}\}$. Normal associative, distributive, and vector addition rules apply.

The addition of vectors ($\mathbf{a} + \mathbf{b}$) forms a new vector by placing the tail of \mathbf{b} to the head of \mathbf{a} .

Any set of n linearly independent vectors forms a *basis* set for a space of dimension n .

Multiplying a scalar times a vector represents dilation of the vector length and also allows direction inversion. Scalars are defined as grade-0 (denoted as $\langle A \rangle_0$) and vectors as grade-1 (denoted as $\langle A \rangle_1$). Scalars and vectors (and higher-grade terms) can be added together to

form a mixed grade *multivector* ($A = \langle A \rangle_0 + \langle A \rangle_1 + \dots + \langle A \rangle_n$), which is conceptually similar to representing complex numbers as the sum of the real and imaginary parts.

4.3.2 Bivectors and Anticommutative Multiplication

Geometric algebra is novel because the outer product (\wedge) is formed by multiplying two vectors together to create a grade-2 *bivector*. A bivector represents the parallelogram defined by two vectors with its orientation defined by the right hand rule. This also applies for oriented volumes (*trivector*) and higher dimensional n-volumes (n-vectors). The diagram in Figure 4.2 portrays the topological basis for the anticommutative property of $\mathbf{a} \wedge \mathbf{b} = -\mathbf{b} \wedge \mathbf{a}$.

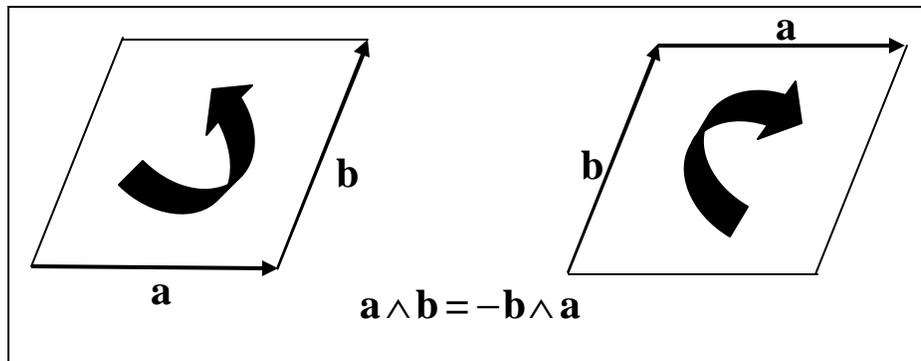


Figure 4.2: Bivector Defines an Oriented Area (via right hand rule)

Clifford developed the mechanism of maintaining orientation in geometric algebra by discovering that a geometric product is the sum of the inner and outer products. The geometric product can be generalized to any number of dimensions because the inner product is a grade reducing operation and the outer product is a grade increasing operation. The following equations define the geometric product for unit length vectors $\{\mathbf{a}, \mathbf{b}\}$ with the angle between them \mathbf{q} :

$$\mathbf{a} \mathbf{b} = -\mathbf{b} \mathbf{a} = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \wedge \mathbf{b} \quad \text{geometric product is inner (dot) plus outer (wedge) products} \quad (4.3)$$

$$\mathbf{a} \cdot \mathbf{b} = \cos q \quad \text{inner product scalar is maximal when unit vectors } \mathbf{a} \text{ and } \mathbf{b} \text{ are collinear} \quad (4.4)$$

$$\mathbf{a} \wedge \mathbf{b} = i \sin q \quad \text{outer product bivector is max when } \mathbf{a} \text{ and } \mathbf{b} \text{ are perpendicular; } i = \sqrt{-1} \quad (4.5)$$

Likewise, once the geometric product is defined, the inner and outer products can be rewritten as the symmetric and anti-symmetric sum (or difference) as in eqns (4.6) and (4.7).

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a} = \frac{1}{2}(\mathbf{a} \mathbf{b} + \mathbf{b} \mathbf{a}) \quad \text{symmetric sum forms the scalar inner product} \quad (4.6)$$

$$\mathbf{a} \wedge \mathbf{b} = -\mathbf{b} \wedge \mathbf{a} = \frac{1}{2}(\mathbf{a} \mathbf{b} - \mathbf{b} \mathbf{a}) \quad \text{anti-symmetric difference forms the bivector outer product} \quad (4.7)$$

Precedence rules exist for inner, outer, and geometric products when parentheses are omitted.

Inner and outer products should always be performed before an adjacent geometric product.

Likewise, outer products have precedence over inner products.

Most uses of geometric (or Clifford) algebra \mathbf{G}_n use its vector properties for expressing relationships within a predefined $n \in \{2, 3, \text{ or } 4\}$ dimensional space of arbitrary extent, where the basis vectors formally defining the space are important but not the primary concern. The present research focuses primarily on the definition and formal properties of discrete high-dimensional spaces generated by *orthonormal* (i.e. unit length and mutually perpendicular) vectors. For the basis vectors, *the inner product is always zero*. Under these

conditions, the geometric and outer products are identical ($\mathbf{a} \mathbf{b} = \mathbf{a} \wedge \mathbf{b}$) so the wedge character is sometimes dropped, when no confusion is possible.

The following properties¹ exist for a plane defined by orthonormal basis vectors $\text{span}\{\mathbf{a}, \mathbf{b}\}$.

$$\mathbf{a}^2 = \mathbf{b}^2 = 1 \text{ (since self collinear) and remember } \mathbf{a} \cdot \mathbf{b} = 0 \quad (4.8)$$

Due to the geometric product, this basis set can also generate the bivector $\mathbf{a} \wedge \mathbf{b} = \mathbf{a} \mathbf{b}$, whose *orientation* is *orthogonal* to both vectors \mathbf{a} and \mathbf{b} (and equivalent to cross product in 3D).

Including the scalars $\{0, \pm 1\}$, this forms the geometric algebra \mathbf{G}_2 which is closed over addition and multiplication and forms a four-dimensional finite algebra. \mathbf{G}_2 defines the graded elements $\{\langle A \rangle_0, \langle A \rangle_1, \langle A \rangle_2\}$: the scalars $\langle A \rangle_0 = \{0, \pm 1\}$, two vectors $\langle A \rangle_1 = \{\mathbf{a}, \mathbf{b}\}$ and one bivector $\langle A \rangle_2 = \{\mathbf{a} \mathbf{b}\}$. The two vectors form the odd grade set $\langle A \rangle_+ = \langle A \rangle_1$, whereas the even grade includes the scalar and bivector elements $\langle A \rangle_- = \langle A \rangle_0 + \langle A \rangle_2$, resulting in the set of all possible multivectors of the form $A = \langle A \rangle_+ + \langle A \rangle_- = \langle A \rangle_0 + \langle A \rangle_1 + \langle A \rangle_2$.

The highest-grade element in this group is called the pseudoscalar (denoted as \mathbf{I}) and for some grades its square is equal to -1 (see eqn 4.9). This pseudoscalar has geometric roots, is defined for arbitrary grade \mathbf{G}_n and for \mathbf{G}_{2-3} has the properties of the unit imaginary ($i^2 = -1$).

$$\mathbf{I}^2 = (\mathbf{a} \mathbf{b})^2 = \mathbf{a} \mathbf{b} \mathbf{a} \mathbf{b} = -\mathbf{a} \mathbf{a} \mathbf{b} \mathbf{b} = -(\mathbf{a})^2 (\mathbf{b})^2 = -1 \quad (4.9)$$

¹ Actually in GA $\mathbf{a}^2 = \pm 1$ but the signature for this dissertation is $\text{Sig} = (n,0)$

The order of multiplication is significant because of GA's anticommutative properties. Multiplying a vector times a bivector can be performed from the left or right sides. Left multiplication rotates a vector 90° clockwise in the bivector plane and right multiplication (used in this paper) rotates a vector 90° counterclockwise. Due to this property, any bivector is called a spinor. Here are two right multiplication examples.

$$\mathbf{a} I = \mathbf{a} (\mathbf{a} \mathbf{b}) = (\mathbf{a} \mathbf{a}) \mathbf{b} = \mathbf{b} \quad \text{similarly} \quad \mathbf{b} I = \mathbf{b} (\mathbf{a} \mathbf{b}) = -\mathbf{a} \mathbf{b} \mathbf{b} = -\mathbf{a} (\mathbf{b} \mathbf{b}) = -\mathbf{a} \quad (4.10)$$

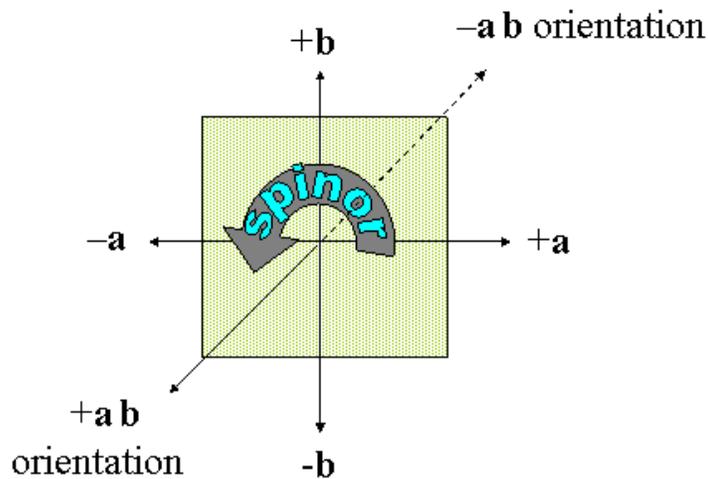


Figure 4.3 Spinor Plane

4.3.3 N-vectors and Mixed Rank Expressions

The definitions in section 4.3.2 can be expanded for larger basis sets. For the orthonormal basis set containing n basis vectors $\{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \dots\}$, the same methodology applies. The n orthonormal vectors generate the geometric algebra of \mathbf{G}_n where the elements in the algebra are the scalars $\langle A \rangle_0$, the vectors $\langle A \rangle_1$, all unique vector product pairs of bivectors $\langle A \rangle_2$, all unique vector product trios of trivectors $\langle A \rangle_3$, etc. until the unique n -vector pseudoscalar $\langle A \rangle_n$ is reached. For any \mathbf{G}_n the pseudoscalar I is always of grade n . The sum of any

combination of the n-vectors $A = \langle A \rangle_0 + \langle A \rangle_1 + \dots + \langle A \rangle_n$ is a valid member of \mathbf{G}_n and is identified as a multivector.

The geometric product of a vector with a higher grade n-vector works for \mathbf{G}_n , as illustrated using a vector \mathbf{a} , a bivector \mathbf{B} and arbitrary grade n-vectors \mathbf{A}_j and \mathbf{B}_k , but a general product of $\mathbf{A}_j \mathbf{B}_k$ does not generally equal $\mathbf{A}_j \cdot \mathbf{B}_k + \mathbf{A}_j \wedge \mathbf{B}_k$ unless one of the factors is a vector.

Table 4.1 : Product Summary

$\mathbf{a} \mathbf{B} = \mathbf{a} \cdot \mathbf{B} + \mathbf{a} \wedge \mathbf{B}$	$\mathbf{a} \mathbf{B}_k = \mathbf{a} \cdot \mathbf{B}_k + \mathbf{a} \wedge \mathbf{B}_k$	$\mathbf{A}_j \mathbf{B}_k \neq \mathbf{A}_j \cdot \mathbf{B}_k + \mathbf{A}_j \wedge \mathbf{B}_k$
$\mathbf{a} \cdot \mathbf{B}$ is a 1-vector	$\mathbf{a} \cdot \mathbf{B}_k = \langle \mathbf{a} \mathbf{B}_k \rangle_{ k-1 }$	$\mathbf{A}_j \cdot \mathbf{B}_k = \langle \mathbf{A}_j \mathbf{B}_k \rangle_{ j-k }$
$\mathbf{a} \wedge \mathbf{B}$ is a 3-vector	$\mathbf{a} \wedge \mathbf{B}_k = \langle \mathbf{a} \mathbf{B}_k \rangle_{ k+1 }$	$\mathbf{A}_j \wedge \mathbf{B}_k = \langle \mathbf{A}_j \mathbf{B}_k \rangle_{ j+k }$

In general $\mathbf{A}_j \mathbf{B}_k = \langle \mathbf{A}_j \mathbf{B}_k \rangle_{j+k} + \langle \mathbf{A}_j \mathbf{B}_k \rangle_{j+k-2} + \dots + \langle \mathbf{A}_j \mathbf{B}_k \rangle_{|j-k|}$, where the inner and outer product is reserved for smallest and largest grade elements. For arbitrary bivectors \mathbf{A} and \mathbf{B} ,

the expression $\mathbf{A} \wedge \mathbf{B} = \frac{1}{2}(\mathbf{A} \mathbf{B} - \mathbf{B} \mathbf{A})$ is the commutator because it vanishes if \mathbf{A} and \mathbf{B}

commute.

For n-vector $\mathbf{A} = \mathbf{a}_0 \mathbf{a}_1 \dots \mathbf{a}_n$, the reverse $\tilde{\mathbf{A}}$ is the vectors listed in reverse order $\mathbf{a}_n \dots \mathbf{a}_1 \mathbf{a}_0$ and this is the same as Hermitian adjoint $\tilde{\mathbf{A}} = \mathbf{A}^\dagger$. Reversion flips the sign of bivectors and trivectors. Reversion does not affect scalars, vectors, or 4-vectors, which means they are self-adjoint or $\mathbf{A} = \tilde{\mathbf{A}} = \mathbf{A}^\dagger$. In general, the reverse of each grade is independent: $\tilde{\mathbf{A}} = \sum_i \langle \mathbf{A} \rangle_i^\dagger$.

4.3.4 Number of Elements in a Geometric Algebra

The total number of unique graded elements in any \mathbf{G}_n is $N = 2^n$. The number of elements of each grade can be quickly found by using Pascal's triangle shown in Figure 4.4 because each row n is a binomial expansion that sums to N . Each entry is the sum of the two numbers above it $C(n, m) = C(n-1, m-1) + C(n-1, m)$ in the triangle, and also represents the number of unique ways for n tokens taken m at a time $\binom{n}{m}$. Since the sum is a power of 2 (and contains *only* factors of 2 then N has *only even* factors).

<i>Row = n</i>	<i>Col = m</i>	$1 + \sum_{m=1}^n \binom{n}{m} = N = 2^n$
0	1	= 1
1	1 1	= 2
2	1 2 1	= 4
3	1 3 3 1	= 8
4	1 4 6 4 1	= 16
5	1 5 10 10 5 1	= 32
6	1 6 15 20 15 6 1	= 64

Figure 4.4: First seven rows of Pascal's Triangle

For a vector space of size n the first slanted column (left side of triangle) represents the scalars $\langle A \rangle_0$. The second column is the number of vectors in set $\langle A \rangle_1$ (in bold and $=n$), followed by the number of bivectors in set $\langle A \rangle_2$, etc for each $\langle A \rangle_m$ until the pseudoscalar $\langle A \rangle_n$ is reached. The total number of elements in an algebra \mathbf{G}_n defined by n vectors grows exponentially. This very large state space will be shown to be identical to the linear state

space requirements for quantum mechanics and matches the Boolean decode developed in the Section 4.4.

4.3.5 Quaternions in Geometric Algebra

Quaternions were invented by William Hamilton to solve the problem of arbitrary rotations in 3D Euclidean space \mathbf{E}_3 defined by orthonormal vectors $\{\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3\}$ (using the GA vector convention of bold font to distinguish from the Pauli matrices introduced later).

The set of n-vectors generated from the basis of \mathbf{E}_3 forms the geometric algebra \mathbf{G}_3 . The odd-grade elements $\mathbf{G}_3^- = \{\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3, \mathbf{s}_1 \mathbf{s}_2 \mathbf{s}_3\}$ consist of the original three 1-vectors and the trivector (or pseudoscalar \mathbf{I}) while the scalar and three bivectors (or spinors) constitute the even-grade elements $\mathbf{G}_3^+ = \{\pm 1, \mathbf{s}_1 \mathbf{s}_2, \mathbf{s}_1 \mathbf{s}_3, \mathbf{s}_2 \mathbf{s}_3\}$. Thus \mathbf{G}_3 can be defined as $\mathbf{G}_3^- + \mathbf{G}_3^+$.

The elements of \mathbf{G}_3^+ define a linear space of four dimensions that is a finite algebra closed under multiplication and addition. This subset is called the *even subalgebra* of \mathbf{G}_3 . In \mathbf{G}_3 and \mathbf{G}_3^+ a bivector times a bivector always returns either another bivector or a value of -1 .

The even spinor subalgebra \mathbf{G}_3^+ is isomorphic to Hamilton's quaternion algebra $\{i, j, k\}$, where $i^2 = j^2 = k^2 = ijk = -1$, $ij = k$, $jk = i$ and $ki = j$. In reality $\{i, j, k\}$ are **bivectors** (where $i = \mathbf{s}_1 \mathbf{s}_2$, $j = \mathbf{s}_3 \mathbf{s}_1$, and $k = \mathbf{s}_2 \mathbf{s}_3$) with their usual anticommutative properties.

Alternatively, the bivectors can be written as a product of the pseudoscalar and a vector where $i = \mathbf{I} \mathbf{s}_3$, $j = \mathbf{I} \mathbf{s}_2$, and $k = \mathbf{I} \mathbf{s}_1$. The only difference between the quaternions and the GA work presented here, is that by convention, every vector and expression in GA would be placed in a standard sort order by applying the right hand rule uniformly, resulting in the

opposite sign for term $j' = -j = s_1 s_3$ and consequently $ij' = -k$, $j'k = -i$ and $ik = j$. That quaternions are equivalent to the even subalgebra of \mathbf{G}_3 is why four dimensions are required when dealing with three spatial dimensions.

The last topic regarding the even subalgebra is the choice of bivectors as the primary basis for \mathbf{E}_3 . In a three-dimensional space \mathbf{E}_3 , the three basis vectors $\{\mathbf{x}, \mathbf{y}, \mathbf{z}\}$ also define three orthogonal planes. Each one of the bivectors defining these planes has an orientation that points *into* the unit box and is the mathematical basis of Gibbs's vector cross product (only works in three dimensions and see [16] for details). These bivectors geometrically constitute a dual for each basis vector ($\mathbf{x} \sim \mathbf{y} \mathbf{z}$, $\mathbf{y} \sim -\mathbf{x} \mathbf{z}$, and $\mathbf{z} \sim \mathbf{x} \mathbf{y}$) as shown in Figure 4.5. This is topologically why quaternions can represent \mathbf{E}_3 using only bivectors.

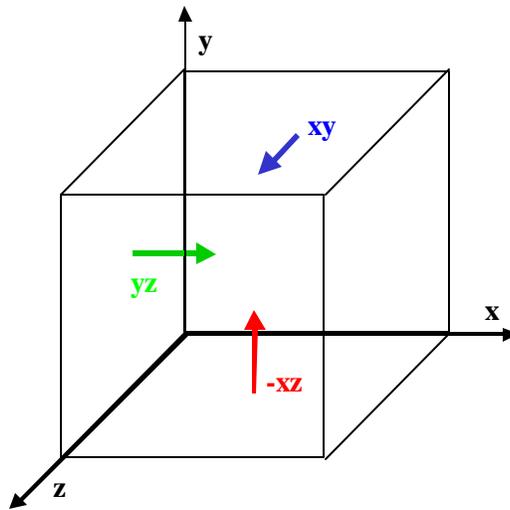


Figure 4.5: Quaternions as Bivectors in three dimensions

Another more pragmatic reason for choosing bivectors over vectors deals with the anticommutative properties of vectors versus bivectors. For orthonormal vectors \mathbf{a} , \mathbf{b} , and \mathbf{c} with the bivector $\mathbf{B} = \mathbf{b} \mathbf{c}$, the following facts are true.

$$\mathbf{a} \mathbf{b} = - \mathbf{b} \mathbf{a}, \text{ vector geometric product is anticommutative but} \quad (4.11)$$

$$\mathbf{a} \mathbf{B} = \mathbf{a} (\mathbf{b} \mathbf{c}) = - \mathbf{b} \mathbf{a} \mathbf{c} = + \mathbf{b} \mathbf{c} \mathbf{a} = \mathbf{B} \mathbf{a} \quad (\text{bivectors commute}) \quad (4.12)$$

The choice of a bivector basis allows the writing of terms in geometric product expressions in any order without needing to keep track of the inversions, which is very useful during blind algebraic substitutions. Substitutions must be used *exceedingly* cautiously because, for example, equation 4.12 only works if the dimensions do not intersect or $\mathbf{a} \cdot \mathbf{B} = 0$.

4.3.6 Inner Product Definition

Many of the examples illustrated so far describe the algebraic details of computing the geometric product. The algebraic rules for computing the inner product are not much more difficult, but the literature descriptions seem complex and confusing. The only cases introduced so far is the inner product of two orthonormal vectors $\mathbf{G}_2 = \text{span}\{\mathbf{a}, \mathbf{b}\}$, such that $\mathbf{a} \cdot \mathbf{b} = 0$ means *orthogonal* and $\mathbf{a} \cdot \mathbf{a} = \mathbf{b} \cdot \mathbf{b} = 1$ means self collinear and of unit length. The next paragraphs describe the other cases of inner products between n-vectors, assuming the restriction that *the vectors $\{\mathbf{a}, \mathbf{b}\}$ in \mathbf{G}_2 are orthonormal.*

The definition of the inner product $\mathbf{x} \cdot \mathbf{Y}$ for vector \mathbf{x} and bivector $\mathbf{Y} = (\mathbf{y} \wedge \mathbf{z})$ is:

$$\mathbf{x} \cdot \mathbf{Y} = \mathbf{Y} \cdot \mathbf{x} = \mathbf{x} \cdot (\mathbf{y} \wedge \mathbf{z}) = (\mathbf{x} \cdot \mathbf{y}) \wedge \mathbf{z} - (\mathbf{x} \cdot \mathbf{z}) \wedge \mathbf{y} \quad (4.13)$$

where \mathbf{x} , \mathbf{y} , and \mathbf{z} are *place holder variables* for actual vector names. Notice that for \mathbf{G}_2 only two actual vectors exist, so if $\mathbf{Y} = (\mathbf{a} \wedge \mathbf{b})$ then the result is $\mathbf{x} \cdot (\mathbf{a} \wedge \mathbf{b}) = (\mathbf{x} \cdot \mathbf{a}) \wedge \mathbf{b} - (\mathbf{x} \cdot \mathbf{b}) \wedge \mathbf{a}$ with either $\mathbf{x} = \mathbf{a}$ or $\mathbf{x} = \mathbf{b}$. This means *only one* of the terms of the sum is *non-zero* since either $\mathbf{a} \cdot \mathbf{a} = 1$ and $\mathbf{a} \cdot \mathbf{b} = 0$ or $\mathbf{b} \cdot \mathbf{a} = 0$ and $\mathbf{b} \cdot \mathbf{b} = 1$. Knowing this important inner product simplification for orthonormal vectors is very useful as the grade of the n -vector increases.

The inner product $\mathbf{w} \cdot \mathbf{Z}$ of vector \mathbf{w} and n -vectors $\mathbf{Y} = (\mathbf{y} \wedge \mathbf{z})$ and $\mathbf{Z} = (\mathbf{x} \wedge \mathbf{Y})$ is defined as:

$$\mathbf{w} \cdot \mathbf{Z} = \mathbf{Z} \cdot \mathbf{w} = \mathbf{w} \cdot (\mathbf{x} \wedge \mathbf{Y}) = (\mathbf{w} \cdot \mathbf{x}) \wedge \mathbf{Y} - \mathbf{x} \wedge (\mathbf{w} \cdot \mathbf{Y}) \quad (4.14)$$

where \mathbf{w} , \mathbf{x} , \mathbf{y} , and \mathbf{z} are again place holder variables for actual vector names. This result depends recursively on the previous bivector inner product so the substitutions can continue.

$$\begin{aligned} \mathbf{w} \cdot \mathbf{Z} &= (\mathbf{w} \cdot \mathbf{x}) \wedge \mathbf{Y} - \mathbf{x} \wedge ((\mathbf{w} \cdot \mathbf{y}) \wedge \mathbf{z} - (\mathbf{w} \cdot \mathbf{z}) \wedge \mathbf{y}) \\ &= (\mathbf{w} \cdot \mathbf{x}) \wedge \mathbf{y} \wedge \mathbf{z} - (\mathbf{w} \cdot \mathbf{y}) \wedge \mathbf{x} \wedge \mathbf{z} + (\mathbf{w} \cdot \mathbf{z}) \wedge \mathbf{x} \wedge \mathbf{y} \end{aligned} \quad (4.15)$$

This result has a very specific pattern (minus sign is due to moving the chosen vector to the beginning of the n -vector) where the initial vector \mathbf{w} forms a dot product with each vector in the n -vector, but only one dot product will be non-zero. Again by using actual vectors for $\mathbf{G}_3 = \text{span}\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ only three vectors exist, so with $\mathbf{x} = \mathbf{a}$, $\mathbf{y} = \mathbf{b}$, and $\mathbf{z} = \mathbf{c}$ then the result is:

$$\mathbf{w} \cdot \mathbf{Z} = (\mathbf{w} \cdot \mathbf{a}) \wedge \mathbf{b} \wedge \mathbf{c} - (\mathbf{w} \cdot \mathbf{b}) \wedge \mathbf{a} \wedge \mathbf{c} + (\mathbf{w} \cdot \mathbf{c}) \wedge \mathbf{a} \wedge \mathbf{b} \quad (4.16)$$

with either $\mathbf{w} = \mathbf{a}$, $\mathbf{w} = \mathbf{b}$, or $\mathbf{w} = \mathbf{c}$. This again means only one term in the sum will be non-zero and the same procedure can be applied to an n -vector \mathbf{Z} of any grade. The literature always writes this expansion as a large sum, but for orthogonal vectors, only one term where $(\mathbf{w} \cdot \mathbf{x}) = 1$ will remain, which occurs *if and only if* $\mathbf{w} = \mathbf{x}$. If vector \mathbf{w} is not one of the dimensions contained in n -vector \mathbf{Z} then $\mathbf{w} \cdot \mathbf{Z} = 0$, which is also true if “ w ” is a scalar.

The next step shows how to compute the inner product of an m -vector, say $\mathbf{X} = (\mathbf{v} \wedge \mathbf{w})$, and an n -vector say $\mathbf{Y} = (\mathbf{x} \wedge \mathbf{y} \wedge \mathbf{z})$ where \mathbf{v} , \mathbf{w} , \mathbf{x} , \mathbf{y} , and \mathbf{z} are vector place holders and $m < n$.

$$\mathbf{X} \cdot \mathbf{Y} = \mathbf{Y} \cdot \mathbf{X} = (\mathbf{u} \wedge \mathbf{w}) \cdot (\mathbf{x} \wedge \mathbf{y} \wedge \mathbf{z}) = (\mathbf{u} \cdot (\mathbf{w} \cdot (\mathbf{x} \wedge \mathbf{y} \wedge \mathbf{z}))) \quad (4.17)$$

This result simply reduces the grade of \mathbf{Y} for each common dimension *from the right end* of \mathbf{X} (while dealing with signs from anticommutative swaps in \mathbf{Y}), else is $\mathbf{X} \cdot \mathbf{Y} = 0$ for vectors in \mathbf{X} not found in \mathbf{Y} . The only remaining step is to illustrate how to compute the inner product of two multivectors. Given n -vectors \mathbf{W} , \mathbf{X} , \mathbf{Y} , and \mathbf{Z} the inner (and outer) product distributes over addition and is the equivalent of all product combinations:

$$\begin{aligned} (\mathbf{W} + \mathbf{X}) \cdot (\mathbf{Y} + \mathbf{Z}) &= (\mathbf{W} \cdot \mathbf{Y}) + (\mathbf{W} \cdot \mathbf{Z}) + (\mathbf{X} \cdot \mathbf{Y}) + (\mathbf{X} \cdot \mathbf{Z}) \\ (\mathbf{W} + \mathbf{X}) \wedge (\mathbf{Y} + \mathbf{Z}) &= (\mathbf{W} \wedge \mathbf{Y}) + (\mathbf{W} \wedge \mathbf{Z}) + (\mathbf{X} \wedge \mathbf{Y}) + (\mathbf{X} \wedge \mathbf{Z}) \end{aligned} \quad (4.18)$$

When the inner products are zero (means orthogonal), then the outer product is equivalent to the geometric product, and thus raises the grade of all product pairs. Note, the definitions in the section apply only when all the vectors are orthonormal, but always applies for qubits!!

4.3.7 Outer Product and Inner Product Examples

The best way to form a concrete understanding for the inner and outer products is to give some specific examples. Table 4.2 shows all combinations for multivectors X and Y for $X \wedge Y$ and $X \cdot Y$ in \mathbf{G}_2 . Notice how the inner product is symmetrical.

In Table 4.2 the inner and outer product are never both zero. The geometric product is simply the sum of the corresponding cell from both inner and outer product tables. The same operators are populated for \mathbf{G}_3 as shown in Table 4.3. Notice the \mathbf{G}_3 cases where the inner and outer product are *both* zero and geometric product is not because $A B \neq A \cdot B + A \wedge B$.

Table 4.2: Comparison of outer and inner products for \mathbf{G}_2

$X \wedge Y$		Y			
		+1	a	b	a b
X	+1	+1	a	b	a b
	a	a	0	a b	0
	b	b	-a b	0	0
	a b	a b	0	0	0

$X \cdot Y$		Y			
		+1	a	b	a b
X	+1	0	0	0	0
	a	0	+1	0	b
	b	0	0	+1	-a
	a b	0	b	-a	-1

Table 4.3: Outer and Inner product pairs for \mathbf{G}_3

$X \wedge Y$		Y							
		+1	a	b	c	a b	a c	b c	a b c
X	+1	+1	a	b	c	a b	a c	b c	a b c
	a	a	0	a b	a c	0	0	a b c	0
	b	b	-a b	0	b c	0	-a b c	0	0
	c	c	-a c	-b c	0	a b c	0	0	0
	a b	a b	0	0	a b c	0	0^1	0^1	0
	a c	a c	0	-a b c	0	0^1	0	0^1	0
	b c	b c	a b c	0	0	0^1	0^1	0	0
	a b c	a b c	0	0	0	0	0	0	0

$X \cdot Y$		Y							
		+1	a	b	c	a b	a c	b c	a b c
X	+1	0	0	0	0	0	0	0	0
	a	0	+1	0	0	b	c	0	b c
	b	0	0	+1	0	-a	0	c	-a c
	c	0	0	0	+1	0	-a	-b	a b
	a b	0	b	-a	0	-1	0	0	-c
	a c	0	c	0	-a	0	-1	0	b
	b c	0	0	c	-b	0	0	-1	-a
	a b c	0	b c	-a c	a b	-c	b	-a	+1

Please remember these results are *only valid for orthonormal vector sets*. Also all of the results quoted in the dissertation depend on the geometric product properties, but do not always stipulate if the result is due to inner product or outer product features. So in general

¹ $A B \neq A \cdot B + A \wedge B$ when A is not a vector because $(ab)(bc) = ac$ yet $(ab) \cdot (bc) = (ab) \wedge (bc) = 0$

the outer or inner product will only be specifically called out when required, otherwise one can assume the geometric product is being used by equations and tools.

4.4 Geometric Algebra Tools for Boolean Logic

Using the GA framework so far established, this section develops how to express all of the Boolean logic primitives of geometric algebra, proving that it is universal or Boolean complete. The result has an unusual XOR-like symmetry for both multiplication and addition. Subsequently, several tools are developed to automate the anticommutative and Boolean logic mapping rules of GA.

4.4.1 Boolean Logic in GA

The approach for implementing Boolean logic in geometric algebra requires the mapping of binary values to scalars in the algebra. Using Manthey's conventions, the only scalar values available are the discrete value set $\langle A \rangle_0 = \{-1, 0, +1\}$. Table 4.4 shows the usual multiplication and addition tables for all the combinations of two input vectors for these values. To maintain the group closure properties, the addition table is implemented as modulo 3, in that $\{-1, 0, +1\}$ is isomorphic to $\{0, 1, 2\}$, but this will prove to have no effect on the results presented in this work. The highlighted cells are the non-zero input conditions.

Table 4.4: Addition and Multiplication Tables for \mathbf{G}_2

+	0	1	-1
0	0	1	-1
1	1	-1	0
-1	-1	0	1

*	0	1	-1
0	0	0	0
1	0	1	-1
-1	0	-1	1

Manthey has previously shown that “0” has the special meaning of “cannot occur.” This insight (which will be supported later) is that a “0” as an *input* will neither occur nor have an effect. Therefore, the value “0” will be excluded as a possible input value and the two Boolean logic values {True, False} will be mapped to the symmetric values {+1, -1} respectively. This mapping can be simplified and displayed as the binary set {+, -} by removing the redundant integer 1. Table 4.5 contains the results rewritten using only these symmetric mapping conventions, plus short descriptive phrases for each behavior. Notice the off-diagonal symmetry for both the multiplication and addition logic tables. Due to this symmetry, vector division is the same as multiplication, or $\mathbf{a/b} = \mathbf{a b}$, just as addition and subtraction are identical for the Galois Fields GF(2). Subtraction is discussed in more detail later in the section on Cartesian distance.

Table 4.5: Binary Mapping Tables for Operators in \mathbf{G}_2

+	+	-
+	-	0
-	0	+
If same then invert If different then cancel		

*	+	-
+	+	-
-	-	+
If same then + If different then -		

Assuming two input vectors $\langle A \rangle_1 = \{\mathbf{a}, \mathbf{b}\}$, the multiplication (*) table is equivalent to the logic XNOR (or even parity) operation and the addition (+) table is a combination of NAND and NOR logic. Table 4.6 illustrates the logic reasoning for these operations, suggesting this mapping is universal because it includes the equivalent primitives of logic NOT, logic OR,

and logic AND. This means it should be possible to map all logic operators into arithmetic expressions in GA using only addition and multiplication. This assertion is demonstrated later.

Table 4.6: Logic Map Reasoning Table for \mathbf{G}_2

GA Operator	Logic Example	For Conditions
*	$\mathbf{a} \text{ XNOR } \mathbf{b} = +$	if $\mathbf{a} = \mathbf{b}$
	$\mathbf{a} \text{ XNOR } \mathbf{b} = -$	if $\mathbf{a} \neq \mathbf{b}$
+	$\mathbf{a} \text{ NAND } \mathbf{b} = -$	if $\mathbf{a} = \mathbf{b} = +$
	$\mathbf{a} \text{ NOR } \mathbf{b} = +$	if $\mathbf{a} = \mathbf{b} = -$

Based on this initial reasoning, several automatic tools were created to demonstrate that GA can express any logic operation; these tools are described in the following subsections. The final result of the derivation is summarized in Table 4.7. Even though the inputs are restricted to $\{+, -\}$, GA is still three-valued, so an *output* can be either $+$, $-$, or 0 . The main convention will be to assign the value “+” to the logic value “True.” With those conventions, the middle column expresses the conventional mapping of assigning the value “-” to “False.” Alternatively, the right column maps the “Non-True” = “False” value to “0” and later this will be shown to be very useful when additively combining individual decode states.

Table 4.7: Boolean Logic Summary Table for \mathbf{G}_2

Boolean Logic Operation	GA Mapping $\{+, -\}$	GA Mapping $\{+, 0\}$
Identity \mathbf{a}	$\mathbf{a} * 1 = \mathbf{a} + 0 = \mathbf{a}$	$-1 - \mathbf{a}$
NOT \mathbf{a}	$\mathbf{a} * -1 = -\mathbf{a}$	$-1 + \mathbf{a}$
$\mathbf{a} \text{ XOR } \mathbf{b}$	$-\mathbf{a} \mathbf{b}$	$-1 + \mathbf{a} \mathbf{b}$
$\mathbf{a} \text{ OR } \mathbf{b}$	$\mathbf{a} + \mathbf{b} - \mathbf{a} \mathbf{b}$	$-1 - \mathbf{a} - \mathbf{b} + \mathbf{a} \mathbf{b}$
$\mathbf{a} \text{ AND } \mathbf{b}$	$+1 - \mathbf{a} - \mathbf{b} - \mathbf{a} \mathbf{b}$	$+1 + \mathbf{a} + \mathbf{b} + \mathbf{a} \mathbf{b}$

The only immediately obvious expressions in Table 4.7 are those for NOT and XOR, yet the other expressions have a form similar to those using XOR-based Galois Fields. The complex and unintuitive GA logic mapping is best handled by the tools and methodology described in Section 4.4.2. Once this framework is in place, the summary results shown in Table 4.7 are easily comprehensible.

4.4.2 GA Evaluator

The sneak preview of the results presented in Section 4.4.1 is unintuitive due to the XOR-like behaviors, and compounded by the anticommutative properties of GA. Therefore, a 400-line Perl program (called *ga.pl*) was written and works for Sun's UNIX and the Linux systems. This utility assumes that all vectors are orthonormal and simplifies expressions containing vector sums (or the product-of-sums) into a standardized sum-of-products format. This final expression can be displayed, or alternatively a table is produced for all combinations of $\{+,-\}$ over the input vector set. This evaluation table gives a bird's eye view of the equations and the values they produce. Sample outputs from *ga.pl* are shown in Figure 4.6 while Appendix B contains the complete source code with user documentation. The *ga.pl* tool was invaluable for this effort.

The examples shown in Figure 4.6 validate the tool implementation of GA multiplication and addition tables, which are very nearly opposite to the Galois Field logic mappings for GF(2). The key to understanding these tables is to realize that the vertical bars (|) separate the table into three major columns containing: (1) the input enumerations, (2) the input, simplified to sum-of-products, and (3) the final sum's output respectively. Extra spaces were manually inserted for these examples only to align the product columns under the appropriate product

term. All terms in equations and products are sorted into an alphanumeric order. Any sorting order works as long as it is applied consistently and corresponds to a handedness for the coordinate system. Products in expressions are listed in order from lowest to highest grade.

```

ga.pl zeros "a + b"           ← zeros flag prints all table rows
Input equation is a + b      ← a plus b
INPUTS  a b | + a + b | OUTPUT ← output = sum of a and b
*****
ROW 00: - - | - - | +
ROW 01: - + | - + | 0
ROW 02: + - | + - | 0
ROW 03: + + | + + | -
*****
Row counts for outputs of ZERO=2, PLUS=1, MINUS=1 for TOTAL=4 rows.

Input equation is a b           ← a times b
INPUTS: a b | + a b | OUTPUT   ← output = a XOR b (even parity)
*****
ROW 00: - - | + | +
ROW 01: - + | - | -
ROW 02: + - | - | -
ROW 03: + + | + | +
*****
Row counts for outputs of ZERO=0, PLUS=2, MINUS=2 for TOTAL=4 rows.

Input equation is a + b + c     ← a plus b plus c
INPUTS: a b c | + a + b + c | OUTPUT ← output = sum mod 3 of inputs
*****
ROW 00: - - - | - - - | 0
ROW 01: - - + | - - + | -
ROW 02: - + - | - + - | -
ROW 03: - + + | - + + | +
*****
ROW 04: + - - | + - - | -
ROW 05: + - + | + - + | +
ROW 06: + + - | + + - | +
ROW 07: + + + | + + + | 0
*****
Row counts for outputs of ZERO=2, PLUS=3, MINUS=3 for TOTAL=8 rows.

Input equation is a b c         ← a times b times c
INPUTS: a b c | + a b c | OUTPUT ← output = a XOR b XOR c (odd parity)
*****
ROW 00: - - - | - | -
ROW 01: - - + | + | +
ROW 02: - + - | + | +
ROW 03: - + + | - | -
*****
ROW 04: + - - | + | +
ROW 05: + - + | - | -
ROW 06: + + - | - | -
ROW 07: + + + | + | +
*****

```

Figure 4.6: Sample Outputs from *ga.pl* Tool

```

ga.pl "(b0 + b1)(a0 + a1)"          ← product of sums (no table)
Input equation is (b0 + b1)(a0 + a1)
- a0 b0 - a0 b1 - a1 b0 - a1 b1    ← standardized form and order imposed

ga.pl zeros "(a + b)(a b)"          ← bivector (or spinor) product
Input equation is (a + b)(a b)
INPUTS: a b | - a + b | OUTPUT
*****
ROW 00: - - | + - | 0
ROW 01: - + | + + | -
ROW 02: + - | - - | +
ROW 03: + + | - + | 0
*****
Row counts for outputs of ZERO=2, PLUS=1, MINUS=1 for TOTAL=4 rows.

ga.pl table "(a0 + a1)(b0 + b1)"    ← default command suppresses rows = 0
Input equation is (a0 + a1)(b0 + b1)
INPUTS: a0 a1 b0 b1 | + a0 b0 + a0 b1 + a1 b0 + a1 b1 | OUTPUT
*****
ROW 00: - - - - | + + + + | +
ROW 03: - - + + | - - - - | -
*****
ROW 12: + + - - | - - - - | -
ROW 15: + + + + | + + + + | +
*****
Row counts for outputs of ZERO=12, PLUS=2, MINUS=2 for TOTAL=16 rows.

```

Figure 4.7: Example Products and Table Controls for *ga.pl* Tool

Another major facility is the ability to accept products-of-sums and expand them to the sum-of-products under the GA's anticommutative and simplification rules. These examples are shown in Figure 4.7. *All products require parentheses and internal spaces within products to ensure correct parsing.* Toggling the sign due to each swap while sorting vectors inside products is naturally accomplished using an instrumented bubble sort. When adjacent vector names match, they are simplified out of the product since $\mathbf{a}^2 = 1$. The spinor product $(\mathbf{a} + \mathbf{b})(\mathbf{a} \mathbf{b})$ expands into $\mathbf{a} \mathbf{a} \mathbf{b} + \mathbf{b} \mathbf{a} \mathbf{b} = \mathbf{b} - \mathbf{a} \mathbf{b} \mathbf{b} = \mathbf{b} - \mathbf{a}$ and the tool produces the identical results. Command line arguments control whether: (1) only the equation result is returned (no parameter), (2) the full table is displayed ("zero"), or (3) zero-valued output rows are excluded ("table"). The feature for suppressing zero-valued rows in the printout will be very useful later since the number of zero-valued states is a large percentage of the overall states.

4.4.3 Universal Logic Decode in GA

The unintuitive XOR-like nature of the GA truth tables makes it difficult to design a specific expression with a particular set of desired output states. Consequently, it is necessary to develop a method of defining GA logic expressions with specific output values for specified table rows, which will be used by the table generation tool described in the Section 4.4.4. .

```

Input equation is a + b + a b
INPUTS: a b | + a + b + a b | OUTPUT
*****
ROW 00: - - | - - + | -
ROW 01: - + | - + - | -
ROW 02: + - | + - - | -
ROW 03: + + | + + + | 0 ← AND-like DECODE of ROW “++”
*****
Row counts for outputs of ZERO=1, PLUS=0, MINUS=3 for TOTAL=4 rows.

```

Figure 4.8: First attempt for multivector $a + b + a b$ in G_2

```

ga.pl zeros "1 + a + b + a b"
Input equation is 1 + a + b + a b
INPUTS: a b | + 1 + a + b + a b | OUTPUT
*****
ROW 00: - - | + - - + | 0
ROW 01: - + | + - + - | 0
ROW 02: + - | + + - - | 0
ROW 03: + + | + + + + | + ← DECODES LOGIC “AND” ROW
*****
Row counts for outputs of ZERO=3, PLUS=1, MINUS=0 for TOTAL=4 rows.

```

Figure 4.9: Second attempt produces two-input Logic AND in G_2

In Galois Fields, the logic inclusive OR equation is “ $a + b + a b$.” Entering this equation into the table evaluator tool produced the result shown in Figure 4.8. The result is meaningful because the indicated row is that specified by logic AND. Next, the result is converted to standard logic AND by adding “+1” to the output values to produce the result in Figure 4.9. This expression specifies “True” for one row out of four when both inputs are “True”, thus selecting or “decoding” row 3_{10} (base ten) (or row 11_2 in base two). Notice that the table row numbers from the *ga.pl* tool are always printed in base ten.

By multiplying by “-1” the output is inverted, producing the logic NAND operator, because False = “-” is returned for the output only on row 3. This result is displayed in Figure 4.10.

```

ga.pl zeros "(1 + a + b + a b)(-1)"
Input equation is (1 + a + b + a b) -1
INPUTS: a b | - 1 - a - b - a b | OUTPUT
*****
ROW 00: - - | - + + - | 0
ROW 01: - + | - + - + | 0
ROW 02: + - | - - + + | 0
ROW 03: + + | - - - - | - ← DECODES LOGIC NAND
*****
Row counts for outputs of ZERO=3, PLUS=0, MINUS=1 for TOTAL=4 rows.

```

Figure 4.10: Inversion of Logic AND Produces 2-input Logic NAND in \mathbf{G}_2

Further experiments using the *ga.pl* tool revealed the logic OR and logic NOR expressions shown in Figure 4.11. Notice the similarity of the algebraic form ($\pm a \pm b \pm a b$) for these logic expressions (logic AND, logic NAND, logic OR, and logic NOR).

```

INPUTS: a b | + a + b - a b | OUTPUT
*****
ROW 00: - - | - - - | 0 *NOTE: mod 3 sums to 0
ROW 01: - + | - + + | + ← DECODES LOGIC "OR" ROWS
ROW 02: + - | + - + | + ← DECODES LOGIC "OR" ROWS
ROW 03: + + | + + - | + ← DECODES LOGIC "OR" ROWS
*****

INPUTS: a b | - a - b + a b | OUTPUT
*****
ROW 00: - - | + + + | 0 *NOTE: mod 3 sums to 0
ROW 01: - + | + - - | - ← DECODES LOGIC "NOR" ROWS
ROW 02: + - | - + - | - ← DECODES LOGIC "NOR" ROWS
ROW 03: + + | - - + | - ← DECODES LOGIC "NOR" ROWS
*****

```

Figure 4.11: Two-input Logic OR and NOR in \mathbf{G}_2

These results are novel and all expressions in the summary of logic operations for $\mathbf{G}_2 = \text{span}\{\mathbf{a}, \mathbf{b}\}$ in Table 4.8 have a similar form that is indicative of the XOR-like symmetry first seen in the Galois Field expressions in Section 3.5. The similarity of the algebraic form between

logic AND/NAND and logic OR/NOR has its roots in information theoretic principles because both equations subdivide the four overall states into one preferred or “decoded” state with three other remaining states (or vice versa). This symmetry indicates the logic mapping is not AND-dominant (as in Galois Fields) and supports the original assertion that the “add” operator symmetrically contains both NAND-like and NOR-like logic properties.

Table 4.8: Summary of Logic Operations for $\{\pm, 0\}$ in \mathbf{G}_2

Logic Gate	GA Equation	Decodes to	Rest are	Inversion of
a AND b	$(+ 1 + \mathbf{a} + \mathbf{b} + \mathbf{a} \mathbf{b})$	Single +	0	NAND below
a NAND b	$(- 1 - \mathbf{a} - \mathbf{b} - \mathbf{a} \mathbf{b})$	Single -	0	AND above
a OR b	$(+ \mathbf{a} + \mathbf{b} - \mathbf{a} \mathbf{b})$	Single 0	+	NOR below
a NOR b	$(- \mathbf{a} - \mathbf{b} + \mathbf{a} \mathbf{b})$	Single 0	-	OR above
a XOR b	$(- 1 + \mathbf{a} \mathbf{b})$	Half +s	0	XNOR below
a XNOR b	$(+ 1 - \mathbf{a} \mathbf{b})$	Half -s	0	XOR above

Taking a similar approach for three vectors, $\mathbf{G}_3 = \text{span}\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ produces the logic AND result in Figure 4.12. To individually select a single row, every unique element of every rank in the group must be included. An in-depth review of the row output choices $\{+, -, 0\}$ will demonstrate the significance of the output values in Figure 4.12.

```

ga.pl zero "(a + b + c + a b + a c + b c + a b c)"
Input expression is (+ a + b + c + a b + a c + b c + a b c)
INPUTS: a b c | + a + b + c + a b + a c + b c + a b c | OUTPUT
*****
ROW 00: - - - | - - - + + + - | -
ROW 01: - - + | - - + + - - + | -
ROW 02: - + - | - + - - + - + | -
ROW 03: - + + | - + + - - + - | -
*****
ROW 04: + - - | + - - - - + + | -
ROW 05: + - + | + - + - + - - | -
ROW 06: + + - | + + - + - - - | -
ROW 07: + + + | + + + + + + + | +          ← DECODES LOGIC "AND"
*****
Row counts for outputs of ZERO=0, PLUS=1, MINUS=7 for TOTAL=8 rows.

```

Figure 4.12: Logic AND for {+, -} in G_3

If the traditional Boolean values {True, False} are mapped to {+, -}, then the table represents the three-input logic AND behavior. As Figure 4.13 illustrates, when +1 is added to any equation producing only values {+, -}, then the mappings change to {-, 0} with corresponding gate inversion (AND \rightarrow NAND). The point is that the choice of mappings for {True, False} is arbitrary yet self-consistent and the choice depends on the context where the equation is to be used.

```

ga.pl zero "(1 + a + b + c + a b + b c + a c + a b c)"
Input equation is (1 + a + b + c + a b + b c + a c + a b c)
INPUTS: a b c | + 1 + a + b + c + a b + a c + b c + a b c | OUTPUT
*****
ROW 00: - - - | + - - - + + + - | 0
ROW 01: - - + | + - - + + - - + | 0
ROW 02: - + - | + - + - - + - + | 0
ROW 03: - + + | + - + + - - + - | 0
*****
ROW 04: + - - | + + - - - - + + | 0
ROW 05: + - + | + + - + - + - - | 0
ROW 06: + + - | + + + - + - - - | 0
ROW 07: + + + | + + + + + + + | -          ← DECODES LOGIC "NAND"
*****
Row counts for outputs of ZERO=7, PLUS=0, MINUS=1 for TOTAL=8 rows.

```

Figure 4.13: Logic NAND for {-, 0} in G_3

Following the same process used for the two-input operators, Table 4.9 summarizes the equations for the three-input logic expressions using $\{+, -\}$ values, followed in the next line with $\{+, 0\}$ version. The invert gates (not shown) are an inversion of each primary equation.

Table 4.9: Summary of Logic Operations in \mathbf{G}_3

Logic Expression	Geometric Algebra Equation	Decodes to	Rest are
a AND b AND c	$(+ \mathbf{a} + \mathbf{b} + \mathbf{c} + \mathbf{a} \mathbf{b} + \mathbf{a} \mathbf{c} + \mathbf{b} \mathbf{c} + \mathbf{a} \mathbf{b} \mathbf{c})$	Single +	-
	$(- 1 - \mathbf{a} - \mathbf{b} - \mathbf{c} - \mathbf{a} \mathbf{b} - \mathbf{a} \mathbf{c} - \mathbf{b} \mathbf{c} - \mathbf{a} \mathbf{b} \mathbf{c})$	Single +	0
a OR b OR c	$(+ \mathbf{a} + \mathbf{b} + \mathbf{c} - \mathbf{a} \mathbf{b} - \mathbf{a} \mathbf{c} - \mathbf{b} \mathbf{c} + \mathbf{a} \mathbf{b} \mathbf{c})$	Single -	+
	$(- 1 - \mathbf{a} - \mathbf{b} - \mathbf{c} + \mathbf{a} \mathbf{b} + \mathbf{a} \mathbf{c} + \mathbf{b} \mathbf{c} - \mathbf{a} \mathbf{b} \mathbf{c})$	Single 0	+
a XOR b XOR c	$(+ \mathbf{a} \mathbf{b} \mathbf{c})$	Half +s	-
	$(- 1 - \mathbf{a} \mathbf{b} \mathbf{c})$	Half +s	0

Further investigation into vector spaces \mathbf{G}_n , where $n \in \{4-10\}$ also produced AND/NAND expressions analogous to the forms found for the 2-3 input vector cases. Expressions for all the combinations of a small vector set were initially entered manually and validated using the *ga.pl* tool. This combination set can be generated for vectors $\{\mathbf{a}, \mathbf{b}, \mathbf{c}, \dots\}$ by expanding the expression $(-1)^n(1 + \mathbf{a})(1 + \mathbf{b})(1 + \mathbf{c}) \dots$ which creates the sum of all terms, with appropriate signs. Since the number of unique terms grows as $N=2^n$, it was clear that creating a tool to automate the generation all the combinations of a vector set would be useful!

The “GA And Generator” Tool (called *gandg.pl*) produces the logic AND expression for a list of input vectors as an expansion of $(-1)^n(1 + \mathbf{a})(1 + \mathbf{b})(1 + \mathbf{c})$. For an even number of input vectors, the signs are all “+” and, when n is odd, the signs are all “-.” Using UNIX pipes to route the output from *gandg.pl* (see Appendix C) into the *ga.pl* tool produces the

results in Figure 4.14. The results constitute a proof-by-example of the AND-expressions for all vector sets up through ten inputs. Only the summary line is shown for the large vector sets and even though not shown, they all successfully selected (or decoded) only the last row.

Notice in Figure 4.14 that, since the intermediate product results have all the same sign, it is possible to construct a proof for creating the AND gate for any number of vectors as follows.

For n orthonormal vectors in \mathbf{G}_n , the AND operator decodes to the numerically sorted last row (0 thru $N-1 = 2^n-1$) of the evaluation table, when all the input values are True = “+” and all the other rows produce a zero output. The reason for choosing the $\{+, 0\}$ mapping for $\{\text{True}, \text{False}\}$ will be clear in Section 4.4.4 when various decoded rows are added together.

```

gandg.pl "a,b,c,d" | ga.pl table ← pipe into ga.pl showing non-zero rows
INPUTS: a b c d | + 1 + a + b + c + d + a b + a c + ... + a b c d | OUTPUT
*****
ROW 15: + + + + | + + + + + + + + + + + + + + + + | +
*****
Row counts for outputs of ZERO=15, PLUS=1, MINUS=0 for TOTAL=16 rows.

gandg.pl "a,b,c,d,e" | ga.pl table ← gandg.pl inverts all signs when n=odd
INPUTS: a b c d e | - 1 - a - b - c - d - e - a b - ... - a b c d e | OUTPUT
*****
ROW 31: + + + + + | - - - - - - - - - - - - - - - - - - - - - - | +
*****
Row counts for outputs of ZERO=31, PLUS=1, MINUS=0 for TOTAL=32 rows.

gandg.pl "a,b,c,d,e,f" | ga.pl table ← LARGE equation output removed
Row counts for outputs of ZERO=63, PLUS=1, MINUS=0 for TOTAL=64 rows.

gandg.pl "a,b,c,d,e,f,g" | ga.pl table ← LARGE equation output removed
Row counts for outputs of ZERO=127, PLUS=1, MINUS=0 for TOTAL=128 rows.

gandg.pl "a,b,c,d,e,f,g,h" | ga.pl table ← LARGE equation output removed
Row counts for outputs of ZERO=255, PLUS=1, MINUS=0 for TOTAL=256 rows.

gandg.pl "a,b,c,d,e,f,g,h,i" | ga.pl table ← LARGE equation output removed
Row counts for outputs of ZERO=511, PLUS=1, MINUS=0 for TOTAL=512 rows.

gandg.pl "a,b,c,d,e,f,g,h,i,j" | ga.pl table ← LARGE equation output removed
Row counts for outputs of ZERO=1023, PLUS=1, MINUS=0 for TOTAL=1024 rows.

```

Figure 4.14: Validation of logic AND for up to ten vectors using $\{+, 0\}$

Since the AND operator selects 1 out of N states, this maximum selectivity requires the maximum number of distinguishing terms, which is the complete binomial expansion for all combinations of the input vector set of every rank (as defined by Pascal's Triangle). Since $N=2^n$ has only even factors, N is therefore not divisible by 3, which means that the modulo 3 sum of N signs (all + or all -) will always give a non-zero result (i.e. $N \bmod 3 \neq 0$). With similar reasoning, if N is divisible by 4 (\mathbf{G}_{even} where n is even), then all +s always add to "+." Likewise if N is not divisible by 4 (\mathbf{G}_{odd} when n is odd) all -s always add to "+." Therefore, "+" will always be the value output for the last decoded row when all the products are all "+" or all "-." Q.E.D.

The last comment for this proof explains *why* every n-product for the last row is either all "+" or all "-." The GA product operation is equivalent to XNOR, which produces a "+" when two inputs are the same. When last row (N-1) is decoded (all input vectors have value "+"), then every product of every rank must produce a "+." Therefore, only the sign in front of each product determines the final sign of that product term. It is therefore evident that logic AND can be expressed in this way for any number of input vectors, based on the algorithm used by the *gandg.pl* tool, which creates the complete combination set and assigns "+" and "-" signs correctly to all terms respectively for \mathbf{G}_{even} and \mathbf{G}_{odd} . It is also evident that the

AND decode equation for the last row of grade n has the form $(-1)^n \sum_{i=0}^n \langle A \rangle_i$, which is equivalent to the fully expanded product of $(-1)^n (1 + \mathbf{a})(1 + \mathbf{b})(1 + \mathbf{c}) \dots$ for every vector. Likewise, selecting row zero can always be expressed as $(-1)^n (1 - \mathbf{a})(1 - \mathbf{b})(1 - \mathbf{c}) \dots$ etc.

Extending this reasoning, *any* row can be selectively obtained by inverting the input vector in every product term equivalent to the binary “0”s in the row number (base two) as Figure 4.15 demonstrates. The selected row always has the same sign for every product of the final sum and represents a rotation of the hypercube based on the orientation of vector-row reference frame. Inverting an entire equation for any row inverts the final output to produce a “-.”

```

Input equation is (1 - a - b + a b) ← invert both a and b
INPUTS: a b | + 1 - a - b + a b | OUTPUT
*****
ROW 00: - - | + + + + | + ← decodes A-- = (1-a)(1-b) = [+000] = R0
*****

Input equation is (1 - a + b - a b) ← invert a only
INPUTS: a b | + 1 - a + b - a b | OUTPUT
*****
ROW 01: - + | + + + + | + ← decodes A-+ = (1-a)(1+b) = [0+00] = R1
*****

Input equation is (1 + a - b - a b) ← invert b only
INPUTS: a b | + 1 + a - b - a b | OUTPUT
*****
ROW 02: + - | + + + + | + ← decodes A+- = (1+a)(1-b) = [00+0] = R2
*****

Input equation is (1 + a + b + a b) ← AND behavior
INPUTS: a b | + 1 + a + b + a b | OUTPUT
*****
ROW 03: + + | + + + + | + ← decodes A++ = (1+a)(1+b) = [000+] = R3
*****

```

Figure 4.15: Arbitrary Row Decode R_k for Vectors in \mathbf{G}_2

Therefore *any* specific row R_k can be selected using the equations $(-1)^n(1 \pm \mathbf{a})(1 \pm \mathbf{b})\dots$, where the signs select the row number. This row-decode technique (and the *gandg.pl* tool) will be used in the Section 4.4.4 to enable the construction of GA expressions for arbitrarily complex logic tables. The table output from *ga.pl* tool can itself be compactly represented as a vector, where each row output R_k is the value in the vector $[R_0, R_1, R_2, R_3, \dots]$ and this vector representation will be used extensively in later sections.

4.4.4 GA Generator

The final step in creating equations that define arbitrary logic expressions is to prove that each row $R_k = (-1)^n(1 \pm \mathbf{a})(1 \pm \mathbf{b})\dots$ is *linearly independent* of the other rows due to the choice of having the undecoded rows sum to zero. Therefore adding (or subtracting) any number of rows will combine linearly to produce the final expression. The row decode process discussed in Section 4.4.3, is used in Figure 4.16 to illustrate the results of adding and subtracting two rows. Since each row contains every product combination (though with differing signs) the sum simply flips signs or cancels for the common n-vectors.

```

Input equation is (1 - a + b - a b) + (1 + a + b + a b)
INPUTS: a b | - 1 - b | OUTPUT
*****
ROW 01: - + | - - | +          ← Contribution of ROW 01 (base 2)
ROW 03: + + | - - | +          ← Contribution of ROW 11 (base 2)
*****
Row counts for outputs of ZERO=2, PLUS=2, MINUS=0 for TOTAL=4 rows.

(1 - a)(1 + b)+(1 + a)(1 + b)= (1 - a + 1 + a)(1 + b)= -1(1 + b)= -1 - b

Input equation is (1 - a + b - a b) + (- 1 - a - b - a b)
INPUTS: a b | + a + a b | OUTPUT
*****
ROW 01: - + | - - | +          ← Contribution of ROW 01 (base 2)
ROW 03: + + | + + | -          ← Contribution of ROW 11 (base 2)
*****
Row counts for outputs of ZERO=2, PLUS=1, MINUS=1 for TOTAL=4 rows.

(1 - a)(1 + b)-(1 + a)(1 + b)= (1 - a - 1 - a)(1 + b)= +a(1 + b)= a + a b

```

Figure 4.16: Addition and Subtraction of ROW 1 and ROW 3 for vectors in G_2

The row decoding and summation techniques are built into a GA-Generator (called *gag.pl*) tool that utilizes the AND generation tool and then routes the results through the *ga.pl*. The tool inputs are an orthonormal vector list followed by the row number list (including signs) or the R_k vector notation. See Appendix D for source code of *gag.pl* tool.

The *gag.pl* tool automatically creates and routes individual row equations into the *ga.pl* simplification tool. The summation process identifies like terms and adds them together using modulo 3 arithmetic rules, which either adjusts the signs or causes terms to cancel formally. This formal cancellation of opposite terms causes the resulting equation always to return the minimal expression, as shown in Figure 4.17.

```

gag.pl "a,b" "+0" zero ← for vectors {a, b} generate a "+" in row 0
INPUTS: a b | + 1 - a - b + a b | OUTPUT ← and print table all rows
*****
ROW 00: - - | + + + + | +
ROW 01: - + | + + - - | 0
ROW 02: + - | + - + - | 0
ROW 03: + + | + - - + | 0
*****
Row counts for outputs of ZERO=3, PLUS=1, MINUS=0 for TOTAL=4 rows.

gag.pl "a,b" "+0,3" zero ← for {a, b} generate a "+" in rows 0 and 3
INPUTS: a b | - 1 - a b | OUTPUT ← and print table with all rows
*****
ROW 00: - - | - - | + ← row decode is (1-a)(1-b)= + 1 - a - b + a b
ROW 01: - + | - + | 0
ROW 02: + - | - + | 0
ROW 03: + + | - - | + ← row decode is (1+a)(1+b)= + 1 + a + b + a b
*****
Row counts for outputs of ZERO=2, PLUS=2, MINUS=0 for TOTAL=4 rows.

```

Figure 4.17: GA Generator Tool Examples

There is exactly one equation in the sum-of-products format for each specific logic table because every row equation is a unique combination of $(1 \pm \mathbf{a})(1 \pm \mathbf{b})\dots$, so any specific sum will also be unique. No other formal logic simplification rules are required besides the rules outlined here. This contrasts with traditional Boolean logic equations, which have a variety of implementations for the same logic expression depending on space-time tradeoffs.

4.5 GA Expression Solver Tool

The last important tool is the *gasolve.pl* utility that iteratively solves an equation identity by substituting every possible operator combination into that equation and printing only those combinations matching the outcome goal. The number of combinations grows extremely fast, since the number of possible product terms is $N=2^n$ and varying the coefficient sign $\{-,0,+\}$ for each term is equivalent to a ternary count with 3^N combinations. Therefore, \mathbf{G}_0 has three combinations, \mathbf{G}_1 has $3^2 = 9$ operator combinations $\{00, 0-, 0+, -0, --, -+, +0, +-, ++\}$, \mathbf{G}_2 has $9^2 = 81$ combinations $\{0000 \text{ to } +++++\}$, \mathbf{G}_3 has $81^2 = 6561$ combinations $\{00000000 \text{ to } +++++++\}$, and \mathbf{G}_4 has $6561^2 = 43,046,721$ combinations.

Increasing the vector space dimensionality by one squares the number of operator combinations that must be searched. Due to the number of operators and the execution speed, the *gasolve.pl* tool can currently prove exhaustively identities up to \mathbf{G}_4 , which is the size of two qubits. Faster implementations are possible, but such searching is not an efficient way to explore large spaces. Nonetheless, this tool proved invaluable for discovering that solutions exist for some unintuitive identities. See the source code for *gasolve.pl* in Appendix E.

The arguments to the tool are *gasolve.pl* “<vector list>” “<main with X>” “<goal with X>,” where the equation combinations are substituted for the capital “X” in both the main or goal equations. This tool’s usefulness is illustrated in Figure 4.18. The case $X = 0$ is skipped, so the total count is $3^N - 1$. The first two examples demonstrate this tool’s ability to search for the multiplicative inverse of several multivectors. The third example illustrates that row decode

operators $(1\pm a)$ and $(1\pm b)$ have two sequential applications that are identical to two concurrent ones. The last example shows that multiple applications of specific operators for $X * X = X \wedge X = X$ are the same as one, which are known as the *idempotent* operators.

```

gasolve.pl "a,b" "(1 + a)(X)" "1"    ← find multiplicative inverse of (1+a)?
Attempted 80 with 0 found.           ← NO solution exists

gasolve.pl "a,b" "(1 + a + b)(X)" "1" ← multiplicative inverse of (1+a+b)?
Found Match for X = - 1 + a + b in (1 + a + b)(X) = + 1
Attempted 80 with 1 found.           ← One solution exists

gasolve.pl "a,b" "(X)(X)" "(-1)(X)" ← solve for X * X = - X == X + X
Found Match for X = - 1 in (X)(X) = + 1
Found Match for X = + 1 - a in (X)(X) = - 1 + a
Found Match for X = + 1 + a in (X)(X) = - 1 - a
Found Match for X = + 1 - b in (X)(X) = - 1 + b
Found Match for X = + 1 + b in (X)(X) = - 1 - b
Found Match for X = + 1 - a - b - a b in (X)(X) = - 1 + a + b + a b
Found Match for X = + 1 + a - b - a b in (X)(X) = - 1 - a + b + a b
Found Match for X = + 1 - a + b - a b in (X)(X) = - 1 + a - b + a b
Found Match for X = + 1 + a + b - a b in (X)(X) = - 1 - a - b + a b
Found Match for X = + 1 - a - b + a b in (X)(X) = - 1 + a + b - a b
Found Match for X = + 1 + a - b + a b in (X)(X) = - 1 - a + b - a b
Found Match for X = + 1 - a + b + a b in (X)(X) = - 1 + a - b - a b
Found Match for X = + 1 + a + b + a b in (X)(X) = - 1 - a - b - a b
Attempted 80 with 13 found. ← combinations and products of (1±a) and (1±b)

gasolve.pl "a0,a1" "(X)(X)" "X" ← solve for X * X = X or idempotent
Found Match for X = + 1 in (X)(X) = + 1
Found Match for X = - 1 - a0 in (X)(X) = - 1 - a0
Found Match for X = - 1 + a0 in (X)(X) = - 1 + a0
Found Match for X = - 1 - a1 in (X)(X) = - 1 - a1
Found Match for X = - 1 + a1 in (X)(X) = - 1 + a1
Found Match for X = - 1 - a0 - a1 - a0 a1 in (X)(X) = - 1 - a0 - a1 - a0 a1
Found Match for X = - 1 + a0 - a1 - a0 a1 in (X)(X) = - 1 + a0 - a1 - a0 a1
Found Match for X = - 1 - a0 + a1 - a0 a1 in (X)(X) = - 1 - a0 + a1 - a0 a1
Found Match for X = - 1 + a0 + a1 - a0 a1 in (X)(X) = - 1 + a0 + a1 - a0 a1
Found Match for X = - 1 - a0 - a1 + a0 a1 in (X)(X) = - 1 - a0 - a1 + a0 a1
Found Match for X = - 1 + a0 - a1 + a0 a1 in (X)(X) = - 1 + a0 - a1 + a0 a1
Found Match for X = - 1 - a0 + a1 + a0 a1 in (X)(X) = - 1 - a0 + a1 + a0 a1
Found Match for X = - 1 + a0 + a1 + a0 a1 in (X)(X) = - 1 + a0 + a1 + a0 a1
Attempted 80 with 13 found.

```

Figure 4.18: Examples of *gasolve.pl* for \mathbf{G}_2

Idempotent operators (denoted as $P_k = -R_k$) are important to identify in geometric algebra because all Boolean projection operators P_k are idempotent [35]. Idempotent operators also have a unique square root that must be their inverse, which is similar to the property $\pm 1^2 = +1$.

The idempotent properties are inherent in the row decode equations $R_{0-3} = (-1)^2(1\pm\mathbf{a})(1\pm\mathbf{b})$ because, as will be shown, all their factors $(-1 \mp \mathbf{a})$ and $(-1 \mp \mathbf{b})$ are also idempotent.

The eight combinations of $P_k = (-1\pm\mathbf{a}\pm\mathbf{b}\pm\mathbf{a} \mathbf{b})$ in the third example in Figure 4.18 cannot be derived using the four combinations of the standard product order $-(1\pm\mathbf{a})(1\pm\mathbf{b})$. The other four combinations come from the *reversed product order* combinations $-(1\pm\mathbf{b})(1\pm\mathbf{a})$. This order swapping converts to the complement of the decoded row and switches from the linearly independent row mapping of $\{+, 0\}$ to the traditional Boolean logic mapping $\{+, -\}$.

Table 4.10: Product terms $(1\pm\mathbf{a})(1\pm\mathbf{b})$ and $(1\pm\mathbf{b})(1\pm\mathbf{a})$ show phase combinations for \mathbf{G}_2

Standard Product Order $R_{0-3} = (1\pm\mathbf{a})(1\pm\mathbf{b})$	Reverse Product Order $R_{4-7} = (1\pm\mathbf{b})(1\pm\mathbf{a})$
Input expression is $(1 - a)(1 - b)$ INPUTS: a b + 1 - a - b + a b OUT ***** ROW 00: - - + + + + + ROW 01: - + + + - - 0 ROW 02: + - + - + - 0 ROW 03: + + + - - + 0 *****	Input expression is $(1 - b)(1 - a)$ INPUTS: a b + 1 - a - b - a b OUT ***** ROW 00: - - + + + - - ROW 01: - + + + - + - ROW 02: + - + - + + - ROW 03: + + + - - - + *****
Input expression is $(1 - a)(1 + b)$ INPUTS: a b + 1 - a + b - a b OUT ***** ROW 00: - - + + - - 0 ROW 01: - + + + + + + ROW 02: + - + - - + 0 ROW 03: + + + - + - 0 *****	Input expression is $(1 + b)(1 - a)$ INPUTS: a b + 1 - a + b + a b OUT ***** ROW 00: - - + + - + - ROW 01: - + + + + - - ROW 02: + - + - - - + ROW 03: + + + - + + - *****
Input expression is $(1 + a)(1 - b)$ INPUTS: a b + 1 + a - b - a b OUT ***** ROW 00: - - + - + - 0 ROW 01: - + + - - + 0 ROW 02: + - + + + + + ROW 03: + + + + - - 0 *****	Input expression is $(1 - b)(1 + a)$ INPUTS: a b + 1 + a - b + a b OUT ***** ROW 00: - - + - + + - ROW 01: - + + - - - + ROW 02: + - + + + - - ROW 03: + + + + - + - *****
Input expression is $(1 + a)(1 + b)$ INPUTS: a b + 1 + a + b + a b OUT ***** ROW 00: - - + - - + 0 ROW 01: - + + - + - 0 ROW 02: + - + + - - 0 ROW 03: + + + + + + + *****	Input expression is $(1 + b)(1 + a)$ INPUTS: a b + 1 + a + b - a b OUT ***** ROW 00: - - + - - - + ROW 01: - + + - + + - ROW 02: + - + + - + - ROW 03: + + + + + - - *****

The relationships for the row decode operators R_k shown in Table 4.10 were only discovered using the *gasolve.pl* tool. Many other operator relationships have been discovered and such equation solving constitutes either an existence proof or provides an exhaustive constructive proof when n is small. The idempotent operators are important because they will later relate the row decode states R_k to the eigenvectors and projection operators for the system.

4.6 Cartesian Distance Metric

Cartesian distance is the traditional metric for the distance between two points in a space. This distance definition relies on the scalar coefficient differences for $N=2^n$ n-vectors. The first step in comparing two points is simply taking the coefficient difference between each matching n-vector dimension using the differences in Table 4.11. The max difference is ± 1 because the only scalars are $\{-1, 0, +1\}$. Identical scalars produce a zero difference.

Table 4.11: Difference between two Scalars for \mathbf{G}_7

Difference $a - b$		b		
		-1	0	+1
a	-1	0	-1	+1
	0	+1	0	-1
	+1	-1	+1	0

The most effective way to visualize the alignment of matching dimension coefficients is first to introduce a ternary number labeling notation for \mathbf{G}_n expressions in the standardized sort order. The convention adopted here for full \mathbf{G}_n equations is to place the least significant grade to the left and the most significant to the right (i.e. $c_0\mathbf{1} + c_1\mathbf{a} + c_2\mathbf{b} + c_3\mathbf{a}\mathbf{b}$). Represent

only the scalar coefficients $c_i \in \{-, 0, +\}$ for each term as an *ordered set* of ternary-valued numbers $\{c_3 c_2 c_1 c_0\}$, with the *least significant* value on the *right* and *higher significance* to the *left*. This compact set notation aligns the coefficients for matching dimensions and removes redundant dimension labeling information not required for distance computation. For example for $\{c_1 c_0\}$: $+1 + 0 \mathbf{a} = \{0 +\}$ where $c_0=+1$ and $c_1=0$; $0 + \mathbf{a} = \{+ 0\}$ where $c_0 = 0$ and $c_1 = +1$; and $+1 + \mathbf{a} = \{+ +\}$ where $c_0 = +1$ and $c_1 = +1$. Also shown in the extreme right column of Table 4.12 are the corresponding *table vector notation* $[R_0 R_1]$, which is a *dual representation for the algebraic set notation* of $\{c_1 c_0\}$ for any \mathbf{G}_n . Don't get confused!

Table 4.12: Ternary Number Label Notation for $\mathbf{G}_1 = \text{span } \{\mathbf{a}\}$

Equation for X	Eqn Label $\{c_1 c_0\}$	Additive Inverse $-\mathbf{X}$	Multiplicative Inverse \mathbf{X}^{-1}	Cartesian Distance	Table Output $[R_0 R_1]$
0	$\{0 0\}$	$\{0 0\}$	none	0	$[0 0]$
$-1 + 0 \mathbf{a}$	$\{0 -\}$	$\{0 +\}$	$\mathbf{X}^{-1} = \mathbf{X}$	$\sqrt{1} = 1$	$[- -]$
$+1 + 0 \mathbf{a}$	$\{0 +\}$	$\{0 -\}$	$\mathbf{X}^{-1} = \mathbf{X}$	$\sqrt{1} = 1$	$[+ +]$
$0 - \mathbf{a}$	$\{- 0\}$	$\{+ 0\}$	$\mathbf{X}^{-1} = \mathbf{X}$	$\sqrt{1} = 1$	$[+ -]$
$0 + \mathbf{a}$	$\{+ 0\}$	$\{- 0\}$	$\mathbf{X}^{-1} = \mathbf{X}$	$\sqrt{1} = 1$	$[- +]$
$-1 - \mathbf{a}$	$\{- -\}$	$\{+ +\}$	none	$\sqrt{2}$	$[+ 0]$
$+1 - \mathbf{a}$	$\{- +\}$	$\{+ -\}$	none	$\sqrt{2}$	$[- 0]$
$-1 + \mathbf{a}$	$\{+ -\}$	$\{- +\}$	none	$\sqrt{2}$	$[0 +]$
$+1 + \mathbf{a}$	$\{+ +\}$	$\{- -\}$	none	$\sqrt{2}$	$[0 -]$

The Cartesian distance between any two multivectors is now computed as simply the square

root of the sums of the coefficient differences squared: $\sqrt{\sum_{i=0}^{N-1} (\text{diff}(c_i))^2}$. The square nullifies

all sign differences because $(\pm 1)^2 = +1$, so the Cartesian distance is simply the square root of

the *number of differing coefficient terms*. The maximum distance in \mathbf{G}_n is $\sqrt{n+1}$, for any equation without zero-valued coefficients ($c_i \neq 0$), and its complement. The additive inverse represents the maximum distance for any expression containing the same terms. Distances less than the maximum can be thought of as a phase difference. The set labeling shorthand for algebraic expressions and the dual table vector notation are used in many later examples.

4.7 Computational Basis and Projectors

Idempotent operators for \mathbf{G}_2 were first mentioned in Section 4.5 and are identified as the *projection operators* P_k where $P_k = -R_k$. *Idempotent operators are important for making measurements in quantum mechanics* because in \mathbf{H}_n every *projection operator* P_k is a *logic decode operator* and vice versa. It is also well known that idempotent projection operators are mathematically related to *eigenvectors* and *eigenvalues* [35]. This section will identify these relationships for geometric algebra and also introduce the underlying topological meaning.

Each projection operator P_k in GA represents the logic-NAND of every input vector in some specific state, and thereby functions as a logic decoding operator. Therefore, they are also equivalent to the *computational basis states* (see more in Section 5.5). An important question is how to find the orthogonal basis states in GA that correspond to the eigenvectors for those projectors? Chris Doran [9] states in a discussion about the geometric algebra equivalent of orthonormal Pauli matrices, that the following properties must be true for an orthonormal frame of eigenvectors $\{\mathbf{e}_k\}$ using $\mathbf{G}_2 = \text{span}\{\mathbf{a0}, \mathbf{a1}\}$. First, the eigenvectors must square to unity $(\mathbf{e}_k)^2 = 1$, which are easily found using the *gasolve.pl* tool in Figure 4.19.

```

gasolve.pl "a0,a1" "(X)(X)" "(1)"
Found Match for X = - 1 in (X)(X) = + 1
Found Match for X = + 1 in (X)(X) = + 1
Found Match for X = - a0 in (X)(X) = + 1
Found Match for X = + a0 in (X)(X) = + 1
Found Match for X = - a1 in (X)(X) = + 1
Found Match for X = + a1 in (X)(X) = + 1
Found Match for X = - a0 - a1 - a0 a1 in (X)(X) = + 1
Found Match for X = + a0 - a1 - a0 a1 in (X)(X) = + 1
Found Match for X = - a0 + a1 - a0 a1 in (X)(X) = + 1
Found Match for X = + a0 + a1 - a0 a1 in (X)(X) = + 1
Found Match for X = - a0 - a1 + a0 a1 in (X)(X) = + 1
Found Match for X = + a0 - a1 + a0 a1 in (X)(X) = + 1
Found Match for X = - a0 + a1 + a0 a1 in (X)(X) = + 1
Found Match for X = + a0 + a1 + a0 a1 in (X)(X) = + 1
Attempted 80 with 14 found

```

Figure 4.19: Solve for eigenvectors where $(\mathbf{e}_k)^2 = 1$

Next, the projection operators $P_k = (1 + \mathbf{e}_k)/2$ must satisfy the relationship $\mathbf{e}_k (1 + \mathbf{e}_k)/2 =$

$(1 + \mathbf{e}_k)/2$. This equation can be simplified for \mathbf{G}_2 because, with $2P_k = P_k + P_k = -R_k = R_k$,

where the row decode R_k is $R_k = 2P_k = 2*(1 + \mathbf{e}_k)/2 = (1 + \mathbf{e}_k)$, and so $P_k = -R_k = -(1 + \mathbf{e}_k)$.

Since projectors P_k are *idempotent* and $R_k = -P_k = (1 + \mathbf{e}_k)$, then the solutions for $\mathbf{e}_k R_k = R_k$

are $\mathbf{e}_k (1 + \mathbf{e}_k) = (1 + \mathbf{e}_k)$, which are found using *gasolve.pl* tool in Figure 4.20. Notice that the

input expression $(1 + X)$ is written as $(1 X)$ because X contains a sign when substituted.

```

gasolve.pl "a0,a1" "(X)(1 X)" "(1 X)"
Found Match for X = - 1 in (X)(1 X) = 0
Found Match for X = + 1 in (X)(1 X) = - 1
Found Match for X = - a0 in (X)(1 X) = + 1 - a0
Found Match for X = + a0 in (X)(1 X) = + 1 + a0
Found Match for X = - a1 in (X)(1 X) = + 1 - a1
Found Match for X = + a1 in (X)(1 X) = + 1 + a1
Found Match for X = - a0 - a1 - a0 a1 in (X)(1 X) = + 1 - a0 - a1 - a0 a1
Found Match for X = + a0 - a1 - a0 a1 in (X)(1 X) = + 1 + a0 - a1 - a0 a1
Found Match for X = - a0 + a1 - a0 a1 in (X)(1 X) = + 1 - a0 + a1 - a0 a1
Found Match for X = + a0 + a1 - a0 a1 in (X)(1 X) = + 1 + a0 + a1 - a0 a1
Found Match for X = - a0 - a1 + a0 a1 in (X)(1 X) = + 1 - a0 - a1 + a0 a1
Found Match for X = + a0 - a1 + a0 a1 in (X)(1 X) = + 1 + a0 - a1 + a0 a1
Found Match for X = - a0 + a1 + a0 a1 in (X)(1 X) = + 1 - a0 + a1 + a0 a1
Found Match for X = + a0 + a1 + a0 a1 in (X)(1 X) = + 1 + a0 + a1 + a0 a1
Attempted 80 with 14 found

```

Figure 4.20: Solve for eigenvectors where $\mathbf{e}_k (1 + \mathbf{e}_k) = (1 + \mathbf{e}_k)$

This result in Figure 4.20 is consistent with Figure 4.19 because the eigenvectors $\mathbf{e}_k = (\pm\mathbf{a}_0\pm\mathbf{a}_1\pm\mathbf{a}_0 \mathbf{a}_1)$ are actually **multivectors** E_k , where $R_k = (1 + E_k) = (+1\pm\mathbf{a}_0\pm\mathbf{a}_1\pm\mathbf{a}_0 \mathbf{a}_1)$ and the idempotent projection operators are $P_k = -(1 + E_k) = (-1\mp\mathbf{a}_0\mp\mathbf{a}_1\mp\mathbf{a}_0 \mathbf{a}_1)$.

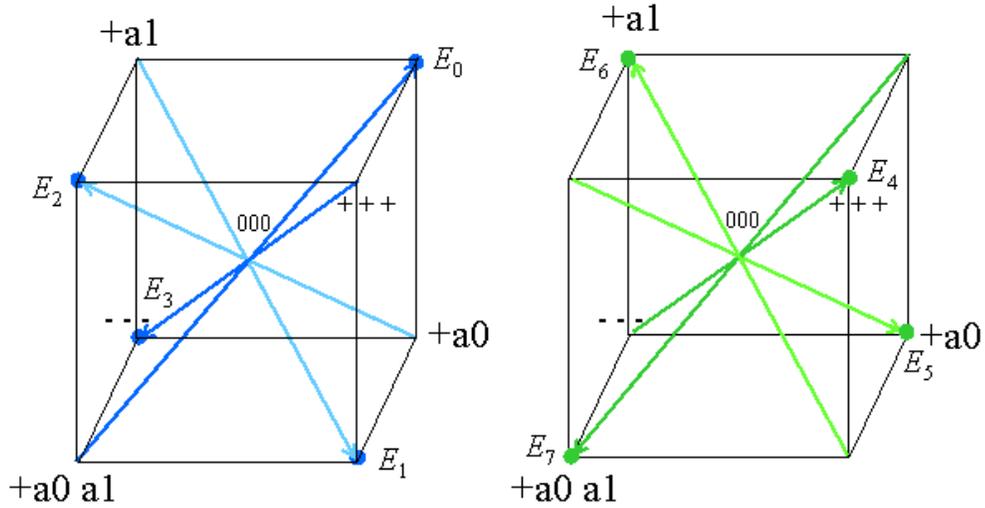


Figure 4.21: Major diagonals form eigenvectors E_{0-3} on left and duals E_{7-4} on right for \mathbf{G}_2

Now that the *eigenmultivectors* $E_k = (\pm\mathbf{a}_0\pm\mathbf{a}_1\pm\mathbf{a}_0 \mathbf{a}_1)$ have been identified in multiple ways, the following geometric interpretation provides the topological meaning. In Figure 4.21, the $E_k = (\pm\mathbf{a}_0\pm\mathbf{a}_1\pm\mathbf{a}_0 \mathbf{a}_1)$ are the four *major cube diagonals* (pointing to the eight corners) formed by all sums of the elements $\{\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_0 \mathbf{a}_1\}$. Four of these solutions match the “-” decode equations of $P_{0-3} = -(1 + E_k) = (-1)(1\pm\mathbf{a}_0)(1\pm\mathbf{a}_1)$ and represent the projection matrix diagonals written as vectors $P_0 = [-000]$, $P_1 = [0-00]$, $P_2 = [00-0]$, and $P_3 = [000-]$. The matching “+” row decodes $R_k = (1 + E_k) = -P_k = (1\pm\mathbf{a}_0)(1\pm\mathbf{a}_1)$ are $R_0 = [+000]$, $R_1 = [0+00]$, $R_2 = [00+0]$, and $R_3 = [000+]$. The partial completeness properties are demonstrated by the sums of $P_0 + P_1 + P_2 + P_3 = [- - - -] = -1$ and $R_0 + R_1 + R_2 + R_3 = [+ + + +] = +1$. The

remaining four eigenvectors are the major diagonals pointing to the *opposite corners* and equivalent to $P_{7-4} = (-1)(1\pm\mathbf{a1})(1\pm\mathbf{a0})$ or $P_7 = [+ - - -]$, $P_6 = [- + - -]$, $P_5 = [- - + -]$, and $P_4 = [- - - +]$ and likewise $R_{7-4} = (1\pm\mathbf{a1})(1\pm\mathbf{a0})$ or $R_7 = [- + + +]$, $R_6 = [+ - + +]$, $R_5 = [+ + - +]$, and $R_4 = [+ + + -]$, which can all be written as linear combinations of the more primitive P_{0-3} and R_{0-3} . The same completeness properties are true: $P_4 + P_5 + P_6 + P_7 = [- - - -] = -1$ and $R_4 + R_5 + R_6 + R_7 = [+ + + +] = +1$ resulting in the important overall completeness properties $P_0 + P_1 + P_2 + P_3 + P_4 + P_5 + P_6 + P_7 = +1$ and $R_0 + R_1 + R_2 + R_3 + R_4 + R_5 + R_6 + R_7 = -1$. This matches the standard required completeness relation $\sum_k P_k = 1$ for projection operators [7]. These results are summarized in Table 4.13 using the row-decode vector notation.

Table 4.13: Eigenvector Summary from $E_k R_k = R_k$ for \mathbf{G}_2

Primary Basis Set				Dual Basis Set			
k =	$E_k = R_{k-1}$	$P_k = -R_k$	$R_k = 1+E_k$	k =	$E_k = R_{k-1}$	$P_k = -R_k$	$R_k = 1+E_k$
0	[0 - - -]	[- 0 0 0]	[+ 0 0 0]	7	[0 + + +]	[- + + +]	[+ - - -]
1	[- 0 - -]	[0 - 0 0]	[0 + 0 0]	6	[+ 0 + +]	[+ - + +]	[- + - -]
2	[- - 0 -]	[0 0 - 0]	[0 0 + 0]	5	[+ + 0 +]	[+ + - +]	[- - + -]
3	[- - - 0]	[0 0 0 - 0]	[0 0 0 + 0]	4	[+ + + 0]	[+ + + -]	[- - - +]
sum	[0 0 0 0]	[- - - -]	[+ + + +]	sum	[0 0 0 0]	[- - - -]	[+ + + +]

These two primary and dual basis sets (identified by corners) form dual non-overlapping tetrahedrons overlaid inside the 3D cube, where the dual(P_k) = P_{7-k} and the dual(R_k) = R_{7-k} . The tetrahedrons shown in Figure 4.22 are geometrically interesting because they represent four equally spaced points, which define six equal-length lines formed by pairs of diagonals on the opposite six faces of the cube. As will be shown geometrically, the row-decode R_k and projection operators P_k represent oriented planes which form the four unique sides of the two tetrahedrons, each of which possesses two orientations.

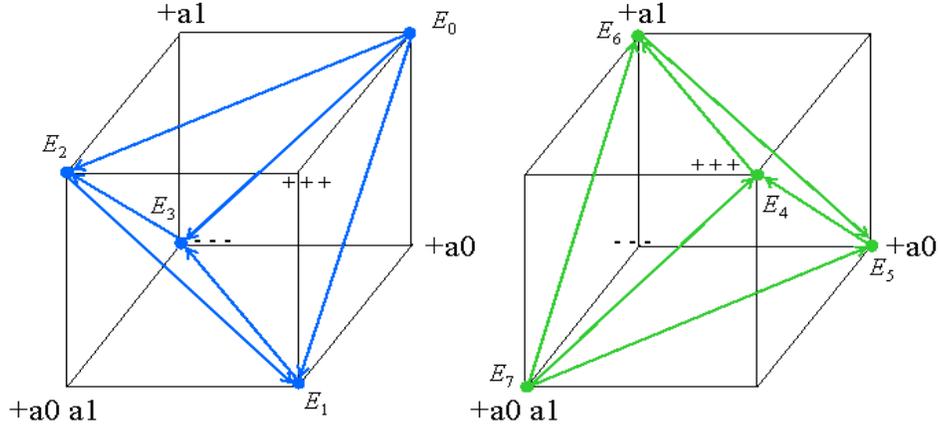


Figure 4.22: Sides of dual tetrahedrons form P_{0-3} on left and duals P_{7-4} on right for \mathbf{G}_2

The primary tetrahedron related to P_{0-3} contains the corner $(-\mathbf{a0} - \mathbf{a1} - \mathbf{a0 a1})$. The dual tetrahedron related to P_{7-4} is formed by the major axis pointing to the opposite corners and contains the corner $(+\mathbf{a0} + \mathbf{a1} + \mathbf{a0 a1})$. See Table 4.14 for a summary of the geometric elements with Cartesian length $\sqrt{2}$ for sides between any two tetrahedron corners.

Table 4.14: Eigenvectors E_K are major axes that form Dual Tetrahedrons in \mathbf{G}_2

Geometric Features		Primary Tetrahedron	Dual Tetrahedron
E_k Basis	→ Corner 1	$E_0 = +\mathbf{a0} + \mathbf{a1} - \mathbf{a0 a1} = [0 \ - \ -]$	$E_7 = -\mathbf{a0} - \mathbf{a1} + \mathbf{a0 a1}$
E_k Basis	→ Corner 2	$E_1 = +\mathbf{a0} - \mathbf{a1} + \mathbf{a0 a1} = [- \ 0 \ -]$	$E_6 = -\mathbf{a0} + \mathbf{a1} - \mathbf{a0 a1}$
E_k Basis	→ Corner 3	$E_2 = -\mathbf{a0} + \mathbf{a1} + \mathbf{a0 a1} = [- \ - \ 0]$	$E_5 = +\mathbf{a0} - \mathbf{a1} - \mathbf{a0 a1}$
E_k Basis	→ Corner 4	$E_3 = -\mathbf{a0} - \mathbf{a1} - \mathbf{a0 a1} = [- \ - \ -]$	$E_4 = +\mathbf{a0} + \mathbf{a1} + \mathbf{a0 a1}$
Front side	$= +\mathbf{a0 a1}$	$(+\mathbf{a0} - \mathbf{a1}) \rightarrow$ Corner 2	$(+\mathbf{a0} + \mathbf{a1}) \rightarrow$ Corner 4
Back side	$= -\mathbf{a0 a1}$	$(-\mathbf{a0} - \mathbf{a1}) \rightarrow$ Corner 4	$(+\mathbf{a0} - \mathbf{a1}) \rightarrow$ Corner 3
Top Side	$= +\mathbf{a1}$	$(-\mathbf{a0} + \mathbf{a0 a1}) \rightarrow$ Corner 3	$(-\mathbf{a0} - \mathbf{a0 a1}) \rightarrow$ Corner 2
Bottom Side	$= -\mathbf{a1}$	$(-\mathbf{a0} - \mathbf{a0 a1}) \rightarrow$ Corner 4	$(+\mathbf{a0} - \mathbf{a0 a1}) \rightarrow$ Corner 3
Right Side	$= +\mathbf{a0}$	$(-\mathbf{a1} + \mathbf{a0 a1}) \rightarrow$ Corner 2	$(+\mathbf{a1} + \mathbf{a0 a1}) \rightarrow$ Corner 4
Left Side	$= -\mathbf{a0}$	$(+\mathbf{a1} + \mathbf{a0 a1}) \rightarrow$ Corner 3	$(+\mathbf{a1} - \mathbf{a0 a1}) \rightarrow$ Corner 2

The equations for the major axes represent an oriented line with Cartesian distance $\sqrt{3}$ passing thru the [0000] center, *pointing toward* (\rightarrow) the corner and pointing *away* from the complement of that corner. Likewise, the diagonals for each side represent a line length of $\sqrt{2}$ *pointing at* (\rightarrow) the same corners. Although line orientations are arbitrary, Figure 4.22 uses three line segments pointing *away* from corner 1 while the other three lines are joined head-to-tail-to-head to form a plane whose right-hand orientation points *toward* corner 1.

The side multivectors in Table 4.14 are multiplied in pairs to form bivectors or oriented planes, which represent the projection and row-decode operators. The remaining combinations produce five duplicates of P_{0-3} and R_{0-3} and twelve additional combinations are variations of P_{7-4} and R_{7-4} . Other vector orientations define the same planes but for different combinations and orientations of vectors. Multiplying the sides in opposite order effectively inverts the orientation of the sides which is equivalent to taking the orientation of the same side in the dual tetrahedron. This can be succinctly expressed as: if $R_k = (Side1)(Side2)$, then the order $R_{7-k} = (Side2)(Side1)$ and likewise, if $P_k = (Side1)(Side2)$, then $P_{7-k} = (Side2)(Side1)$, where *Side2* and *Side1* are multivectors. These relationships are illustrated in the Table 4.15.

Table 4.15: Computational Operators encoded as geometric products for \mathbf{G}_2

Computation Plane	Primary Tetrahedron	Dual Tetrahedron	Equation Label
$P_0 = [- 0 0 0]$	$= (Front)(Top)$	$= (Left)(Bottom)$	$\{- + + -\}$
$P_1 = [0 - 0 0]$	None for orientations	$= (Front)(Top)$	$\{+ - + -\}$
$P_2 = [0 0 - 0]$	$= (Left)(Top)$	None for orientations	$\{+ + - -\}$
$P_3 = [0 0 0 -]$	$= (Front)(Bottom)$	$= (Back)(Top)$	$\{- - - -\}$
$R_0 = [+ 0 0 0]$	$= (Right)(Front)$	$= (Back)(Bottom)$	$\{+ - - +\}$
$R_1 = [0 + 0 0]$	$= (Right)(Back)$	$= (Left)(Front)$	$\{- + - +\}$
$R_2 = [0 0 + 0]$	$= (Back)(Top)$	$= (Front)(Bottom)$	$\{- - + +\}$
$R_3 = [0 0 0 +]$	$= (Left)(Bottom)$	$= (Right)(Top)$	$\{+ + + +\}$

So, geometrically, the row-decode and projection operators represent oriented planes which form the four sides of the two tetrahedrons, each of which possesses two orientations. These planes are oriented at 45° angles off the three separate axes, because the tetrahedron sides are all equilateral triangles based on the diagonals of three sides of a cube.

The multivector eigenvectors E_k require additional comment. Even though all the other properties appear to be true for the E_k eigenvectors, due to the topology of the tetrahedrons, it is clear that the E_k are only pair wise $E_0 \bullet E_3 = E_1 \bullet E_2 = 0$ orthogonal (see axis shading in Figure 4.21). The four major diagonals of a cube cannot be orthogonal to each other because a cube is spanned by only three axes. Additionally, these diagonals are perpendicular to the tetrahedron sides, which intersect at an angle less than 90° to form the tetrahedron. So even though the overall size of the space is $N = 4$, the eigenvectors only occupy a space of $N-1 = 3$. By computing the inner product between all pairs of projection operators we find $P_j \bullet P_k \neq 0$ except for $P_0 \bullet P_3 = P_1 \bullet P_2 = 0$. Also the three main axes of the cube are not orthogonal because $\mathbf{a} \bullet (\mathbf{a} \mathbf{b}) \neq 0$ and $\mathbf{b} \bullet (\mathbf{a} \mathbf{b}) \neq 0$. These orthogonality constraints for eigenvectors and projection operators need to be studied further.

Another interesting point is that the eigenvectors E_k are effectively out-of-phase notch filters where the projection and row-decode operators match each notch. This is related to the spectral decomposition characteristics of projection operators expressed using eigenvectors, and implies that the eigenvectors form the Fourier basis, as well as the computational basis.

Given eigenvectors E_k and projectors P_k , then any *symmetric operator* \mathbf{S} can be written as the sum $\mathbf{S} = \sum_k I_k P_k$, where $I_k \in \{-, 0, +\}$ are the eigenvalues of the eigenvector E_k . As can

already be seen, this is the form for *any expression* in \mathbf{G}_2 denoted as $[-I_0 \quad -I_1 \quad -I_2 \quad -I_3]$

because the current vector notation is actually being expressed as $\mathbf{S} = \sum_k I_k R_k = \sum_k -I_k P_k$.

Using the row-decode vector notation and standard sort order to illustrate the spectral properties of E_k in \mathbf{G}_2 , the left-most vector acts like a low frequency pattern and each vector following in the sorted order doubles that frequency, so $+\mathbf{a0} = [- - + +]$ and $+\mathbf{a1} = [- + - +]$ while inversion creates an out-of-phase pattern $-\mathbf{a0} = [+ + - -]$ and $-\mathbf{a1} = [+ - + -]$.

Eigenvectors and eigenvalues were explored here because they are related to quantum measurement, the Boolean decode of row states, and may be required to design arbitrary operators for \mathbf{G}_n . This analysis also appears to be relevant for larger \mathbf{G}_n because the row-decode primitives R_k are all product combinations of small idempotent operators. This concludes a brief introductory section on eigenvectors and eigenvalues, to show the mathematical relationship of E_k to the topologically derived R_k and P_k used in this research.

4.8 Determinants in Geometric Algebra

Computing the scalar-valued *determinant* of a geometric algebra multivector expression X is valuable because it captures important properties of X in a basis-invariant manner. If $\det(X)=0$, then the expression X is defined as *singular* and the multiplicative inverse $1/X$ is undefined because computing $1/X$ uses the determinant in the denominator (i.e. $1/\det(X)$) as a volume scaling factor. Using the *gasolve.pl* tool for vector \mathbf{x} , several cases of the form $X = (\pm 1 \pm \mathbf{x})$ have been discovered where $1/X$ is not defined, which implies $\det(\pm 1 \pm \mathbf{x})$ must be 0.

Another fact that will be shown later is that *none* of the operators P_k and R_k have defined multiplicative inverses, which mathematically means their determinants must also be zero. In general for \mathbf{G}_n , $\det(P_j)\det(P_k) = \det(P_j P_k)$, which means $\det(P_j P_k) = 0$ if the determinant of any of its factors is zero. So even without defining how to analytically compute the determinant, this implies that the P_k and R_k for all larger spaces must also have $\det(P_j P_k) = 0$ because $\det(P_j) = \det(P_k) = 0$.

Conversely, the *non-singular* cases are defined for $\det(X) \neq 0$, which means that $X^{-1} = 1/X$ exists and the operator is *reversible*. Since the only scalar values defined for \mathbf{G}_n are $\{-1, 0, +1\}$, all *non-singular multivectors* X are *unitary* because they have the property $|\det(X)| = 1$ (Exercise 6.5.2 on page 309 in [7]). This property of unitary operators is actually derived from the formal definition of *unitary*, which is $X^{-1} = (X^*)^T = (X^T)^*$ where X^* is the complex conjugate, X^T is the transpose and X^{-1} is dependent on $1/\det(X)$. So we have shown that all non-singular expressions in \mathbf{G}_n are *unitary* even though the operators X^* and X^T have not been defined. These operators are namely unnecessary in \mathbf{G}_n because the complex conjugate is related to complex numbers and the transpose is defined only for matrices. Unitary operators are Hermitian if they have the property $X = (X^*)^T$, which is true only for eigenvectors, because $X = X^{-1} = (X^*)^T$ or $X^2 = 1$ (page 3 in [25]).

The singular cases were found by searching with *gasolve.pl*, but it is also possible to compute X^{-1} analytically by expressing the determinant for orthonormal vectors \mathbf{a}_i and \mathbf{b}_j to form a

matrix $(i \times j)$ with entries $\mathbf{a}_{ij} = \mathbf{a}_i \cdot \mathbf{b}_j$ which is equivalent to the inner product expression

$\det(\mathbf{a}_{ij}) = \det(\mathbf{a}_i \cdot \mathbf{b}_j) = (\mathbf{a}_n \wedge \dots \wedge \mathbf{a}_1) \cdot (\mathbf{b}_1 \wedge \dots \wedge \mathbf{b}_n)$. This expression can be expanded out into

the equivalent Laplace expansion of the inner product, but this has the same computational

complexity problems as computing the determinant for Hilbert spaces. An alternative

approach for computing X^{-1} would be to find the \mathbf{G}_n equivalent of X^* and X^T .

4.9 Summary of Logic in Geometric Algebra

The foregoing analysis results are both novel and quite remarkable. With the help of some

custom tools, it has been shown that any $\mathbf{G}_{n=2}$ can represent any Boolean logic as *linear*

expressions using only addition and multiplication. The symmetric three-valued choice of $\{-, 0, +\}$

allows a separate undefined state, independent of the two Boolean logic values, which

is semantically similar to the undriven state from tristate logic. The extra “0” state enables

the linear independence of decode states and is illustrated in the Karnaugh map in Table 4.16.

Table 4.16: Linear Independence of Decode States in Karnaugh Map for $\mathbf{G}_2 = \text{span}\{\mathbf{a}, \mathbf{b}\}$

	$-\mathbf{b}$	$+\mathbf{b}$
$-\mathbf{a}$	row decode $00_2 = A_{--} = (1 - \mathbf{a})(1 - \mathbf{b}) = [+000]$	row decode $01_2 = A_{-+} = (1 - \mathbf{a})(1 + \mathbf{b}) = [0+00]$
$+\mathbf{a}$	row decode $10_2 = A_{+-} = (1 + \mathbf{a})(1 - \mathbf{b}) = [00+0]$	row decode $11_2 = A_{++} = (1 + \mathbf{a})(1 + \mathbf{b}) = [000+]$

In traditional Karnaugh maps where values $\{+, -\}$ are assigned to $\{1, 0\}$, the logic inclusive

OR combines terms in a linearly independent fashion, which is evident when engineers use

“+” to denote logic OR. Likewise, when using states $\{-, 0, +\}$, with the meaning of state “0”

being “cannot occur,” individual row/states of a decode table are similarly linearly independent, but the AND decode expression $(-1)^2(1\pm\mathbf{a})(1\pm\mathbf{b})$ for each cell is more complex and related to the idempotent projection operators and eigenvectors.

The ability to express reversible logic operators in a finite linear space using addition and multiplication operators does not come for free. In a vector space \mathbf{G}_n , when n is large, the total number of elements required for maximum selectivity (a single row) grows as $N=2^n$ and the number of operator combinations grows as 3^N . Even though these $(N-n-1)$ linearly independent higher-rank dimensions exist separately from the n input vectors, they are all interlocked via addition as co-occurrences to form a time-independent self-consistent whole. Maintaining self-consistency for an arbitrarily large system indicates this encoding is equivalent to a *singularity* (outside normal spacetime), otherwise spacetime segregation would disrupt any consistency. These ideas are synonymous with the idealized simultaneity principles of co-occurrence and with the concept of quantum-gravity-as-entropy from black hole mechanics mentioned earlier.

As proven earlier, traditional Boolean logic operators cannot be expressed in a reversible linear space unless embedded into a higher dimensional space. With the above reasoning, building classical computers with GA-based logic is not practical because of the large circuit resources implied. Fortunately, such expanded linear spaces are exactly what are required for representing quantum computing, which leads to the topic of Chapter Five!

CHAPTER 5

SINGLE QUBIT REPRESENTED IN GEOMETRIC ALGEBRA

5.1 Qubit as Co-Occurrence of Two States

Much of the research in geometric algebra entails mapping known mathematical properties into its domain; the same approach is taken here. A qubit state is traditionally defined in a Hilbert space as the sum of two complex numbers (\mathbf{H}_2) represented in Dirac's ket notation for vectors ($|\mathbf{f}\rangle = \mathbf{a}|0\rangle + \mathbf{b}|1\rangle$), where $\mathbf{a}, \mathbf{b} \in \mathbf{C}$ and $|\mathbf{a}|^2 + |\mathbf{b}|^2 = 1$) [15]. Topologically this represents a four-dimensional, real-valued space. The best-explored two-level quantum system is the spin- $1/2$ particle that contains the two basis states: *spin-up* (using the notation $|\uparrow\rangle$ or $|0\rangle$) and *spin-down* (using the notation $|\downarrow\rangle$ or $|1\rangle$). The basis $|0\rangle$ and $|1\rangle$ can be observed and represents classical (i.e. non-superimposed) bit states.

Based on this understanding, a qubit can be represented in geometric algebra. A qubit A contains two bit states *that can occur simultaneously* (denoted as orthonormal basis vectors $\{\mathbf{a}_0, \mathbf{a}_1\}$ generating \mathbf{G}_2). That is, a concrete computational bit is now viewed as potentially having simultaneously values of both logic states "0" and "1." Co-occurrence and \mathbf{G}_2 principles dictate this can be expressed very simply as the *sum* of the two independent state vectors of \mathbf{G}_2^- .

$$\text{qubit } A = (\pm \mathbf{a}_0 \pm \mathbf{a}_1) \tag{5.1}$$

This deceptively simple expression is extremely meaningful. It is evident, for example, that a vector set in \mathbf{G}_2 generates (or spans) a linear space with dimension $N = 2^2 = 4$ and contains the elements $\{\pm 1, \mathbf{a0}, \mathbf{a1}, \mathbf{a0 a1}\}$. The size of this space exactly matches the number of dimensions in \mathbf{H}_2 . As Figure 5.1 shows, the next step explores the meaning of the sign in front of each vector using the *ga.pl* evaluator tool.

```

Input expression is (a0 + a1)      → = R0 - R3 = P3 - P0
INPUTS: a0 a1 | + a0 + a1 | OUTPUT
*****
ROW 00: - - | - - | +      both OFF encoded as + = R0
ROW 01: - + | - + | 0      values cancel which means "cannot occur"
ROW 02: + - | + - | 0      values cancel which means "cannot occur"
ROW 03: + + | + + | -      both ON encoded as - = P3
*****
Row counts for outputs of ZERO=2, PLUS=1, MINUS=1 for TOTAL=4 rows.

```

Figure 5.1: Qubit as Co-occurrence of \mathbf{G}_2^- state vectors

The result segregates the outputs into two groups where either (1) both states have the same value or (2) both states are opposite. To study the situation with only one state ON at a time (where one state is + and other is -), Figure 5.2 shows the state difference (+ $\mathbf{a0} - \mathbf{a1}$). Notice how both phases can be expressed as a pair wise difference $P_0 - P_3$ and $P_1 - P_2$, but remember from end of section 4.7 that these sets are pair wise orthogonal: $P_0 \cdot P_3 = P_1 \cdot P_2 = 0$.

```

Input expression is (a0 - a1)      → = R1 - R2 = P2 - P1
INPUTS: a0 a1 | + a0 - a1 | OUTPUT
*****
ROW 00: - - | - + | 0      values cancel which means "cannot occur"
ROW 01: - + | - - | +      a1 ON encoded as + = R1
ROW 02: + - | + + | -      a0 ON encoded as - = P2
ROW 03: + + | + - | 0      values cancel which means "cannot occur"
*****
Row counts for outputs of ZERO=2, PLUS=1, MINUS=1 for TOTAL=4 rows.

```

Figure 5.2: Qubit State Difference produces Classical States in \mathbf{G}_2^-

Using the co-occurrence interpretation, Figure 5.1 and Figure 5.2 suggest that two separate modes exist, each excluding the other's valid states. Using the $\{\text{True}, \text{False}\} \rightarrow \{+, -\}$ mapping, then $+\mathbf{a0}$ means "state $\mathbf{a0}$ is ON" and $-\mathbf{a0}$ means "state $\mathbf{a0}$ is NOT ON." Therefore, we will use the anti-symmetric sum (state difference or opposite signs) to represent the two *classical states*, and the symmetric sum (both same sign) to represent the *superposition states* from quantum mechanics. Table 5.1 summarizes this interpretation of these bimodal results, and these two modes are generally called the two *phases* of the qubit. Notice how addition is used in both in \mathbf{G}_2 and \mathbf{H}_2 representations to denote the linear combination of independent states, i.e. *co-occurrence is the computational way of expressing linear combinations*.

Table 5.1: Summary of Qubit State Meaning

Mode or Phase	Qubit State	State Meaning		Corresponding Hilbert States
opposite states are classical	$A_0 = (+ \mathbf{a0} - \mathbf{a1})$	$\mathbf{a0}$ ON	$\mathbf{a1}$ OFF	$1 0\rangle + 0 1\rangle = 0\rangle$
	$A_1 = (- \mathbf{a0} + \mathbf{a1})$	$\mathbf{a0}$ OFF	$\mathbf{a1}$ ON	$0 0\rangle + 1 1\rangle = 1\rangle$
like states in superposition	$A_+ = (+ \mathbf{a0} + \mathbf{a1})$	$\mathbf{a0}$ ON	$\mathbf{a1}$ ON	$1/\sqrt{2} (1\rangle + 0\rangle)$
	$A_- = (- \mathbf{a0} - \mathbf{a1})$	$\mathbf{a0}$ OFF	$\mathbf{a1}$ OFF	$1/\sqrt{2} (1\rangle - 0\rangle)$

The topological meaning of the multivectors in the 4D space will be explored in Section 5.5. Both algebras use values $\{-1, 0, +1\}$ but with slightly different interpretations. The GA approach appears to support a mathematically and visually simpler approach to expressing qubits with real-valued GA vectors, than the complex-valued Hilbert space coefficients using the ket notation.

5.2 Pseudoscalar is Spinor Operator

The states $\mathbf{G}_2^- = \{\mathbf{a0}, \mathbf{a1}\}$ define both a plane and the pseudoscalar $\mathbf{S}_A = (\mathbf{a0} \mathbf{a1})$, which acts like a spinor on that plane. The spinor works *independently on each vector* in a sum and Table 5.2 summarizes the right-handed application of the spinor on every vector and qubit state. The primary conclusion from Table 5.2 is that the spinor action switches between the classical and superposition phases. This action is identical to the quantum Hadamard gate.

Table 5.2: Summary of Spinor $\mathbf{S}_A = (\mathbf{a0} \mathbf{a1})$ Action on Qubit States in \mathbf{G}_2^-

Start Phase	Qubit State	Each times spinor	Final State	Final Phase
Classical	$+\mathbf{a0} - \mathbf{a1}$	$+\mathbf{a0} (\mathbf{a0} \mathbf{a1}) = +\mathbf{a1}$	$+\mathbf{a0} + \mathbf{a1}$	<i>Superposed</i>
	$-\mathbf{a0} + \mathbf{a1}$	$-\mathbf{a0} (\mathbf{a0} \mathbf{a1}) = -\mathbf{a1}$	$-\mathbf{a0} - \mathbf{a1}$	
<i>Superposed</i>	$+\mathbf{a0} + \mathbf{a1}$	$+\mathbf{a1} (\mathbf{a0} \mathbf{a1}) = -\mathbf{a0}$	$-\mathbf{a0} + \mathbf{a1}$	Classical
	$-\mathbf{a0} - \mathbf{a1}$	$-\mathbf{a1} (\mathbf{a0} \mathbf{a1}) = +\mathbf{a0}$	$+\mathbf{a0} - \mathbf{a1}$	

5.3 Hadamard Transform

The Hadamard transform is a crucial gate in quantum computing because it is used to convert a known classical state (usually $|0\rangle$) into a superposition state. This is often the first step in a quantum computation. The best way to visualize how the Hadamard (or spinor) transform affects the qubit states is to place the states on a plane diagram such as the one shown in Figure 5.3. Since the right-handed spinor rotates any state counter-clockwise by 90° , it also rotates any sum of states by that amount. A spinor rotation moves a qubit state by 90° to the next counter-clockwise corner of the plane, thereby alternately swapping between classical and superposition phases. The two ellipses indicate the 180° rotation produced by inversion.

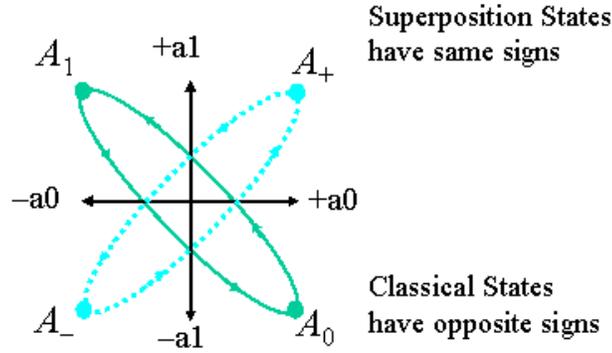


Figure 5.3: Illustration of Qubit States for 90° and 180° Rotations

It is now geometrically obvious why multiplying twice times a spinor \mathbf{S}_A produces an inversion and reinforces the reason why the pseudoscalar \mathbf{I} is equivalent to the unit imaginary

$\mathbf{S}_A = i = \sqrt{-1} = \sqrt{NOT}$ [8], since $\mathbf{S}_A \mathbf{S}_A = (\sqrt{-1})^2 = -1$. The spinor is identical to the

Hadamard transform in \mathbf{G}_2^- because the representation contains all the grade-1 vectors $\langle A \rangle_1$

while the pseudoscalar is grade-2 $\langle A \rangle_2$, so their product always returns a grade-1 result:

$\langle A \rangle_1 \langle A \rangle_2 = \langle A \rangle_1$ in \mathbf{G}_2 . This will be important later when it is shown that alternate basis

states can be selected.

5.4 Pauli Spin Matrix Transforms

The three Pauli spin matrices $\{\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3\}$ are traditionally important in any treatment of

quantum mechanics because, for \mathbf{H}_2 , every 2X2 Hermitian matrix (containing only off-

diagonal elements) can be expressed as a linear combination of the Pauli spin matrices and

the unit matrix using the following matrix equation (not in GA). The matrix equations (5.2)

assumes the standard basis set $\{|0\rangle, |1\rangle\}$ in \mathbf{H}_2 , where b^* is the complex conjugate of b .

$$\begin{bmatrix} a & b \\ b^* & d \end{bmatrix} = \frac{1}{2}(a+d)\mathbf{1} + \frac{1}{2}(b+b^*)\mathbf{s}_1 + \frac{1}{2}i(b-b^*)\mathbf{s}_2 + \frac{1}{2}(a-d)\mathbf{s}_3 \quad (5.2)$$

$$\text{Where } \mathbf{1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{s}_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \mathbf{s}_2 = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad \mathbf{s}_3 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Every physically observable quantity in quantum mechanics corresponds to a Hermitian operator and can be expanded as Pauli matrices. Likewise, any possible unitary evolution in a quantum state due to noise can also be characterized using Pauli matrices [7].

The inspiration to use the interpretive labels for the various kinds of noise was the key to mapping the Pauli matrices into GA as follows. The three kinds of noise: bit flip, phase flip, and simultaneously both bit and phase flips, are illustrated with examples in the three cases and equations (5.3), (5.4), and (5.5):

- 1) Bit flip error: \mathbf{s}_1 causes the overall states to flip flop, which is the action of inversion.

Case	Hilbert notation	Use case	GA equivalent is (-1) or inversion
[a]	$\mathbf{s}_1 0\rangle \rightarrow 1\rangle$	[a]	$(+\mathbf{a0} - \mathbf{a1})(-1) \rightarrow (-\mathbf{a0} + \mathbf{a1})$
[b]	$\mathbf{s}_1 1\rangle \rightarrow 0\rangle$	[b]	$(-\mathbf{a0} + \mathbf{a1})(-1) \rightarrow (+\mathbf{a0} - \mathbf{a1})$

(5.3)

The GA inversion operator has the same meaning as \mathbf{s}_1 (also used in control-not gate in \mathbf{H}_2)

- 2) Phase flip error: \mathbf{s}_3 causes a single state ($\mathbf{a1}$) to flip sign, which is the action of a spinor.

Case	Hilbert notation	Use cases	GA equivalent is ($\mathbf{a0} \mathbf{a1}$) or spinor
[a]	$\mathbf{s}_3 1\rangle \rightarrow - 1\rangle$	[a]&[b]	$(\mathbf{a0} + \mathbf{a1})(-\mathbf{a0} \mathbf{a1}) \rightarrow (+\mathbf{a0} - \mathbf{a1})$
[b]	$\mathbf{s}_3 0\rangle \rightarrow 0\rangle$		
[c]	$-\mathbf{s}_3 1\rangle \rightarrow 1\rangle$	[b]&[c]	$(\mathbf{a0} - \mathbf{a1})(\mathbf{a0} \mathbf{a1}) \rightarrow (+\mathbf{a0} + \mathbf{a1})$

(5.4)

3) Both bit and phase flip error: \mathbf{s}_2 includes both bit and phase flip errors *simultaneously*, which is the *co-occurrence* of the preceding two operators. The shared cases are highlighted in all three tables.

Case	Hilbert notation	Use cases	GA equivalent is $(-1 + \mathbf{a0} \mathbf{a1}) = \mathbf{P}_A$
[a]	$\mathbf{s}_2 0\rangle \rightarrow +i 0\rangle$	[a]&[b]	$(\mathbf{a0} - \mathbf{a1})(-1 + \mathbf{a0} \mathbf{a1}) \rightarrow -\mathbf{a1}$
[b]	$\mathbf{s}_2 1\rangle \rightarrow -i 1\rangle$		

(5.5)

This result is unexpected because the sum of $+\mathbf{a0}$ and $-\mathbf{a0}$ cancels, producing only a single vector, instead of the expected complicated state as found in \mathbf{H}_2 . The meaning of this result is addressed in the next section, on alternative basis sets. We can nevertheless conclude with the strong and positive result that the GA representation of a single qubit contains equivalent mappings for the Pauli spin matrix operators.

5.5 Alternative Basis Sets

Up to this point in the discussion, the qubit representation uses the so-called *standard basis* states, written as the sum of vectors of \mathbf{G}_2^- . Mathematically speaking, basis vectors are hard to grasp unless they are embedded in a geometric context. Simply stated, basis vectors define an intrinsic orientation or reference frame from which the internal states may be observed externally, thereby providing a mechanism for quantum measurement.

In \mathbf{H}_2 , the default **standard basis** of spin-up $|0\rangle$ and spin-down $|1\rangle$ get their names because a spinor is equivalent to a top oriented in a three-dimensional space. If the top is spinning using the right hand rule, then the direction the thumb points (up or down) determines the label. Using the Hadamard Transform, the standard basis states can be reoriented

($H|0\rangle = |0'\rangle$ and $H|1\rangle = |1'\rangle$) into the **dual basis** (also **Hadamard** or **Fourier basis**) shown in equation (5.6) and vice versa ($H|0'\rangle = |0\rangle$ and $H|1'\rangle = |1\rangle$) see Figure 5.4). The \mathbf{H}_2 standard and dual bases are identical to the classical and superposition definitions in our qubit representation.

$$|0'\rangle = 1/\sqrt{2} [|0\rangle + |1\rangle] = 1/\sqrt{2} \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad |1'\rangle = 1/\sqrt{2} [|0\rangle - |1\rangle] = 1/\sqrt{2} \begin{bmatrix} 1 \\ -1 \end{bmatrix} \quad (5.6)$$

Using this approach in our GA qubit representation, the classical states could be labeled left-diagonal and the superposition states labeled right-diagonal, which are equivalent to the standard and dual bases respectively (cf. Figure 5.4). However, there are three different versions of the Hadamard transform for the various bases in \mathbf{H}_2 , but not for GA. A circular polarization basis is also defined for a qubit as $|0''\rangle = 1/\sqrt{2} [|0\rangle + i|1\rangle]$, $|1''\rangle = 1/\sqrt{2} [|0\rangle - i|1\rangle]$, which are simply defined for GA in Table 5.3 using the even grade plane ($\pm 1 \pm \mathbf{a0} \mathbf{a1}$).

Relying on the geometric roots of \mathbf{G}_2 , it is easy to comprehend *all* of these bases for a qubit. This understanding becomes apparent when a qubit state (only odd grade terms \mathbf{G}_2^-) is multiplied by $(-1 + \mathbf{a0} \mathbf{a1})$ containing only even grade terms \mathbf{G}_2^+ . Table 5.3 examines all the possibilities of grade changes due to various operators of the same grade, which thus defines the basis states. A meaningful label for each basis is provided in the top row of the table.

The key to this table is that the odd grade vectors $\mathbf{G}_2^- = \{\mathbf{a0}, \mathbf{a1}\}$ define a plane and also implicitly define the even grade plane $\mathbf{G}_2^+ = \{\pm 1, \mathbf{a0} \mathbf{a1}\}$. In \mathbf{G}_2^- , the **vertical/horizontal** or **Pauli bases** are on-axis while the **left-diagonal basis** (standard) and **right-diagonal basis**

(dual) define the 45° off-axis encodings. Likewise in \mathbf{G}_2^+ , the **direct basis** forms the on-axis encoding, while the 45° off-axis is the **circular basis**. The **direct basis** result is either an invariant (constant ± 1) or a spinor which seems similar to an actual measurement but is reversible and this will be used later in the Toffoli gate derivation. Any basis is equivalent to any other, since any choice is reversible to the starting basis by multiplying by the multiplicative inverse (see 2nd last row in Table 5.3).

Table 5.3: Alternative Reversible Basis Encodings for Qubit A in \mathbf{G}_2

Basis Label \rightarrow	Stand/Dual	Pauli = Ver/Hor	Circular	Direct
Conversion \rightarrow	Start state A	$A(-1 + \mathbf{a0} \mathbf{a1}) =$	$A(\mathbf{a0}) =$	$A(+\mathbf{a0} - \mathbf{a1}) =$
classical 0 classical 1	$+\mathbf{a0} - \mathbf{a1}$	$-\mathbf{a1}$	$(+1 + \mathbf{a0} \mathbf{a1})$	-1
	$-\mathbf{a0} + \mathbf{a1}$	$+\mathbf{a1}$	$(-1 - \mathbf{a0} \mathbf{a1})$	$+1$
superposition + superposition -	$+\mathbf{a0} + \mathbf{a1}$	$+\mathbf{a0}$	$(+1 - \mathbf{a0} \mathbf{a1})$	$+\mathbf{a0} \mathbf{a1}$
	$-\mathbf{a0} - \mathbf{a1}$	$-\mathbf{a0}$	$(-1 + \mathbf{a0} \mathbf{a1})$	$-\mathbf{a0} \mathbf{a1}$
Return to starting state \rightarrow		Pauli $(1 + \mathbf{a0} \mathbf{a1})$	Cir $(\mathbf{a0})$	Dir $(-\mathbf{a0} + \mathbf{a1})$
Cartesian dist from start \rightarrow		$\sqrt{1} = 1$	$\sqrt{4} = 2$	$\sqrt{3}$

An exhaustive analysis of every same-grade basis in \mathbf{G}_2 , by consecutive spinor applications rotating thru all four states, reveals that an even grade spinor always acts as a Hadamard transform. Even grade inversion also works for any \mathbf{G}_n , therefore the Pauli spin operators all function as before. Making a real measurement depends on what basis state the system is in, which means a given state is defined *inside a particular basis*. Mixed grade bases (such as $+\mathbf{a0} + \mathbf{a0} \mathbf{a1}$) are NOT viable basis choices because the states are NOT reversible since they contain a singular factor $1+\mathbf{a1}$. Also, the *trine* basis operators of the form $X = (1 \pm \mathbf{a} \pm \mathbf{a0} \mathbf{a1})$ are solutions to the equality $(X)^3 = 1$ so are 120° apart and are invertible because $X^{-1} = X^2$.

As the discussion of the product of multiple qubits will reveal, the choice of $(\pm \mathbf{a0} \pm \mathbf{a1})$ as the representation of qubits was fortunate, but also one might argue, in favor of it as the *obvious* approach in GA. Note that the Vertical/Horizontal basis encoding is also shown as a way of encoding both noise states (i.e. $\mathbf{a0}$ and $\mathbf{a1}$ in Eq (5.5) using the Pauli spin operator.

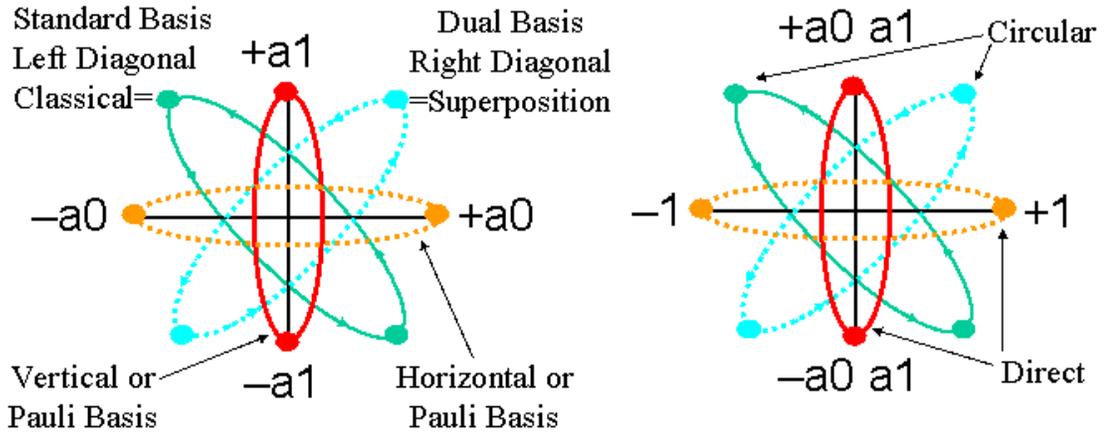


Figure 5.4: On-axis and Off-axis Bases for \mathbf{G}_2^- on left and \mathbf{G}_2^+ on right

Now we will shift gears and discuss the result of using the *irreversible* projection operators P_k , which define the *computational basis*. The standard and computational bases are identical in \mathbf{H}_2 but distinct in \mathbf{G}_2 . This difference occurs because multiplication is AND-like in \mathbf{H}_2 but XOR-like in \mathbf{G}_2 . The corresponding AND-like decode of states in \mathbf{G}_2 is identical to the row-decode formulas, and represent linearly independent states of the smallest topological features. Table 5.4 shows all the computational basis transformations starting from the standard and dual states. If the multiplicative order of the operators is swapped, then the dual of this table is created where all stand-alone terms of $(\pm 1 \pm \mathbf{a1})$ are replaced with $(\pm 1 \pm \mathbf{a0})$.

Table 5.4: Computational Basis Measurement $(A)(1 \pm \mathbf{a0})(1 \pm \mathbf{a1})$ for \mathbf{G}_2

Start State A	$A(1+\mathbf{a0})(1-\mathbf{a1})=$	$A(1-\mathbf{a0})(1+\mathbf{a1})=$	$A(1+\mathbf{a0})(1+\mathbf{a1})=$	$A(1-\mathbf{a0})(1-\mathbf{a1})=$
$A_0 = + \mathbf{a0} - \mathbf{a1}$	$- 1 + \mathbf{a1} = \mathbf{I}^+$	$+ 1 + \mathbf{a1} = \mathbf{I}^-$	$- \mathbf{a0} (+ 1 + \mathbf{a1})$	$\mathbf{a0} (- 1 + \mathbf{a1})$
$A_I = - \mathbf{a0} + \mathbf{a1}$	$+ 1 - \mathbf{a1} = \mathbf{I}^-$	$- 1 - \mathbf{a1} = \mathbf{I}^+$	$- \mathbf{a0} (- 1 - \mathbf{a1})$	$\mathbf{a0} (+ 1 - \mathbf{a1})$
$A_+ = + \mathbf{a0} + \mathbf{a1}$	$- \mathbf{a0} (+ 1 - \mathbf{a1})$	$\mathbf{a0} (- 1 - \mathbf{a1})$	$- 1 - \mathbf{a1} = \mathbf{I}^+$	$+ 1 - \mathbf{a1} = \mathbf{I}^-$
$A_- = - \mathbf{a0} - \mathbf{a1}$	$- \mathbf{a0} (- 1 + \mathbf{a1})$	$\mathbf{a0} (+ 1 + \mathbf{a1})$	$+ 1 + \mathbf{a1} = \mathbf{I}^-$	$- 1 + \mathbf{a1} = \mathbf{I}^+$
End state =>	$A \Rightarrow A_0$	$A \Rightarrow A_I$	$A \Rightarrow A_+$	$A \Rightarrow A_-$
Operator Label	Classical Measurement		Superposition Measurement	
Return Operator	NO return because multiplicative inverse does not exist for $(\pm 1 \pm \mathbf{a1})$			

Several new ideas are introduced in the Table 5.4 that are important for computational basis measurements or transformations for any \mathbf{G}_n . The primary idea is that the results \mathbf{I}^\pm in the table represent a class of sparse invariant operators where the *invariant identity state* $\mathbf{I}^+ = (-1 \pm \mathbf{a1})$ contains only state values $\{+, 0\}$ and the *invariant inversion state* $(+1 \pm \mathbf{a1}) = \mathbf{I}^-$ contains only state values $\{-, 0\}$ as shown in the Table 5.5. The $A R_k$ results represent the *answers*, where the qubit state is changed to the *end state*, which is a many-to-one mapping.

Table 5.5: Sparse Invariant States \mathbf{I}^\pm for \mathbf{G}_2

	Phase 1 invariant using A_I and A_0	Phase 2 invariant using A_0 and A_I
\mathbf{I}^-	Input is $(- \mathbf{a0} + \mathbf{a1})(1 + \mathbf{a0})(1 - \mathbf{a1})$ INPUTS: $\mathbf{a0} \ \mathbf{a1} \ \ + 1 - \mathbf{a1} \ \ \text{OUTPUT}$ ***** ROW 00: $- \ - \ \ + \ + \ \ -$ ROW 01: $- \ + \ \ + \ - \ \ 0$ ROW 02: $+ \ - \ \ + \ + \ \ -$ ROW 03: $+ \ + \ \ + \ - \ \ 0$ *****	Input is $(+ \mathbf{a0} - \mathbf{a1})(1 - \mathbf{a0})(1 + \mathbf{a1})$ INPUTS: $\mathbf{a0} \ \mathbf{a1} \ \ + 1 + \mathbf{a1} \ \ \text{OUTPUT}$ ***** ROW 00: $- \ - \ \ + \ - \ \ 0$ ROW 01: $- \ + \ \ + \ + \ \ -$ ROW 02: $+ \ - \ \ + \ - \ \ 0$ ROW 03: $+ \ + \ \ + \ + \ \ -$ *****
\mathbf{I}^+	Input is $(+ \mathbf{a0} - \mathbf{a1})(1 + \mathbf{a0})(1 - \mathbf{a1})$ INPUTS: $\mathbf{a0} \ \mathbf{a1} \ \ - 1 + \mathbf{a1} \ \ \text{OUTPUT}$ ***** ROW 00: $- \ - \ \ - \ - \ \ +$ ROW 01: $- \ + \ \ - \ + \ \ 0$ ROW 02: $+ \ - \ \ - \ - \ \ +$ ROW 03: $+ \ + \ \ - \ + \ \ 0$ *****	Input is $(- \mathbf{a0} + \mathbf{a1})(1 - \mathbf{a0})(1 + \mathbf{a1})$ INPUTS: $\mathbf{a0} \ \mathbf{a1} \ \ - 1 - \mathbf{a1} \ \ \text{OUTPUT}$ ***** ROW 00: $- \ - \ \ - \ + \ \ 0$ ROW 01: $- \ + \ \ - \ - \ \ +$ ROW 02: $+ \ - \ \ - \ + \ \ 0$ ROW 03: $+ \ + \ \ - \ - \ \ +$ *****

For any grade m -vector \mathbf{X} , all multivector states of the form $(\pm 1 \pm \mathbf{X}) = \mathbf{I}^\pm$ are invariant because the addition operator with ± 1 selects the matching values. These single qubit invariant states are identical to properties of the invariants found for the Bell and magic operators for two qubits. Table 5.5 also shows how the sum of the two phases equals either $+1$ or -1 so it represents a sparse ± 1 result for some operators.

The next idea is that the products $(1 \pm \mathbf{a0})(1 \pm \mathbf{a1})$ are non-reversible because they introduce a loss of state information due to the many-to-one mapping. Because the R_k are singular the qubit is forced to the corresponding end state. Also, the odd grade operators are not commutative, so applying them causes a phase change; $\mathbf{a1}(1 + \mathbf{a0})(1 + \mathbf{a1}) = (1 - \mathbf{a0})(1 + \mathbf{a1})$. Additionally no multiplicative inverse exists for $(1 \pm \mathbf{a0})$ or $(1 \pm \mathbf{a1})$ which is verified with *gasolve.pl* tool, so the previous state cannot be restored using an invertible operator.

```
gasolve.pl "a0,a1" "(1 - a0)(X)" "1" → Attempted 80 with 0 found.
gasolve.pl "a0,a1" "(1 + a0)(X)" "1" → Attempted 80 with 0 found. (5.7)
gasolve.pl "a0,a1" "(1+a0)(1+a1)(X)" "1" → Attempted 80 with 0 found.
```

No operators exist to *undo* any of the products of the computational basis R_k . Searching for other operators produce the same result. So although $(A_+)(1 + \mathbf{a0})(1 + \mathbf{a1}) = (-1 - \mathbf{a1})$ is always true, a solution X *does not* exist for either $(-1 - \mathbf{a1})(X) = A_+$ or $(X)(-1 - \mathbf{a1}) = A_+$.

```
gasolve.pl "a0,a1" "(+ a0 + a1)(X)" "(-1 - a1)"
Found Match for X = + 1 + a0 + a1 + a0 a1 in (+ a0 + a1)(X) = - 1 - a1
Attempted 80 with 1 found.

gasolve.pl "a0,a1" "(-1 - a1)(X)" "(+ a0 + a1)" (5.8)
Attempted 80 with 0 found.

gasolve.pl "a0,a1" "(X)(-1 - a1)" "(+ a0 + a1)"
Attempted 80 with 0 found.
```

The computational basis operators perform an irreversible measurement, because they destroy qubit information due to the many-to-one mapping, but also seen as breaking the co-exclusion state-symmetry in answer (primitive states are not orthogonal $P_0 \cdot P_2 \neq 0 \neq P_1 \cdot P_3$). As a result, even though the answer mapping is one-to-one, it is *irreversible* because the lost phase information is irrecoverable. Once the measurement has been made, then the initial symmetric phase information is lost in the measurement answer expression.

Table 5.6: Computational Basis Measurement Destroys Qubit Symmetry for G_2

Original states for A_I and A_+	States measured with $(1 + a_0)(1 + a_1)$
Input is $A_I = (- a_0 + a_1)$ INPUTS: $a_0 a_1$ $- a_0 + a_1$ OUTPUT ***** ROW 00: - - + - 0 ROW 01: - + + + - ROW 02: + - - - + ROW 03: + + - + 0 *****	Input is $(- a_0 + a_1)(1 + a_0)(1 + a_1)$ INPUTS: $a_0 a_1$ $+ a_0 + a_0 a_1$ OUTPUT ***** ROW 00: - - - + 0 ROW 01: - + - - + ROW 02: + - + - 0 ROW 03: + + + + - *****
Input is $A_+ = (+ a_0 + a_1)$ INPUTS: $a_0 a_1$ $+ a_0 + a_1$ OUTPUT ***** ROW 00: - - - - + ROW 01: - + - + 0 ROW 02: + - + - 0 ROW 03: + + + + - *****	Input is $(+ a_0 + a_1)(1 + a_0)(1 + a_1)$ INPUTS: $a_0 a_1$ $- 1 - a_1$ OUTPUT ***** ROW 00: - - - + 0 ROW 01: - + - - + ROW 02: + - - + 0 ROW 03: + + - - + *****

Table 5.6 gives two examples of the broken phase symmetry. This same scenario appears in the Bell operator for two qubits, where the loss of information is also exactly a phase change.

5.6 Qutrits for Spin-1 Particles

A qutrit state is traditionally defined in a Hilbert space as the sum of three complex numbers

H_3 represented in Dirac's bra-ket notation as vectors $|\Psi\rangle = \mathbf{a}|0\rangle + \mathbf{b}|1\rangle + \mathbf{l}|2\rangle$, where

$\mathbf{a}, \mathbf{b}, \mathbf{l} \in \mathbf{C}$, $|\mathbf{a}|^2 + |\mathbf{b}|^2 + |\mathbf{l}|^2 = 1$, and the standard basis is $\{|0\rangle, |1\rangle, |2\rangle\}$. The space H_3

represents eight dimensions (due to Hermitian generators of SU(3)) and represents spin-1 particles such as photons [15].

Just as for the qubit, the analogous mapping for qutrit A is the co-occurrence of *three* vectors $A = (\pm \mathbf{a0} \pm \mathbf{a1} \pm \mathbf{a2})$ in \mathbf{G}_3 . This mapping forms a linear space of eight dimensions: one scalar dimension $\langle A \rangle_0 = \{\pm 1, 0\}$, three vectors $\langle A \rangle_1 = \{\mathbf{a0}, \mathbf{a1}, \mathbf{a2}\}$, three bivectors $\langle A \rangle_2 = \{\mathbf{a0 a1}, \mathbf{a0 a2}, \mathbf{a1 a2}\}$, and one pseudoscalar $\langle A \rangle_3 = I = \{\mathbf{a0 a1 a2}\}$. Qutrits are not pursued further in this dissertation.

5.7 Qudit for H_d Quantum Systems

It is possible to define the states of an arbitrary d -dimensional Hilbert system H_d using the corresponding 1-vector d -set in geometric algebra with $N=2^d$ equivalent real dimensions. The name for such an H_d system is a “qudit.” The primary way of building a large qudit quantum system is however to use quantum registers containing d qubits rather than one d -qudit. Representing quantum registers in geometric algebra is discussed in the next chapter.

5.8 Phase Shift Transform

A qubit defines a spinor that is identical to the rigid right-handed spinning top in Figure 5.5. The two 1-vectors define the plane of rotation and the orthogonal pseudoscalar defines the axis of rotation, thus forming a 3D structure. The n -vectors are all orthonormal so this top-structure is always present independent of its orientation in a 3D Euclidean space \mathbf{e}_3 (defined by quaternions cf. Chapter Four).

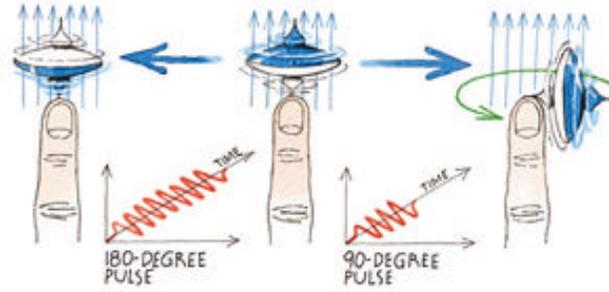


Figure 5.5: Spinor in Spin-down, spin-up, and superposition states (from IBM)

Generalized rotations can be defined in any \mathbf{G}_n because a spinor represents a rotation-dilation in a plane defined by some bivector $\{\mathbf{e}_1\mathbf{e}_2\}$. The general rotation of a vector “ \mathbf{a} ” through an angle \mathbf{q} (in radians) to a new vector \mathbf{a}' is achieved by the dual-sided product:

$$\mathbf{a}' = R\mathbf{a}\tilde{R}, \quad (5.9)$$

where R is called a rotor, defined as the sum of a scalar and a bivector using the equations:

$$R = \mathbf{a} - \mathbf{e}_1\mathbf{e}_2\mathbf{b} \quad \text{and} \quad \tilde{R} = \mathbf{a} + \mathbf{e}_1\mathbf{e}_2\mathbf{b} \quad (\text{where } \mathbf{a} = \cos\mathbf{q}/2 \text{ and } \mathbf{b} = \sin\mathbf{q}/2) \quad (5.10)$$

where the reverse \tilde{R} and scalars $\{\mathbf{a}, \mathbf{b}\}$ are dependent on $1/2$ the angle \mathbf{q} . This overall angle \mathbf{q} is derived from two half-angle reflections in the plane. The formula requires thinking about rotations taking place in a plane rather than around an axis, and works for any grade multivector, in any dimension, of any signature. Alternatively, due to the relationship between sine, cosine, and exponential, if a unit bivector \hat{B} is defined by equation (5.11)

$$\hat{B} = \mathbf{e}_1 \wedge \mathbf{e}_2 / \sin(\mathbf{q}), \quad \text{so } \hat{B}^2 = -1, \quad (5.11)$$

then the rotor components can alternatively be expressed as the familiar exponentials:

$$R = \exp(-\hat{B}\mathbf{q}/2) \quad \text{and} \quad \tilde{R} = \exp(\hat{B}\mathbf{q}/2) \quad (5.12)$$

The sum of a scalar and scaled bivector represents a generalized rotor (equivalent to an angle and a bivector plane). It is well known from qubit measurement theory that even though a qubit has three non-scalar dimensions (with $n=2$ for $N=2^n-1$) with many possible states, only one classical bit of information can be extracted per qubit. The internal phase causes the other states to be indistinguishable from each other as was demonstrated by the alternate basis sets earlier in this chapter, where only one basis set can be active per qubit (which represents one co-exclusion).

The two classical bit-states are topologically equivalent to a particular qubit state, and its inversion, for any choice of basis. All quantum gates reversibly move states around the state space by using arbitrary-phase rotors to align the qubit top inside the three-dimensional Euclidean space \mathbf{E}_3 defined by orthogonal spatial dimensions $\{\mathbf{x}, \mathbf{y}, \mathbf{z}\}$. Phase gates in quantum computing are namely nothing but rotors for arbitrary angles. The quantum computing approach is to align the basis (the axis of the top) along some spatial axis, leaving the remaining phase angles as the computational resource. A Hadamard gate is simply a single sided spinor with a preset angle of $\mathbf{q} = 90^\circ$, and can also be expressed as a generalized double-sided rotor. Measurement is simply the choice of selecting the “computational basis” and applying it as an *irreversible* operator R_k , which returns the answer as a sparse invariant or random ± 1 value. This is not to be confused with the *reversible* “direct basis”, which re-encodes the qubit a constant ± 1 or spinor. For example, $A_1 A_0 = +1$ and $A_1 A_1 = -1$, but $A_1 A_+ = \mathbf{S}_A$, yet are reversible. Also see Table 7.5 for many examples of reversible basis transforms.

This concludes the chapter on single qubit. Quantum registers will be discussed next.

CHAPTER 6

MULTIPLE QUBITS REPRESENTED IN GEOMETRIC ALGEBRA

6.1 Qubit Interaction in Quantum Register as Tensor Product

Quantum registers enable the combining of q qubits to form larger Hilbert spaces, defined by the tensor product (\otimes) of $|\mathbf{y}\rangle = |\mathbf{y}_1\rangle \otimes |\mathbf{y}_2\rangle \otimes \dots \otimes |\mathbf{y}_q\rangle$. Two qubits create an \mathbf{H}_4 space with

$|\mathbf{y}\rangle = \mathbf{a}_{00}|00_2\rangle + \mathbf{a}_{01}|01_2\rangle + \mathbf{a}_{10}|10_2\rangle + \mathbf{a}_{11}|11_2\rangle$ where $\mathbf{a}_{00}, \mathbf{a}_{01}, \mathbf{a}_{10}, \mathbf{a}_{11} \in \mathbf{C}$; the standard basis states are $\{|00_2\rangle, |01_2\rangle, |10_2\rangle, |11_2\rangle\}$ and the unitarity constraint $|\mathbf{a}_{00}|^2 + |\mathbf{a}_{01}|^2 + |\mathbf{a}_{10}|^2 + |\mathbf{a}_{11}|^2 = 1$.

For a general quantum register containing q qubits with basis $B = \{|i\rangle | i \in 0 \leq i < 2^q\}$ and

unitarity constraint $\sum_{i=0}^{2^q-1} |\mathbf{a}_i|^2 = 1$, the general qubit state of the register is $|\mathbf{y}\rangle = \sum_{i=0}^{2^q-1} \mathbf{a}_i |i\rangle$. The

number of states for \mathbf{H}_{2^q} grows as 2^q due to the tensor product.

Compared to the complexity of the above definitions, the equivalent \mathbf{G}_{2^q} definition simply assumes several multivector qubits $\{A, B, C, \dots\}$, each containing the respective orthonormal vector states $\{\mathbf{a0}, \mathbf{a1}\}$, $\{\mathbf{b0}, \mathbf{b1}\}$, $\{\mathbf{c0}, \mathbf{c1}\}$, etc. All the q qubit co-occurrences interact using the geometric product. This notation formally defines the *quantum geometric algebra* $\mathbf{Q}_q = \mathbf{G}_{2^q}$.

$$A B C \dots = (\pm \mathbf{a0} \pm \mathbf{a1})(\pm \mathbf{b0} \pm \mathbf{b1})(\pm \mathbf{c0} \pm \mathbf{c1}) \dots \quad (6.1)$$

For a quantum register \mathcal{Q}_2 with $q = 2$ qubits, this *product of sums* definition produces a corresponding *sum of products* expansion, which is identical to the tensor product definition.

For example, if $A_+ = (+ \mathbf{a0} + \mathbf{a1})$ and $B_+ = (+ \mathbf{b0} + \mathbf{b1})$, then the product $A_+ B_+$ is:

$$A_+ B_+ = (+ \mathbf{a0} + \mathbf{a1})(+ \mathbf{b0} + \mathbf{b1}) = + \mathbf{a0} \mathbf{b0} + \mathbf{a0} \mathbf{b1} + \mathbf{a1} \mathbf{b0} + \mathbf{a1} \mathbf{b1} \quad (6.2)$$

The initial qubit representation as a multivector $A = (\pm \mathbf{a0} \pm \mathbf{a1})$ is fortuitous because *the goal of generating the tensor product \otimes operator to expand the linear space for quantum registers \mathcal{Q}_q is automatically achieved using the geometric (or outer) product*. By choosing the co-occurrence of concrete computation vectors as the representation for \mathcal{Q}_2 , the geometric product $A B$ naturally and obviously replaces the tensor product operator $|\mathbf{y}\rangle = |\mathbf{y}_A\rangle \otimes |\mathbf{y}_B\rangle$. All qubits in this example are in the “+” superposition state and the GA evaluator results for \mathcal{Q}_2 and \mathcal{Q}_3 are shown in Figure 6.1.

```

Input equation is (a0 + a1)(b0 + b1) ← two qubits
INPUTS: a0 a1 b0 b1 | + a0 b0 + a0 b1 + a1 b0 + a1 b1 | OUTPUT
*****
ROW 00: - - - - | + + + + | + → two ways to get +
ROW 03: - - + + | - - - - | - → two ways to get -
*****
ROW 12: + + - - | - - - - | - → two ways to get -
ROW 15: + + + + | + + + + | + → two ways to get +
*****
Row counts for outputs of ZERO=12, PLUS=2, MINUS=2 for TOTAL=16 rows.

Input expression is (a0 + a1)(b0 + b1)(c0 + c1) ← three qubits
INPUTS: a0 a1 b0 b1 c0 c1 | + a0 b0 c0 + a0 b0 c1 + a0 b1 c0 + a0 b1 c1
+ a1 b0 c0 + a1 b0 c1 + a1 b1 c0 + a1 b1 c1 | OUTPUT
*****
ROW 00: - - - - - | - - - - - | + → four ways to get +
ROW 03: - - - - + + | + + + + + + + | - → four ways to get -
ROW 12: - - + + - - | + + + + + + + | - → four ways to get -
ROW 15: - - + + + + | - - - - - | + → four ways to get +
*****
ROW 48: + + - - - - | + + + + + + + | - → four ways to get -
ROW 51: + + - - + + | - - - - - | + → four ways to get +
ROW 60: + + + + - - | - - - - - | + → four ways to get +
ROW 63: + + + + + + | + + + + + + + | - → four ways to get -
*****
Row counts for outputs of ZERO=56, PLUS=4, MINUS=4 for TOTAL=64 rows.

```

Figure 6.1: Non-Zero Qubit States for \mathcal{Q}_2 and \mathcal{Q}_3

Any quantum register in \mathcal{Q}_q defines a product $A B C \dots$ of the q 2-co-occurrences $\{A, B, C, \dots\}$. When this product is expanded into the sum of products, each product combines one state from each qubit to form a total of 2^q q -vectors. These q -vectors are semantically identical to the 2^q basis vectors in \mathbf{H}_{2^q} in that they represent the binary enumerations of all the unique vector products. The grade- $2q$ vector for \mathcal{Q}_q is the pseudoscalar. The grade- q vectors always represent the unique middle column in even rows of Pascal's triangle since $2q$ is always even. Any quantum register \mathcal{Q}_q contains $n = 2q$ orthogonal vectors and $N = 2^{2q} = 4^q$ possible states. As Figure 6.2 and Figure 6.3 illustrate, the last line of each table output summarizes the numbers of various states for 4-6 qubits.

```

ga.pl table "(a0 + a1)(b0 + b1)(c0 + c1)(d0 + d1)" ← four qubits
Input expression is (a0 + a1)(b0 + b1)(c0 + c1)(d0 + d1)
INPUTS: a0 a1 b0 b1 c0 c1 d0 d1 | + a0 b0 c0 d0 + <SNIP> + a1 b1 c1 d1 | OUTPUT
*****
ROW 000: - - - - - - - - | + + + + + + + + + + + + + + | +
ROW 003: - - - - - - + + | - - - - - - - - - - - - - - | -
*****
ROW 012: - - - - + + - - | - - - - - - - - - - - - - - | -
ROW 015: - - - - + + + + | + + + + + + + + + + + + + + | +
*****
ROW 048: - - + + - - - - | - - - - - - - - - - - - - - | -
ROW 051: - - + + - - + + | + + + + + + + + + + + + + + | +
*****
ROW 060: - - + + + + - - | + + + + + + + + + + + + + + | +
ROW 063: - - + + + + + + | - - - - - - - - - - - - - - | -
*****
ROW 192: + + - - - - - - | - - - - - - - - - - - - - - | -
ROW 195: + + - - - - + + | + + + + + + + + + + + + + + | +
*****
ROW 204: + + - - + + - - | + + + + + + + + + + + + + + | +
ROW 207: + + - - + + + + | - - - - - - - - - - - - - - | -
*****
ROW 240: + + + + - - - - | + + + + + + + + + + + + + + | +
ROW 243: + + + + - - + + | - - - - - - - - - - - - - - | -
*****
ROW 252: + + + + + + - - | - - - - - - - - - - - - - - | -
ROW 255: + + + + + + + + | + + + + + + + + + + + + + + | +
*****
Row counts for outputs of ZERO=240, PLUS=8, MINUS=8 for TOTAL=256 rows.

```

Figure 6.2: Non-Zero Qubit States for \mathcal{Q}_4

```

ga.pl table "(a0 + a1)(b1 + b0)(c0 + c1)(d0 + d1)(e0 + e1)" ← 5 qubits
Input expression is (a0 + a1)(b1 + b0)(c0 + c1)(d0 + d1)(e0 + e1)
INPUTS: a0 a1 b0 b1 c0 c1 d0 d1 e0 e1 | + a0 b0 c0 d0 e0 + <SNIP> |OUTPUT
*****
ROW 0000: - - - - - - - - - | - ... 30 ... - | +
<SNIP>
ROW 1023: + + + + + + + + + | + ... 30 ... + | -
*****
Row counts for outs of ZERO=992, PLUS=16, MINUS=16 for TOTAL=1024 rows.

ga.pl table "(a0 + a1)(b1 + b0)(c0 + c1)(d0 + d1)(e0 + e1)(f0 + f1)" ←6 qubits
*****
ROW 0000: - - - - - - - - - - - | + + ... 60 ... + + | +
<SNIP>
ROW 4095: + + + + + + + + + + + | + + ... 60 ... + + | +
*****
Row counts for outs of ZERO=4032, PLUS=32, MINUS=32 for TOTAL=4096 rows.

```

Figure 6.3: Non-Zero Qubit States for Q_5 and Q_6

If the qubit state for each pair $\{a_0, a_1\}$ etc. is analyzed in Figure 6.2 and Figure 6.3, the only decoded non-zero states are the superposition states $\{A_+, A_-, B_+, B_-\}$. This is easy to comprehend because the two zero-valued states per qubit (from compact left-side *product of sums* format) should persist in the corresponding right-side expanded *sum of product* equation format. For any row states in a multivector where $A = 0$, then it always follows that $A B C = 0$. Therefore, this zero-valued state propagation must persist for any quantum register Q_q .

The number of null states for Q_q grows as $4^q - 2^q$ because the overall state space grows exponentially as $N = 2^{2q} = 4^q$ while the number of non-zero states grows slower as 2^q . After only 16 qubits, the valid states are less than 0.01% of the overall state space. Figure 6.4 and the discussion that follows provide the intuition and proof for how such a large number of zero-valued states can persist and be consistently represented in the full sum of products format. This proof is dependent on addition cancellation properties that produce zero values.

6.2 Propagation of Null States

All the intermediate product terms from the previous tables produce the same sign (either all + or all -) for each non-zero state. These states occur whether an individual qubit is in the classical or superposition phase, because in either phase half the states are non-zero. Using the same reasoning used by the AND decode proof in section 4.4.3, the number of intermediate product terms in the final sum for Q_2 is 2^2 . This result contains only even factors, so a zero-value output can *only* be produced by a GA sum from an equal number of “+” and “-” terms because $2^q \bmod 3 \neq 0$. Therefore, any zero-valued output can only be produced by pair-wise cancellation. The visual confirmation of these states for Q_2 can be seen in Figure 6.4.

```

Input expression is (a0 + a1)(b0 + b1)
INPUTS: a0 a1 b0 b1 | + a0 b0 + a0 b1 + a1 b0 + a1 b1 | OUTPUT
*****
ROW 00: - - - - | + + + + | +    ← contains all +s
ROW 01: - - - + | + - + - | 0    ← contains 2 +s and 2 -s
ROW 02: - - + - | - + - + | 0    ← contains 2 +s and 2 -s
ROW 03: - - + + | - - - - | -    ← contains all -s
*****
ROW 04: - + - - | + + - - | 0    ← contains 2 +s and 2 -s
ROW 05: - + - + | + - - + | 0    ← contains 2 +s and 2 -s
ROW 06: - + + - | - + + - | 0    ← contains 2 +s and 2 -s
ROW 07: - + + + | - - + + | 0    ← contains 2 +s and 2 -s
*****
ROW 08: + - - - | - - + + | 0    ← contains 2 +s and 2 -s
ROW 09: + - - + | - + + - | 0    ← contains 2 +s and 2 -s
ROW 10: + - + - | + - - + | 0    ← contains 2 +s and 2 -s
ROW 11: + - + + | + + - - | 0    ← contains 2 +s and 2 -s
*****
ROW 12: + + - - | - - - - | -    ← contains all -s
ROW 13: + + - + | - + - + | 0    ← contains 2 +s and 2 -s
ROW 14: + + + - | + - + - | 0    ← contains 2 +s and 2 -s
ROW 15: + + + + | + + + + | +    ← contains all +s
*****
Row counts for outputs of ZERO=12, PLUS=2, MINUS=2 for TOTAL=16 rows.

```

Figure 6.4: Pair-wise Cancellation of Null States in Q_2

6.3 Pauli Spin and Cross-Qubit Singlets

All qubit interactions \mathcal{Q}_q generated using the product $A B C \dots$ produce the sum of products, where each product term is called a *singlet*. As demonstrated in Section 5.5, applying the *maximal Pauli spin operator* $\mathbf{P}_X = (-1 + \mathbf{x0} \mathbf{x1})$ to some qubit $X = (\pm\mathbf{x0} \pm\mathbf{x1})$ was equivalent to switching from the diagonal basis to the vertical/horizontal basis. A singlet can therefore be created simply by applying the maximal Pauli spin \mathbf{P}_X to every qubit in a quantum register \mathcal{Q}_q followed by the application of the appropriate spinor and sign to choose $\pm\mathbf{x0}$ or $\pm\mathbf{x1}$.

Since the Pauli operators commute, applying the corresponding Pauli operator \mathbf{P}_X in any order produces a unique singlet depending on the starting states.

$$A_0 B_0 \mathbf{P}_A \mathbf{P}_B = A_0 \mathbf{P}_A B_0 \mathbf{P}_B = (\mathbf{a0} - \mathbf{a1})(-1 + \mathbf{a0} \mathbf{a1})(\mathbf{b0} - \mathbf{b1})(-1 + \mathbf{b0} \mathbf{b1}) = \mathbf{a1} \mathbf{b1} \quad (6.3)$$

This means that each cross-qubit entanglement singlet due to \mathbf{P}_X operator can be thought of as qubits expressed in the alternate Pauli basis. Consequently, the *sum of singlets* produced by the product $A B$ is simply the *co-occurrence* of two out-of-phase qubit states, e.g. $\{A_0, A_-\}$ or $\{A_+, A_+\}$, which are represented in the Pauli basis as $(A_0\mathbf{P}_A + A_-\mathbf{P}_A) = (-\mathbf{a0} -\mathbf{a1})$ and $(A_+\mathbf{P}_A + A_+\mathbf{P}_A) = (\mathbf{a1} + \mathbf{a0})$ respectively. This dual interpretation is *somewhat different* from the meaning of these basis states in \mathbf{H}_4 because the computational singlets are not generated using the Pauli operator. This dual interpretation is useful in proofs presented later.

One additional insight about cross-qubit singlets is quite telling when two Pauli spin expressions $(-1 + \mathbf{x0} \mathbf{x1})$ are thought of as a combined operator $\mathbf{P}_A \mathbf{P}_B$. This operator represents *all possible combinations of the spinors (even grade)* and always commutes:

$$\mathbf{P}_A \mathbf{P}_B = (-1 + \mathbf{a0} \mathbf{a1})(-1 + \mathbf{b0} \mathbf{b1}) = (1 - \mathbf{a0} \mathbf{a1} - \mathbf{b0} \mathbf{b1} + \mathbf{a0} \mathbf{a1} \mathbf{b0} \mathbf{b1}) \quad (6.4)$$

6.4 Sequential and Concurrent Application of Spinor Operators

Since quantum registers are expressed as the interaction of two (or more) qubits, formed by the geometric product, it is important to recognize the impact of multiplication order. My single qubit convention requires performing operator multiplication on the right side. This same rule will be applied to quantum registers. For example, in equation (6.5) two qubits are each multiplied by their respective Hadamard transform. Since bivectors always commute, multiplication order is unimportant for spinors. Consequently all Pauli operators commute.

$$A_0 B_0 \mathbf{S}_A \mathbf{S}_B = (+\mathbf{a0} - \mathbf{a1})(+\mathbf{b0} - \mathbf{b1})(\mathbf{a0} \mathbf{a1})(\mathbf{b0} \mathbf{b1}) = (+\mathbf{a0} + \mathbf{a1})(+\mathbf{b0} + \mathbf{b1}) = A_+ B_+ \quad (6.5)$$

The product of the qubit pseudoscalars forms a grade-2q vector and hence the master pseudoscalar $I_2 = (\mathbf{a0} \mathbf{a1})(\mathbf{b0} \mathbf{b1})$ for \mathcal{Q}_2 . Applying I_Q , the overall state changes one

Hadamard operator at a time, thereby *sequentially* placing all the qubits into superposition:

$$I_A I_B \dots = \mathbf{S}_A \mathbf{S}_B \dots = (\mathbf{a0} \mathbf{a1})(\mathbf{b0} \mathbf{b1}) \dots = (\mathbf{a0} \mathbf{a1} \mathbf{b0} \mathbf{b1} \dots) = I_Q \quad (6.6)$$

This prompts the question: Can all the qubits be placed in superposition states *simultaneously* rather than sequentially? This is possible to express if the Hadamard operators are applied *concurrently* using the interpretation of *co-occurrence* as addition, cf. Figure 6.5.

```
ga.pl table "(a0 - a1)(b0 - b1)(a0 a1 + b0 b1)"
INPUTS: a0 a1 b0 b1 | - a0 b0 + a1 b1 | OUTPUT
*****
ROW 01: -- - + | - - | +    qubit A is in superposition
ROW 02: -- + - | + + | -    qubit A is in superposition
*****
ROW 04: - + -- | - - | +    qubit B is in superposition
ROW 07: - + ++ | + + | -    qubit B is in superposition
*****
ROW 08: + - -- | + + | -    qubit B is in superposition
ROW 11: + - ++ | - - | +    qubit B is in superposition
*****
ROW 13: ++ - + | + + | -    qubit A is in superposition
ROW 14: ++ + - | - - | +    qubit A is in superposition
*****
Row counts for outputs of ZERO=8, PLUS=4, MINUS=4 for TOTAL=16 rows.
```

Figure 6.5: Concurrent Hadamard Transform for \mathcal{Q}_2

Figure 6.5 offers several unexpected results. First, two of the four tensor product terms $\{a_0 b_1, a_0 b_1\}$ cancel due to the summation of opposite signs. Second, the only valid states (non-zero) occur when exactly one qubit is in superposition (highlighted in yellow) and all *other* qubit states are classical. This situation works for larger Q_n , as validated using the *ga.pl* tool for Q_3, Q_4, Q_5, Q_6 and Q_7 (see partial results in Figure 6.6). Based on this partial analysis, the same symmetric result is expected to work for any Q_q but no proof is outlined nor any further intuition developed here.

```

Input expression is (a0 - a1)(b0 - b1)(c0 - c1)(a0 a1 + b0 b1 + c0 c1)
INS: a0 a1 b0 b1 c0 c1 | -a0 b0 c1 -a0 b1 c0 -a0 b1 c1 -a1 b0 c0 -a1 b0 c1 -a1 b1 c0 |OUT
*****
ROW 05: -- - + - + | - - + + - - | + <snip-removed 10 rows>
ROW 58: ++ + - + - | + + - - + + | -
*****
Row counts for outputs of ZERO=40, PLUS=12, MINUS=12 for TOTAL=64 rows.

Input expr is (a0 - a1)(b0 - b1)(c0 - c1)(d0 - d1)(a0 a1 + b0 b1 + c0 c1 + d0 d1)
*****
ROW 21: -- - + - + - + | + - - - + + - - - + | + <snip-removed 30 rows>
ROW 234: ++ + - + - + - | + - - - + + - - - + | +
*****
Row counts for outputs of ZERO=192, PLUS=32, MINUS=32 for TOTAL=256 rows.

In(a0 - a1)(b0 - b1)(c0 - c1)(d0 - d1)(e0 - e1)(a0 a1 + b0 b1 + c0 c1 + d0 d1 + e0 e1)
*****
ROW 85: -- - + - + - + - + | + - - - + - - + - + + - + - - - + | + <snip>
ROW 938: ++ + - + - + - + - | - + + + - + + - - - - + - + + - + + - | -
*****
Row counts for outputs of ZERO=864, PLUS=80, MINUS=80 for TOTAL=1024 rows.

```

Figure 6.6: Concurrent Hadamard Transform for Q_3, Q_4 and Q_5 (truncated tables)

The concurrent Hadamard transform is unusual because every valid state has exactly one qubit in superposition, with all the other qubits in the classical phase. Also, the operator doubles the starting number of non-zero row states from four to eight, creating a *less* constrained system. It is interesting to note this state pattern naturally arises from concurrency and symmetry in GA and can be expressed using just the a subset of the

complete tensor expansion, in which all are q -vectors of the same rank. The literature does not succinctly identify this operator but the next sections demonstrate its relationship to the Bell states. These superposition states are *not* separable and this will be discussed in detail in the next two sections.

6.5 Concurrent Hadamard Transform and Ebits

The most interesting result from the above concurrent Hadamard transform applies only to \mathbf{Q}_2 , where the result $(-\mathbf{a}_0 \mathbf{b}_0 + \mathbf{a}_1 \mathbf{b}_1)$ contains *each input vector exactly once*. The form of this result (sum of two bivectors) closely matches one of the Bell states $|00\rangle + |11\rangle$ (less the sign), which is maximally entangled and *not separable*. The Bell states are the important states that represent an “ebit” (for EPR-bit), which allows two remotely located EPR encoded qubits to remain non-locally connected thru entanglement. Most likely a similar Bell state exists for pairs of photons using two qutrits. This discussion of Bell states continues in Section 6.7.

6.6 Concurrent Hadamard Transform and Alternative Bases

For \mathbf{H}_4 , besides the four standard basis $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ and the four dual basis

$\{|0'0'\rangle, |0'1'\rangle, |1'0'\rangle, |1'1'\rangle\}$, four *Bell states* Φ^\pm and Ψ^\pm are also defined as

$$\{ \Phi^+ = \mathbf{a} [|00\rangle + |11\rangle], \Phi^- = \mathbf{b} [|00\rangle - |11\rangle], \Psi^+ = \mathbf{b} [|01\rangle + |10\rangle], \Psi^- = \mathbf{a} [|01\rangle - |10\rangle] \}$$

with normalizing constants $\mathbf{a} = \mathbf{b} = 1/\sqrt{2}$. Likewise, the \mathbf{H}_4 *magic bases* are identical to

the Bell bases except that one of the normalizing constants change slightly to $\mathbf{b} = i/\sqrt{2}$.

This suggests that the Bell and magic states are topologically related. The remainder of this

section will define the Bell and magic states for \mathcal{Q}_q , the operators to enter these bases, and the operators to move between elements of the set.

The following notation will be used to facilitate the derivations used in this section. For qubit multivectors A (and likewise for B), the individual odd grade qubit states are again denoted as $\{A_0 = (\mathbf{a0} - \mathbf{a1}), A_1 = (-\mathbf{a0} + \mathbf{a1}), A_+ = (+\mathbf{a0} + \mathbf{a1}), A_- = (-\mathbf{a0} - \mathbf{a1})\}$. The related even grade expressions are the pseudoscalar (spinor) $I_A = S_A = (\mathbf{a0} \mathbf{a1})$ and the max Pauli spin term $P_A = (-1 + S_A) = (-1 + \mathbf{a0} \mathbf{a1})$. The max pseudoscalar is $I_2 = I_A I_B = S_A S_B$. The cross-qubit spinors $\{S_{00} = (\mathbf{a0} \mathbf{b0}), S_{01} = (\mathbf{a0} \mathbf{b1}), S_{10} = (\mathbf{a1} \mathbf{b0}), S_{11} = (\mathbf{a1} \mathbf{b1})\}$ also represent *entangled bivectors* and are alternatively written as $\{E_{00}, E_{01}, E_{10}, E_{11}\}$. Using this shorthand notation, the traditional classical starting state in \mathcal{Q}_2 is simply the product $A_0 B_0$.

The four cyclic Bell state multivectors for \mathcal{Q}_2 $\{B_0 = -S_{00} + S_{11}, B_1 = S_{01} + S_{10}, B_2 = S_{00} - S_{11}, B_3 = -S_{01} - S_{10}\}$ can be defined using the *recursive operator* $B_{(i+1) \bmod 4} = B_i (S_A + S_B)$.

The expanded form of these compact equations has been verified using the *ga.pl* tool.

$$\begin{aligned}
\mathbf{B}_0 &= A_0 B_0 (S_A + S_B) = \Phi^+ \quad \text{and also } \mathbf{B}_0 = \mathbf{B}_3 (S_A + S_B) \\
\mathbf{B}_1 &= A_0 B_0 (S_A + S_B) (S_A + S_B) = \mathbf{B}_0 (S_A + S_B) = \Psi^+ \\
\mathbf{B}_2 &= \mathbf{B}_1 (S_A + S_B) = -\Phi^+ = \Phi^- \\
\mathbf{B}_3 &= \mathbf{B}_2 (S_A + S_B) = -\Psi^+ = \Psi^- \quad (\text{then cycles back to } \mathbf{B}_0)
\end{aligned} \tag{6.7}$$

The concurrent Hadamard transform converts the standard basis for $A_0 B_0$ into the Bell states of Φ^+ . Repeated applications of this Bell operator in \mathcal{Q}_2 act similarly to the individual

Hadamard operator for \mathcal{Q}_I because an alternate phase state Ψ^+ is first achieved, and thence complement states $-\Phi^+ = \Phi^-$ and $-\Psi^+ = \Psi^-$, and lastly back to the starting state Φ^+ .

The magic state multivectors $\{\mathbf{M}_0 = \mathbf{S}_{0I} - \mathbf{S}_{10}, \mathbf{M}_1 = -\mathbf{S}_{00} - \mathbf{S}_{11}, \mathbf{M}_2 = -\mathbf{S}_{0I} + \mathbf{S}_{10}, \mathbf{M}_3 = \mathbf{S}_{00} + \mathbf{S}_{11}\}$ can also be generated using the recursive operator $\mathbf{M}_{(i+1) \bmod 4} = \mathbf{M}_i (\mathbf{S}_A - \mathbf{S}_B)$ starting from the initial state $\mathbf{M}_0 = A_0 B_0 (\mathbf{S}_A - \mathbf{S}_B)$ but also from $\mathbf{M}_0 = \mathbf{M}_3 (\mathbf{S}_A - \mathbf{S}_B)$.

Mathematically, the operators $\mathbf{B} = (\mathbf{S}_A + \mathbf{S}_B)$ and $\mathbf{M} = (\mathbf{S}_A - \mathbf{S}_B)$ are both a basis and phase converter, plus a generator for four states (two states and their complements).

In order for an operator squared to produce the complement, it must represent the same property as $\mathbf{S}_A^2 = -1$ or $\mathbf{S}_A = \sqrt{-1} = \sqrt{NOT}$. The concurrent Hadamard equivalent can be understood by computing its square, which is $\mathbf{B}^2 = (\mathbf{S}_A + \mathbf{S}_B)^2 = (1 - \mathbf{S}_A \mathbf{S}_B) = (1 - \mathbf{I}_2)$. This relationship is illustrated in Figure 6.7 and represents the table vector output notation of $(1 - \mathbf{I}_2) = [0- -0 -00- -00- 0- -0]$.

```

ga.pl table "(a0 a1 + b0 b1)(a0 a1 + b0 b1)"
Input expression is (a0 a1 + b0 b1)(a0 a1 + b0 b1)
INPUTS: a0 a1 b0 b1 | + 1 - a0 a1 b0 b1 | OUTPUT
*****
ROW 01: - - - + | + + | 0
ROW 02: - - + - | + + | 0
*****
ROW 04: - + - - | + + | 0
ROW 07: - + + + | + + | 0
*****
ROW 08: + - - - | + + | 0
ROW 11: + - + + | + + | 0
*****
ROW 13: + + - + | + + | 0
ROW 14: + + + - | + + | 0
*****
Row counts for outputs of ZERO=8, PLUS=0, MINUS=8 for TOTAL=16 rows.

```

Figure 6.7: Square of Concurrent Hadamard $(\mathbf{S}_A + \mathbf{S}_B)^2 = (1 - \mathbf{S}_A \mathbf{S}_B)$ in \mathcal{Q}_2

This result displays the valid states for $(1 - \mathbf{S}_A \mathbf{S}_B)$ and highlights the fact that all valid output states possess the same “-” value. This operator equation represents \mathbf{I}^- , an *invariant inversion operator*, but *not* a constant scalar value. Likewise, using the vector notation, its inverted operator $(-1 + \mathbf{S}_A \mathbf{S}_B) = [0++0+00+ +00+0++0]$ generates only eight “+” output states and represents an *invariant identity operator* \mathbf{I}^+ . Both invariants contain encoded phase information due to the ir component spinors, because the opposite phase version of \mathbf{I}^\pm also exists. As previously seen, Pauli spins and all n-vectors \mathbf{X} of the multivector form $\mathbf{I}^\pm = (\pm 1 \pm \mathbf{X})$ possess table outputs of the same sign for all states because the GA sum containing a constant value of ± 1 acts like a comb filter for the matching half of the product parity states.

Any state X in \mathbf{Q}_2 experiences a bimodal phase switch between two mutually exclusive sets of states because $(\mathbf{S}_A + \mathbf{S}_B) = \text{rotate}(90^\circ)$ and $(\mathbf{S}_A + \mathbf{S}_B)^2 = \mathbf{I}^- = \text{rotate}(180^\circ)$. Each set contains the inverted output states, thereby defining a single co-exclusion. The eight valid state rows in Φ^+ have different row numbers from the eight valid row states of Ψ^+ , so the bimodal operator switches between these two phase sets while using only eight non-zero operator states.

The proof for these two phase equations for any \mathbf{Q}_q is based on the concept that a recursive operator can evolve either (1) the system state from an initial starting state, or (2) itself since $(\mathbf{S}_A + \mathbf{S}_B)^3 = -(\mathbf{S}_A + \mathbf{S}_B)$ and $(\mathbf{S}_A + \mathbf{S}_B)^5 = (\mathbf{S}_A + \mathbf{S}_B)$. This invariant $(\mathbf{S}_A + \mathbf{S}_B)^5 = (\mathbf{S}_A + \mathbf{S}_B)$ means the expression $(\mathbf{S}_A + \mathbf{S}_B)^4 = (-1 + \mathbf{S}_A \mathbf{S}_B) = \mathbf{I}^+$ is a non-constant *invariant unitary* idempotent operator. Likewise, $(\mathbf{S}_A + \mathbf{S}_B)^2 = (1 - \mathbf{S}_A \mathbf{S}_B) = \mathbf{I}^-$ so \mathbf{I}^- is its *own invariant*

multiplicative inverse operator. In other words, the invariant \mathbf{I}^- squared produces the unitary operator $\mathbf{I}^- \mathbf{I}^- = (1 - \mathbf{S}_A \mathbf{S}_B)(1 - \mathbf{S}_A \mathbf{S}_B) = (-1 + \mathbf{S}_A \mathbf{S}_B) = \mathbf{I}^+$, but does not produce a constant +1 value. These relationships are most likely true due to the fact the four-vector $\mathbf{S}_A \mathbf{S}_B$ is the only n-vector in \mathbf{Q}_2 that is its own reverse $(\mathbf{S}_A \mathbf{S}_B)^\dagger = (\mathbf{S}_A \mathbf{S}_B)$, so it is self-adjoint.

Likewise for larger \mathbf{Q}_q , the equations $(\mathbf{S}_A + \mathbf{S}_B + \mathbf{S}_C + \dots)^3 = -(\mathbf{S}_A + \mathbf{S}_B + \mathbf{S}_C + \dots)$ and $(\mathbf{S}_A + \mathbf{S}_B + \mathbf{S}_C + \dots)^5 = (\mathbf{S}_A + \mathbf{S}_B + \mathbf{S}_C + \dots)$ have been easily validated for $q \in \{1 \dots 10\}$. This validation reveals that, due to the tensor product, the intermediate phase states are simply all the combinations of the pseudoscalars taken two at a time (all 4-vectors). This insight constitutes a proof of $X(\mathbf{S}_A + \mathbf{S}_B + \mathbf{S}_C + \dots)^2 = X \mathbf{I}^- = -X$ for any \mathbf{Q}_q independent of the starting state of multivector X because the second application of the even grade commuting spinor to any n-vector in X cancels the first spinor while leaving a minus sign $\mathbf{S}_J \mathbf{S}_J = -1$. Therefore for any X in \mathbf{Q}_q , the concurrent Hadamard squared produces the complement $-X$.

Conversely, the *pseudoscalar product* does not represent a Bell operator because

$(\mathbf{S}_A \mathbf{S}_B \mathbf{S}_C \dots)^3 = -(\mathbf{S}_A \mathbf{S}_B \mathbf{S}_C \dots)$ and $(\mathbf{S}_A \mathbf{S}_B \mathbf{S}_C \dots)^5 = (\mathbf{S}_A \mathbf{S}_B \mathbf{S}_C \dots)$ are only true for \mathbf{Q}_q for *some* grades $q \in \{1, 3, 4, 7, 8, \dots\}$. This concludes the proof that the concurrent Hadamard transform is a recursive operator for any \mathbf{Q}_q , and can also be used as the generator for Bell states.

Table 6.1 summarizes the definitions of the Bell and magic basis states along with the other bases discussed. The notation is the scalar coefficients from $(\mathbf{a}\mathbf{S}_{00} + \mathbf{b}\mathbf{S}_{01} + \mathbf{m}\mathbf{S}_{10} + \mathbf{u}\mathbf{S}_{11})$,

written as a vector $[\mathbf{a} \ \mathbf{b} \ \mathbf{m} \ \mathbf{u}]$. Table 6.1 is organized slightly differently from the usual tables for \mathbf{H}_4 because the column numbers, represented as a 2-bit Grey code, reflect the phase and inversion distinctions. The vector notation using spinors applies only to this table.

Table 6.1: Summary of Basis States using spinor singlets $\{\mathbf{S}_{00}, \mathbf{S}_{01}, \mathbf{S}_{10}, \mathbf{S}_{11}\}$ for \mathbf{Q}_2

Basis	Basis State 00 ₂	Basis State 01 ₂	Basis 11 ₂ = -00 ₂	Basis 10 ₂ = -01 ₂
Standard	$A_0 B_0 = [+ - - +]$	$A_0 B_1 = [- + + -]$	$A_1 B_1 = A_0 B_0$	$A_1 B_0 = A_0 B_1$
Dual	$A_- B_- = [+ + + +]$	$A_- B_+ = [- - - -]$	$A_+ B_+ = A_- B_-$	$A_+ B_- = A_- B_+$
Pauli $\mathbf{P} = \mathbf{P}_A \mathbf{P}_B$	$A_0 B_0 \mathbf{P} = [0 \ 0 \ 0 \ +]$	$A_0 B_1 \mathbf{P} = [0 \ 0 \ 0 \ -]$	$A_1 B_1 \mathbf{P} = A_0 B_0 \mathbf{P}$	$A_1 B_0 \mathbf{P} = A_0 B_1 \mathbf{P}$
	$A_0 B_- \mathbf{P} = [0 \ 0 \ + \ 0]$	$A_0 B_+ \mathbf{P} = [0 \ 0 \ - \ 0]$	$A_1 B_+ \mathbf{P} = A_0 B_- \mathbf{P}$	$A_1 B_- \mathbf{P} = A_0 B_+ \mathbf{P}$
	$A_- B_0 \mathbf{P} = [0 \ + \ 0 \ 0]$	$A_- B_1 \mathbf{P} = [0 \ - \ 0 \ 0]$	$A_+ B_1 \mathbf{P} = A_- B_0 \mathbf{P}$	$A_+ B_0 \mathbf{P} = A_- B_1 \mathbf{P}$
	$A_- B_- \mathbf{P} = [+ \ 0 \ 0 \ 0]$	$A_- B_+ \mathbf{P} = [- \ 0 \ 0 \ 0]$	$A_+ B_+ \mathbf{P} = A_- B_- \mathbf{P}$	$A_+ B_- \mathbf{P} = A_- B_+ \mathbf{P}$
Bell	$\mathbf{B}_0 = [- \ 0 \ 0 \ +]$	$\mathbf{B}_1 = [0 \ + \ + \ 0]$	$\mathbf{B}_2 = [+ \ 0 \ 0 \ -]$	$\mathbf{B}_3 = [0 \ - \ - \ 0]$
Magic	$\mathbf{M}_0 = [0 \ + \ - \ 0]$	$\mathbf{M}_1 = [- \ 0 \ 0 \ -]$	$\mathbf{M}_2 = [0 \ - \ + \ 0]$	$\mathbf{M}_3 = [+ \ 0 \ 0 \ +]$
\mathbf{H}_4 basis	$ 00\rangle$ or Φ^+	$ 01\rangle$ or Ψ^+	$ 11\rangle$ or Φ^-	$ 10\rangle$ or Ψ^-

Moreover, the standard and dual bases are co-occurrences of *all singlets* and produce indistinguishable basis pairs because an inversion of either qubit produces the same overall result. The inseparable Bell and magic states each encode a distinct co-exclusion. The Bell and magic states are topologically similar but do not overlap because they differ by a phase angle, which is proven in the next section. The \mathbf{Q}_2 Pauli states have isolated singlet terms similar to the \mathbf{H}_4 standard basis. The computational basis cannot be expressed here because, e.g. $A_0 B_0 (1 + \mathbf{a}_0)(1 + \mathbf{a}_1)(1 + \mathbf{b}_0)(1 + \mathbf{b}_1) = \mathbf{S}_{00} - \mathbf{S}_A \mathbf{b}_0 + \mathbf{a}_0 \mathbf{S}_B - \mathbf{S}_A \mathbf{S}_B$.

The application of the Bell/magic operators loses phase information upon entering the Bell/magic states, making the Bell/magic operators irreversible. This information loss strands the \mathcal{Q}_2 system state in the entangled states. This fundamental definition of the Bell states is related to phase loss and will be quantitatively explored in the Section 6.7.

The Bell state operator $\mathbf{B} = (\mathbf{S}_A + \mathbf{S}_B)$ gives an ordered cycle $\mathbf{B}_0 \rightarrow \mathbf{B}_1 \rightarrow \mathbf{B}_2 \rightarrow \mathbf{B}_3 \rightarrow \mathbf{B}_0$. Another recursive operator $\mathbf{P}_A \mathbf{P}_B = (-1 + \mathbf{S}_A)(-1 + \mathbf{S}_B)$ generates the reverse cyclic order $\mathbf{B}_3 \rightarrow \mathbf{B}_2 \rightarrow \mathbf{B}_1 \rightarrow \mathbf{B}_0 \rightarrow \mathbf{B}_3$ because the product $\mathbf{P}_A \mathbf{P}_B \mathbf{B} = (-1 + \mathbf{S}_A \mathbf{S}_B)$ is the previously discovered invariant identity operator \mathbf{I}^+ . So the *ring direction is reversible* but the phase loss is not. Also $-\mathbf{P}_A \mathbf{P}_B$ produces the normal cyclic order, so is *similar to \mathbf{B}* (denoted as \sim).

Table 6.2: Summary of Bell/magic States times Recursive Operators for \mathcal{Q}_2

Definition of the Bell and magic States		$-\mathbf{S}_{00}$ $+\mathbf{S}_{11}$	$+\mathbf{S}_{01}$ $+\mathbf{S}_{10}$	$+\mathbf{S}_{00}$ $-\mathbf{S}_{11}$	$-\mathbf{S}_{01}$ $-\mathbf{S}_{10}$	$+\mathbf{S}_{01}$ $-\mathbf{S}_{10}$	$-\mathbf{S}_{00}$ $-\mathbf{S}_{11}$	$-\mathbf{S}_{01}$ $+\mathbf{S}_{10}$	$+\mathbf{S}_{00}$ $+\mathbf{S}_{11}$
Recursive operator	$A_0 B_0$	\mathbf{B}_0	\mathbf{B}_1	\mathbf{B}_2	\mathbf{B}_3	\mathbf{M}_0	\mathbf{M}_1	\mathbf{M}_2	\mathbf{M}_3
$\mathbf{B} = (\mathbf{S}_A + \mathbf{S}_B)$	\mathbf{B}_0	$\mathbf{B}_1 \rightarrow$	$\mathbf{B}_2 \rightarrow$	$\mathbf{B}_3 \rightarrow$	$\mathbf{B}_0 \rightarrow$	0	0	0	0
$-\mathbf{B} = (-\mathbf{S}_A - \mathbf{S}_B)$	\mathbf{B}_3	$\leftarrow \mathbf{B}_3$	$\leftarrow \mathbf{B}_0$	$\leftarrow \mathbf{B}_1$	$\leftarrow \mathbf{B}_2$	0	0	0	0
$\mathbf{M} = (+\mathbf{S}_A - \mathbf{S}_B)$	\mathbf{M}_0	0	0	0	0	$\mathbf{M}_1 \rightarrow$	$\mathbf{M}_2 \rightarrow$	$\mathbf{M}_3 \rightarrow$	$\mathbf{M}_0 \rightarrow$
$-\mathbf{M} = (-\mathbf{S}_A + \mathbf{S}_B)$	\mathbf{M}_3	0	0	0	0	$\leftarrow \mathbf{M}_3$	$\leftarrow \mathbf{M}_0$	$\leftarrow \mathbf{M}_1$	$\leftarrow \mathbf{M}_2$
$-\mathbf{P}_A \mathbf{P}_B \sim \mathbf{B}$	$-\mathbf{S}_{11}$	$\mathbf{B}_1 \rightarrow$	$\mathbf{B}_2 \rightarrow$	$\mathbf{B}_3 \rightarrow$	$\mathbf{B}_0 \rightarrow$	$=\mathbf{M}_0$	$=\mathbf{M}_1$	$=\mathbf{M}_2$	$=\mathbf{M}_3$
$\mathbf{P}_A \mathbf{P}_B \sim -\mathbf{B}$	\mathbf{S}_{11}	$\leftarrow \mathbf{B}_3$	$\leftarrow \mathbf{B}_0$	$\leftarrow \mathbf{B}_1$	$\leftarrow \mathbf{B}_2$	$\mathbf{M}_{2 \times 0}$	$\mathbf{M}_{3 \times 1}$	$\mathbf{M}_{0 \times 2}$	$\mathbf{M}_{1 \times 3}$
$(-1 - \mathbf{S}_A) \mathbf{P}_B \sim \mathbf{M}$	$-\mathbf{S}_{01}$	$\mathbf{B}_{2 \times 0}$	$\mathbf{B}_{3 \times 1}$	$\mathbf{B}_{0 \times 2}$	$\mathbf{B}_{1 \times 3}$	$\mathbf{M}_1 \rightarrow$	$\mathbf{M}_2 \rightarrow$	$\mathbf{M}_3 \rightarrow$	$\mathbf{M}_0 \rightarrow$
$\mathbf{P}_A (-1 - \mathbf{S}_B) \sim -\mathbf{M}$	$-\mathbf{S}_{10}$	$\mathbf{B}_{2 \times 0}$	$\mathbf{B}_{3 \times 1}$	$\mathbf{B}_{0 \times 2}$	$\mathbf{B}_{1 \times 3}$	$\leftarrow \mathbf{M}_3$	$\leftarrow \mathbf{M}_0$	$\leftarrow \mathbf{M}_1$	$\leftarrow \mathbf{M}_2$
$(\mathbf{P}_A)^{-1} (\mathbf{P}_B)^{-1} \sim \mathbf{B}$	\mathbf{S}_{00}	$\mathbf{B}_1 \rightarrow$	$\mathbf{B}_2 \rightarrow$	$\mathbf{B}_3 \rightarrow$	$\mathbf{B}_0 \rightarrow$	$\mathbf{M}_{2 \times 0}$	$\mathbf{M}_{3 \times 1}$	$\mathbf{M}_{0 \times 2}$	$\mathbf{M}_{1 \times 3}$
$-(\mathbf{P}_A)^{-1} (\mathbf{P}_B)^{-1} \sim -\mathbf{B}$	$-\mathbf{S}_{00}$	$\leftarrow \mathbf{B}_3$	$\leftarrow \mathbf{B}_0$	$\leftarrow \mathbf{B}_1$	$\leftarrow \mathbf{B}_2$	$=\mathbf{M}_0$	$=\mathbf{M}_1$	$=\mathbf{M}_2$	$=\mathbf{M}_3$

The magic state order $\mathbf{M}_3 \rightarrow \mathbf{M}_2 \rightarrow \mathbf{M}_1 \rightarrow \mathbf{M}_0 \rightarrow \mathbf{M}_3$ also has a reverse order recursive operator $\mathbf{P}_A (-1 \oplus \mathbf{S}_B)$ whose product with the magic basis operator $\mathbf{M} = (\mathbf{S}_A - \mathbf{S}_B)$ forms the other idempotent identity operator $\mathbf{I}^+ = (-1 + \mathbf{S}_A)(-1 \oplus \mathbf{S}_B)(\mathbf{S}_A - \mathbf{S}_B) = (-1 \oplus \mathbf{S}_A \mathbf{S}_B)$, but the rows are out of phase when compared to $(-1 \oplus \mathbf{S}_A \mathbf{S}_B)$. These “reverse” operators do not force entry into the Bell or magic states. A summary of the Bell and magic operators is provided in Table 6.2.

It is not clear from the literature how any of these recursive operators can be expressed using the “ket” notation in \mathbf{H}_d , nor is it immediately obvious that half of these states are in fact complements of the others. Incidentally, taking an operator \mathbf{F} to the n^{th} power in \mathbf{H}_n is the same as the n^{th} tensor product, which defines the *tensor power* operator, denoted as $\mathbf{F}^{\otimes n}$. Of course, these specialized tensor product and tensor power operators are not needed in \mathbf{Q}_q .

6.7 Entanglement Means Co-occurrence with Cross-Qubit Spinor

The Bell basis $\mathbf{B}_0 = -\mathbf{S}_{00} + \mathbf{S}_{11}$ and magic basis $\mathbf{M}_3 = \mathbf{S}_{00} + \mathbf{S}_{11}$ in \mathbf{Q}_2 are identical except for a sign change, which suggests a phase difference. The exact form of this phase change can be derived starting with $\mathbf{M}_3 = (+\mathbf{a}_0 \mathbf{b}_0 + \mathbf{a}_1 \mathbf{b}_1)$. The approach for this proof uses the fact that for any multivector A using mod 3 addition, $A + A = -A$, or conversely $-A - A = +A$:

$$\begin{aligned}
 \mathbf{M}_3 &= (\mathbf{a}_0 \mathbf{b}_0 + \mathbf{a}_1 \mathbf{b}_1) \\
 \rightarrow &(-\mathbf{a}_0 \mathbf{b}_0 + \mathbf{a}_1 \mathbf{b}_1 - \mathbf{a}_0 \mathbf{b}_0) \\
 \rightarrow &(-\mathbf{a}_0 \mathbf{b}_0 + \mathbf{a}_1 \mathbf{b}_1) - (\mathbf{a}_0 \mathbf{b}_0) \\
 \rightarrow &(\mathbf{a}_0 - \mathbf{a}_1)(\mathbf{b}_0 - \mathbf{b}_1)(\mathbf{a}_0 \mathbf{a}_1 + \mathbf{b}_0 \mathbf{b}_1) - (\mathbf{a}_0 \mathbf{b}_0) \\
 \rightarrow &A_0 B_0 (\mathbf{S}_A + \mathbf{S}_B) - \mathbf{S}_{00} = \mathbf{B}_0 - \mathbf{S}_{00} \quad \text{but also } \mathbf{M}_3 = \mathbf{B}_2 (\mathbf{S}_{01} + \mathbf{S}_{10}) = -\mathbf{B}_0 (\mathbf{S}_{01} + \mathbf{S}_{10})
 \end{aligned} \tag{6.8}$$

This result reveals a *co-occurrence* within \mathbf{M}_i , in which a cross-qubit spinor $\mathbf{S}_{ij} = \mathbf{E}_{ij}$ is equivalent to a phase change (cf. Section 6.6). This derivation is confirmed in Figure 6.8 using the *ga.pl* tool, where the highlighted rows show the valid states with multiple superposition phases. State transitions can therefore be expressed either as *concurrent phase changes* or as a multiplicative operator.

```

Input expression is (a0 - a1)(b0 - b1)(a0 a1 + b0 b1) - a0 b0
INPUTS: a0 a1 b0 b1 | + a0 b0 + a1 b1 | OUTPUT
*****
ROW 00: -- -- | ++ | - → A in superposition, B in superposition
ROW 03: -- ++ | -- | + → A in superposition, B in superposition
*****
ROW 05: - + - + | ++ | - → A is classical, B is classical
ROW 06: - + + - | -- | + → A is classical, B is classical
*****
ROW 09: + - - + | -- | + → A is classical, B is classical
ROW 10: + - + - | ++ | - → A is classical, B is classical
*****
ROW 12: ++ -- | -- | + → A in superposition, B in superposition
ROW 15: ++ ++ | ++ | - → A in superposition, B in superposition
*****
Row counts for outputs of ZERO=8, PLUS=4, MINUS=4 for TOTAL=16 rows.

```

Figure 6.8: Validation of Magic State Phase Relationship $\mathbf{M}_3 = \mathbf{B}_0 - \mathbf{S}_{00}$ in \mathbf{Q}_2

This co-occurrence with a phase spinor is an important concept and will be used in section 6.8 to prove that the Bell and magic states represent an irretrievable loss of phase information caused by their application. This loss of information due to applying \mathbf{B} and \mathbf{M} is important because it formally shows why the Bell/magic states are entangled and inseparable.

6.8 Bell Basis States are Irreversible in \mathbf{Q}_2

Any basis (except the computational) can be entered and *reversibly exited* in \mathbf{Q}_1 because, for every multiplicative basis operator, there exists a multiplicative inverse, thus reversing the process and *exiting* the state back to the starting state. All n-vectors are their own

multiplicative inverse (with appropriate sign change). Unfortunately, a multiplicative inverse does not exist for all multivectors (sums of arbitrary rank n -vectors). This lack of operator reversibility strongly suggests that some loss of information must therefore impact the system state. This section proves that entering any of the Bell or magic states causes a loss of phase information, and details exactly how these irreversible recursive operators irretrievably throw away phase information.

This claim will be substantiated by showing exactly what phase information is discarded by the recursive operators and then by demonstrating that the recursive operators are irreversible because no multiplicative inverse exists. Finally, the reason the operators discard information will be directly related to the action of the multiplicative operator in canceling some states.

6.8.1 Discarded Phase Information in Bell States

While searching for the reversibility of the Bell operator, the missing phase information was discovered. Start with qubits A and B in their superposition states:

$$A_0 B_0 (\mathbf{S}_A \mathbf{S}_B) = A_+ B_+ = (\mathbf{S}_{00} + \mathbf{S}_{01} + \mathbf{S}_{10} + \mathbf{S}_{11}) \quad (6.9)$$

Next apply the concurrent Hadamard or Bell operator to produce the Bell state:

$$\mathbf{B}_2 = A_+ B_+ (\mathbf{S}_A + \mathbf{S}_B) = (\mathbf{S}_{00} - \mathbf{S}_{11}) \quad (6.10)$$

So the ideal reversible solution would be the inverse operator $(\mathbf{S}_A + \mathbf{S}_B)^{-1}$ such that:

$$A_+ B_+ =? \mathbf{B}_2 (\mathbf{S}_A + \mathbf{S}_B)^{-1} \quad (6.11)$$

Using trial and error, the closest value for $(\mathbf{S}_A + \mathbf{S}_B)^{-1}$ is either $(-1 + \mathbf{S}_A)$ or $(-1 + \mathbf{S}_B)$:

$$\mathbf{B}_2 (-1 + \mathbf{S}_A) = \mathbf{B}_2 (-1 + \mathbf{S}_B) = (-\mathbf{S}_{00} + \mathbf{S}_{01} + \mathbf{S}_{10} + \mathbf{S}_{11}) = (A_+ B_+) - \mathbf{S}_{00} \quad (6.12)$$

The result is exactly a phase difference away from a desired return state. This exact multiplicative solution exists only if $(\mathbf{S}_A + \mathbf{S}_B)^{-1}$ exists. It is suspected that $1/(\mathbf{S}_A + \mathbf{S}_B)$ does not exist but an exhaustive proof confirms it in the next subsection. In addition, the known invariant inversion operator $(1 - \mathbf{S}_A \mathbf{S}_B)$ reverses the state iteration direction without exiting.

6.8.2 No Multiplicative Inverse for $(\mathbf{S}_A + \mathbf{S}_B)$

The Bell states would be reversibly exited if a solution existed for $X = (\mathbf{S}_A + \mathbf{S}_B)^{-1}$ such that $(\mathbf{S}_A + \mathbf{S}_B)(X) = 1$. The *gasolve.pl* tool exhaustively tested all 43,046,720 possibilities in a five-day run for solutions to $(\mathbf{a0} \mathbf{a1} + \mathbf{b0} \mathbf{b1})(X) = 1$ but found none, so $(\mathbf{S}_A + \mathbf{S}_B)^{-1}$ does not exist in \mathcal{Q}_2 . This is also proved again below using the Cancellation Principle of Multiplication.

6.8.3 Recursive Operator Erases Phase Information

The recursive operator erases information because some operator states *multiplicatively* cancel to 0 (nilpotent). This is shown using the following equations and Pauli substitutions:

$$\begin{aligned}
 A_+ B_+ &= A_0 B_0 \mathbf{S}_A \mathbf{S}_B = +\mathbf{S}_{00} + \mathbf{S}_{10} + \mathbf{S}_{01} + \mathbf{S}_{11} = (\mathbf{S}_{00} + \mathbf{S}_{11}) + (\mathbf{S}_{10} + \mathbf{S}_{01}) = \mathbf{M}_3 + \mathbf{B}_I \\
 \mathbf{S}_{11} &= A_0 B_0 (\mathbf{P}_A)(-1)(\mathbf{P}_B)(-1) = A_0 B_0 \mathbf{P}_A \mathbf{P}_B (+1) \\
 \mathbf{S}_{00} &= A_0 B_0 (\mathbf{P}_A \mathbf{S}_A)(\mathbf{P}_B \mathbf{S}_B) = A_0 B_0 \mathbf{P}_A \mathbf{P}_B \mathbf{S}_A \mathbf{S}_B \\
 \mathbf{S}_{01} &= A_0 B_0 (\mathbf{P}_A \mathbf{S}_A)(\mathbf{P}_B (-1)) = -A_0 B_0 \mathbf{P}_A \mathbf{P}_B \mathbf{S}_A \\
 \mathbf{S}_{10} &= A_0 B_0 (\mathbf{P}_A (-1))(\mathbf{P}_B \mathbf{S}_B) = -A_0 B_0 \mathbf{P}_A \mathbf{P}_B \mathbf{S}_B
 \end{aligned} \tag{6.13}$$

Now substitute, combine pairs, and remove the common highlighted $A_0 B_0$ to produce:

$$\begin{aligned}
 (A_0 B_0) \mathbf{S}_A \mathbf{S}_B &= \mathbf{M}_3 + \mathbf{B}_I = (A_0 B_0) \mathbf{P}_A \mathbf{P}_B (1 + \mathbf{S}_A \mathbf{S}_B) - (A_0 B_0) \mathbf{P}_A \mathbf{P}_B (\mathbf{S}_A + \mathbf{S}_B) \\
 \mathbf{S}_A \mathbf{S}_B &= \mathbf{P}_A \mathbf{P}_B (1 + \mathbf{S}_A \mathbf{S}_B) - \mathbf{P}_A \mathbf{P}_B (\mathbf{S}_A + \mathbf{S}_B)
 \end{aligned} \tag{6.14}$$

Use the *ga.pl* tool to simplify $\mathbf{P}_A \mathbf{P}_B (1 + \mathbf{S}_A \mathbf{S}_B) = (-1 - \mathbf{S}_A \mathbf{S}_B)$ and combine terms:

$$\begin{aligned}
 \mathbf{S}_A \mathbf{S}_B &= (-1 - \mathbf{S}_A \mathbf{S}_B) - \mathbf{P}_A \mathbf{P}_B (\mathbf{S}_A + \mathbf{S}_B) \\
 (1 - \mathbf{S}_A \mathbf{S}_B) &= -\mathbf{P}_A \mathbf{P}_B (\mathbf{S}_A + \mathbf{S}_B)
 \end{aligned} \tag{6.15}$$

Earlier we found the concurrent Hadamard operator squared is $(1 - \mathbf{S}_A \mathbf{S}_B) = (\mathbf{S}_A + \mathbf{S}_B)^2$, so:

$$(1 - \mathbf{S}_A \mathbf{S}_B) = (\mathbf{S}_A + \mathbf{S}_B) (\mathbf{S}_A + \mathbf{S}_B) = -\mathbf{P}_A \mathbf{P}_B (\mathbf{S}_A + \mathbf{S}_B) \quad (6.16)$$

Now apply the Cancellation Principle of Multiplication for multivectors (p 38 in [16]), which states: if $YX = ZX$ then $Y = Z$ if and only if $1/X$ exists. So for $X = Y = \mathbf{B}$ and $Z = -\mathbf{P}_A \mathbf{P}_B$:

$$Z = -\mathbf{P}_A \mathbf{P}_B = (-1 + \mathbf{S}_A + \mathbf{S}_B - \mathbf{S}_A \mathbf{S}_B) = Y + (-1 - \mathbf{S}_A \mathbf{S}_B) \quad (6.17)$$

So $Y = Z$ only if $(-1 - \mathbf{S}_A \mathbf{S}_B) = 0$. Since $(-1 - \mathbf{S}_A \mathbf{S}_B)$ is always non-zero, this contradiction means that $(\mathbf{S}_A + \mathbf{S}_B)^{-1}$ does not exist. The overall equality is true because $(\mathbf{S}_A + \mathbf{S}_B)(-1 - \mathbf{S}_A \mathbf{S}_B) = 0$. So even though the equality $\mathbf{B} \mathbf{B} = -\mathbf{P}_A \mathbf{P}_B \mathbf{B}$ is true, the imputed equality $\mathbf{B} = -\mathbf{P}_A \mathbf{P}_B$ is *never true*. This means two $(-1 - \mathbf{S}_A \mathbf{S}_B)$ of the four terms from the multivector $-\mathbf{P}_A \mathbf{P}_B$ “do not occur” since they are multiplicatively masked by the operator $\mathbf{B} = (\mathbf{S}_A + \mathbf{S}_B)$. Also the inverse of the expression $-(\mathbf{P}_A \mathbf{P}_B)^{-1} = -(1 + \mathbf{S}_A)(1 + \mathbf{S}_B)$ exists and does not exit the Bell states either. Table 6.3 shows the table output for the three important states.

Table 6.3: Masked Operator States $(-1 - \mathbf{S}_A \mathbf{S}_B)(\mathbf{S}_A + \mathbf{S}_B) = 0$ for \mathcal{Q}_2

Complete Operator $-\mathbf{P}_A \mathbf{P}_B$	Masking Operator $(\mathbf{S}_A + \mathbf{S}_B)$	Filtered States $(-1 - \mathbf{S}_A \mathbf{S}_B)$
$a_0 a_1 b_0 b_1 -1 + a_0 a_1 +$ $+ b_0 b_1 - a_0 a_1 b_0 b_1 \text{OUT}$ ***** ROW 00 - - - - - + + - 0 ROW 03 - - + + - + + - 0 ***** ROW 05 - + - + - - - - - ROW 06 - + + - - - - - - ***** ROW 09 + - - + - - - - - ROW 10 + - + - - - - - - ***** ROW 12 - - - - - + + - 0 ROW 15 - - + + - + + - 0 *****	$a_0 a_1 b_0 b_1 $ $a_0 a_1 + b_0 b_1 \text{OUT}$ ***** ROW 00: - - - - + + - ROW 03: - - + + + + - ***** ROW 05: - + - + - - + ROW 06: - + + - - - + ***** ROW 09: + - - + - - + ROW 10: + - + - - - + ***** ROW 12: + + - - + + - ROW 15: + + + + + + - *****	$a_0 a_1 b_0 b_1 $ $-1 - a_0 a_1 b_0 b_1 \text{OUT}$ ***** ROW 00: - - - - - - + ROW 03: - - + + - - + ***** ROW 05: - + - + - - + ROW 06: - + + - - - + ***** ROW 09: + - - + - - + ROW 10: + - + - - - + ***** ROW 12: + + - - - - + ROW 15: + + + + - - + *****
$-1 + a_0 a_1 + b_0 b_1 - a_0 a_1 b_0 b_1 = (a_0 a_1 + b_0 b_1) + (-1 - a_0 a_1 b_0 b_1)$		

Since $(-1 - \mathbf{S}_A \mathbf{S}_B)$ is a factor of \mathbf{M}_i , therefore $\mathbf{M}_i \mathbf{B} = 0$. Similarly the multivector $(1 - \mathbf{S}_A \mathbf{S}_B)$ is a factor of \mathbf{B}_i , therefore $\mathbf{B}_i \mathbf{M} = 0$, which means half the states in equation 6.13 are erased using either the \mathbf{B} or \mathbf{M} operators. Also, if the system is in a Bell state, multiplying by the magic operator $\mathbf{B}_{0-3} (\mathbf{S}_A - \mathbf{S}_B) = 0$, or vice versa $\mathbf{M}_{0-3} (\mathbf{S}_A + \mathbf{S}_B) = 0$, always produces zero.

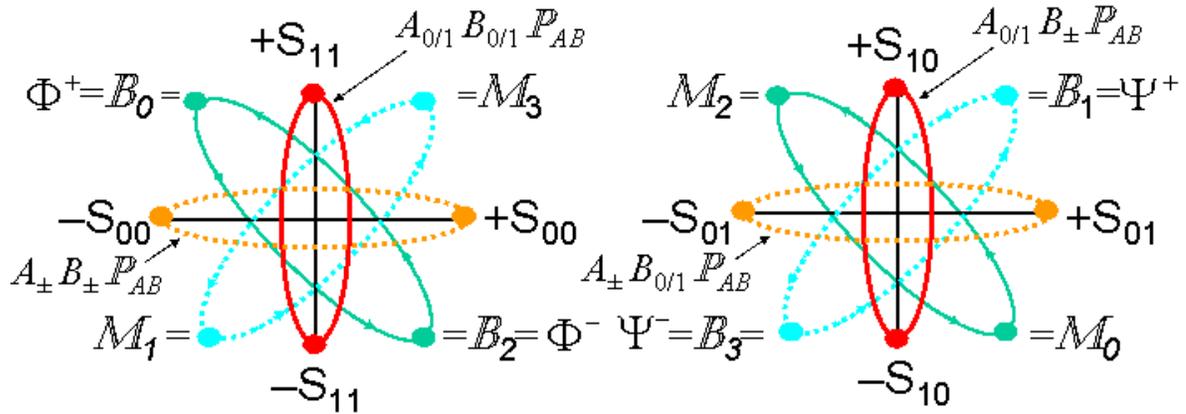


Figure 6.9 Summary of Bell and magic states using Pauli basis singlets

In summary, many significant results have been uncovered for the alternate bases of \mathbf{Q}_2 . The interpretation of co-occurrence and co-exclusion has added great insight to this process, and the natural spinor representation in geometric algebra has allowed simple proofs of previously unexplored topics. The Bell and magic bases are fundamentally different from the standard and dual bases because more distinct states are maintained in their concurrent relationships, *due to a loss of phase information to enter those states*. Therefore, the Bell and magic states are entangled and not separable because no multiplicative inverse operator exists to exit those states. Mathematically speaking, this is equivalent to the \mathbf{H}_4 tensor product

definition of separability but gives more insight into the irreversible nature of the Bell and magic states. The recursive ***B*** and ***M*** operators exclude some operator states, thus irreversibly losing phase information compared to the unentangled standard and dual bases.

CHAPTER 7

QUANTUM COMPUTING IN GEOMETRIC ALGEBRA

7.1 Single Qubit Operators

The primary operators or *gates* that exist for a single qubit are the Hadamard, Inverter, and Phase gates, plus their additive combinations that define the Pauli operators. These gates have previously been defined but are summarized again in Table 7.1. In addition, basis change operators (and their multiplicative combinations) exist, including the measurement and computational operators. For qubit A with $\mathcal{Q}_1 = \text{span}\{\mathbf{a0}, \mathbf{a1}\}$, the co-occurrence is $A = (\pm\mathbf{a0} \pm\mathbf{a1})$, spinor $\mathbf{S}_A = (\mathbf{a0} \mathbf{a1})$, spinor reversion $\tilde{\mathbf{S}}_A = (\mathbf{a1} \mathbf{a0}) = (-\mathbf{a0} \mathbf{a1})$, the multivector rotor $R = \mathbf{a} - \mathbf{b}\mathbf{S}_A$, and the rotor reversion $\tilde{R} = \mathbf{a} - \mathbf{b}\tilde{\mathbf{S}}_A = \mathbf{a} + \mathbf{b}\mathbf{S}_A$.

Table 7.1: Operator and Basis Summary for Single Qubit A in \mathcal{Q}_1

Gates	Geometric Algebra	Operator comments regarding $A = (\pm\mathbf{a0}\pm\mathbf{a1})$
Hadamard	$A (\mathbf{S}_A) = A (\mathbf{a0} \mathbf{a1})$	Spinor \mathbf{S}_A rotates 90° causing phase flip
Inverter	$\mathbf{S}_A A \tilde{\mathbf{S}}_A = A \mathbf{S}_A^2 = -A$	Two spinors rotate 180° causing spin flip
Phase Gate	$RA\tilde{R}$	Angle q then $\mathbf{a} = \cos(q/2)$, $\mathbf{b} = \sin(q/2)$
Basis Operators	Geometric Algebra	Basis comments regarding $A = (\pm\mathbf{a0}\pm\mathbf{a1})$
Pauli Basis	$A (-1 + \mathbf{S}_A)$	Rotates 45° to/from diag/ver-hor basis planes
Circular Basis	$A (\pm\mathbf{a0})$ or $A (\pm\mathbf{a1})$	Changes to/from odd/even grade basis planes
Direct Basis	$A (\pm \mathbf{a0} \pm \mathbf{a1})$	Produces ± 1 or 50/50 random value-reversible
Trine Basis	$A (1 \pm \mathbf{a?} \pm \mathbf{S}_A)$	Rotates 120° where $\mathbf{a?}$ is either $\mathbf{a0}$ or $\mathbf{a1}$
Computational	$A (1 \pm \mathbf{a0})(1 \pm \mathbf{a1})$	Produces \mathbf{I}^\pm or 50/50 random; is irreversible

The most interesting results are the operators where no multiplicative inverse exists. Those same states also have interesting properties for their squares, where some have the *nilpotent* property $X^2 = 0$; some operators are idempotent because they have no additional effect when applying them more than once $X^2 = X = \sqrt{X}$; while others are indistinguishable between inversion, addition, and multiplication: $X^2 = -X = X + X = 2X$. These properties restrict designing operators for specific purposes. Probably the most interesting are the multivector states $\{A_0, A_-\}$ and their additive inverses $\{A_+, A_+\}$ which have the property $X^2 = -1$ or $(X)(-X) = 1$, which means the standard and dual basis states have *identical additive and multiplicative inverses* or $X^{-1} = -X$, which is a useful simplification used later.

7.1.1 Operators as Computational Basis Vectors

The R_k computational bases were previously shown to be linearly independent of each other and this signifies something important from an operator perspective. Since each row decode expression R_k is encoded to be an linearly independent vector, all algebraic states and operators can be expressed as additive combinations of these primitive row decode computational states, as illustrated in Table 7.3, where gp = geometric product.

Table 7.3: Computational Basis Products as Independent States for \mathcal{Q}_1

$A_- = (1 - \mathbf{a0})(1 - \mathbf{a1}) =$ $R_0 = [+ \ 0 \ 0 \ 0]$	$A_{+-} = (1 + \mathbf{a0})(1 - \mathbf{a1}) =$ $R_2 = [0 \ 0 \ + \ 0]$	$A_- - A_{+-} = A_0$ $+ [0 \ + \ 0 \ 0]$
$A_{-+} = (1 - \mathbf{a0})(1 + \mathbf{a1}) =$ $R_1 = [0 \ + \ 0 \ 0]$	$A_{++} = (1 + \mathbf{a0})(1 + \mathbf{a1}) =$ $R_3 = [0 \ 0 \ 0 \ +]$	$- [0 \ 0 \ + \ 0]$ <hr/> $[0 \ + \ - \ 0]$
$[a \ b \ c \ d] \square [w \ x \ y \ z] = [a \square w, b \square x, c \square y, d \square z]$ for operator $\square \in \{+, gp\}$		

Under these conditions, all addition and n-vector multiplication operators can be simply implemented on individual elements of the two vectors. Inversions due to anti-commutative swaps must be dealt with separately. These vectors are equivalent to matrix diagonals (since $[++++] = +1$ and $[----] = -1$) and as well to combinations of the idempotent projection operators $(-1 \pm \mathbf{a0})$ and $(-1 \pm \mathbf{a1})$ or $(-1 \pm \mathbf{a0})(-1 \pm \mathbf{a1})$.

Using the above basis definitions, all vector operators can be simply implemented pair-wise on individual pairs of elements in both vectors. Whenever a symbolic inversion is detected due to anti-commutative reordering, the vector (highlighted) must be first inverted. Table 7.4 illustrates that this symbolic algebra is equivalent to matrix operations on diagonal elements using the computational basis with non-commutative inversions appropriately applied.

Table 7.4: Operators as Concurrent Orthogonal Vectors in \mathcal{Q}_1

Example Operators	Invert Anti-Commutative?	$[A_{--} A_{-+} A_{+-} A_{++}] = A_{--} + A_{-+} + A_{+-} + A_{++}$
+1	no	+ [++++] = +1 or identity matrix
-1	no	- [++++] = [----] = -1
+ a0	no	[-++ +]
- a0	no	- [-++ +] = [++--]
+ a1	no	[-+- +]
- a1	no	- [-+- +] = [+-- -]
$A_0 = + \mathbf{a0} - \mathbf{a1}$	no	[-++ +] + [-+- +] = [0+-0]
$A_1 = - \mathbf{a0} + \mathbf{a1}$	no	[++--] + [-+- +] = [0-+0]
$A_+ = + \mathbf{a0} + \mathbf{a1}$	no	[-++ +] + [-+- +] = [+00-]
$A_- = - \mathbf{a0} - \mathbf{a1}$	no	[++--] + [-+- +] = [-00+]
$S_A = + \mathbf{a0} \mathbf{a1}$	no	[-++ +] gp [-+- +] = [+---]
$\mathbf{a0} S_A = \mathbf{a1}$	no	[-++ +] gp [++--] = [-+- +]
$\mathbf{a1} S_A = - \mathbf{a0}$	INVERT	[-+- +] gp (-[++--]) = [++--]
$S_A S_A = - 1$	INVERT	[++--] gp (-[++--]) = [----]
$(1+\mathbf{a0})(1+\mathbf{a1}) =$ $= 1 + \mathbf{a0} + \mathbf{a1} + \mathbf{a0} \mathbf{a1}$	no	[00--] gp [0-0-] = [000+] = = +1 + [-+- +] + [-+- +] + [++--]
$(1+\mathbf{a1})(1+\mathbf{a0}) =$ $= 1 + \mathbf{a0} + \mathbf{a1} - \mathbf{a0} \mathbf{a1}$	no	(1+[-+- +])(1+[++--]) = [+----] = = +1 + [-+- +] + [-+- +] - [++--]

As demonstrated by the last example in Table 7.4, all multivector products must be distributed first to guarantee the inversion happens only to the most specific n-vector. Therefore in this vector notation, multiplication must have precedence over addition due to non-commutativity rules. Maintaining the non-commutativity order information still requires some algebraic information (or can alternatively maintain the sorting order by using the relative frequency information of input vectors), so that the matrices demonstrate the computational basis properties. This compact table output form has been used in this dissertation since Chapter Four and can be generated instead of a full table by choosing the option “*ga.pl vector <equation>*.” Vector addition works in all spaces but *gp* only in \mathcal{Q}_1 .

The final detail to point out about the computational bases is that there are twice as many output values compared to Hilbert spaces for each qubit because the computational bases are not segregated into separate sets for classical and superposition states, but they are pair wise orthogonal $R_0 \bullet R_3 = R_1 \bullet R_2 = 0$. Also, even though the computational bases are not reversible operators, appropriate sums can produce invertible and reversible solutions. These bases are simply the most primitive topological feature of the algebra and have the most precise coverage compared to either the input vector set or n-vector product singlets. Since the computational bases are the same as table row outputs, they could be used as an efficient alternative mechanism for computing the table outputs, effectively in parallel.

7.1.2 Single qubit measurements

Besides the ability to place qubits into superposition, the next most important aspect of quantum computing is to find the *answer* residing in the qubit state via a *measurement*.

Quantum measurement has traditionally been confusing and misunderstood, due in part to the

concurrent nature of superposition states, but also due to the unintuitive representation of basis states using complex numbers. Since the qubit representation \mathcal{Q}_1 in geometric algebra simplifies these concepts, the topic of measurement can be addressed more intuitively.

Measurement is confusing because the internal qubit axes are completely independent of the orientation within our normal 3D \mathbf{E}_3 space, which is especially hard to visualize for high-dimensional entangled spaces. The mapping process into the \mathbf{E}_3 space therefore represents a projection (or shadow) into the lower dimensional space. For the orthogonal \mathcal{Q}_1 vectors, the two angles $\{\mathbf{f}, \mathbf{q}\}$ for projecting state A into \mathbf{E}_3 are imposed by the major spin axis orientation \mathbf{f} and a phase gate angle \mathbf{q} , but only when the measurement is made from an apparatus located inside \mathbf{E}_3 and oriented using $\{\mathbf{f}, \mathbf{q}\}$. The result of a measurement is the *answer* (+1 or -1 each with some probability) *plus* the qubit changing to the *end state* due to the application of the singular measurement operator.

For simplicity sake, let's disallow arbitrary phase and limit the qubit state to the discrete basis previously discussed. Measurement is nothing more than asking if the qubit currently occupies one of the two states of a particular basis. For example, when the qubit occupies exactly either A_0 or A_1 and a measurement asks if the qubit "Occupies the spin-up state?", the answer will be either "yes" or "no" respectively, with 100% probability. Asking the basis question assumes that the qubit basis is known so that the orientation is aligned with the measurement apparatus orientation. This is why it is crucial to *know or maintain (by design)* the correct basis in order to make a valid projection measurement.

If the qubit occupies a state in another basis, then some random result will occur with exactly a $1/2 = 50\%$ probability. These probabilities are the ratio of the number of states of each sign in the vector notation, which are generally 0%, 50% or 100%. Measuring a spinor state $S_A = [+ - - +]$ will always produce a 50/50% random value. For Q_1 the equivalent probability amplitudes are simply the square root of the probability: $\sqrt{1/2} = \pm 1/\sqrt{2}$. Introducing some arbitrary phase angle due to phase gate operation will create other probability values.

As illustrated in Table 7.5, many state and operator products generate constants or spinors. The important difference is these examples are all *reversible* so do not constitute a measurement, even though they are useful. A measurement only occurs using a singular operator as originally shown in Table 5.4, which produces an answer, but also side-effects the qubit end state to change to the state matching the question. So for singular operator $R_k = (1 \pm \mathbf{a0})(1 \pm \mathbf{a1})$ the matching end state is $\pm \mathbf{a0} \pm \mathbf{a1}$. The only “constant” answers are due to sparse invariants I^\pm and not the constants ± 1 , which are really valid reversible states.

Table 7.5: Reversible Basis Encoding Results in Q_1

Is Qubit State A in basis Y?	Y=Standard?	Y=Dual?	Circular?	H Pauli?	V Pauli?
	A (+ a0 - a1)	A (- a0 - a1)	A (1- S_A)	A (a0)	A (a1)
$A_0 = (+ \mathbf{a0} - \mathbf{a1})$	-1	+ a0 a1	+ a1	+1+ a0 a1	-1+ a0 a1
$A_I = (- \mathbf{a0} + \mathbf{a1})$	+1	- a0 a1	- a1	-1- a0 a1	+1- a0 a1
$A_+ = (+ \mathbf{a0} + \mathbf{a1})$	+ a0 a1	+1	- a0	+1- a0 a1	+1+ a0 a1
$A_- = (- \mathbf{a0} - \mathbf{a1})$	- a0 a1	-1	+ a0	-1+ a0 a1	-1- a0 a1
$A = (+1 + \mathbf{a0 a1})$	+ a1	+ a0	-1	+ a0 - a1	+ a0 + a1
$A = (-1 - \mathbf{a0 a1})$	- a1	- a0	+1	- a0 + a1	- a0 - a1
$A = + \mathbf{a0}$	+ 1 - a0 a1	- 1 - a0 a1	+ a0 - a1	+1	+ a0 a1
$A = - \mathbf{a0}$	- 1 + a0 a1	+ 1 + a0 a1	- a0 + a1	-1	- a0 a1
$A = + \mathbf{a1}$	- 1 - a0 a1	- 1 + a0 a1	+ a0 + a1	- a0 a1	+1
$A = - \mathbf{a1}$	+ 1 + a0 a1	+ 1 - a0 a1	- a0 - a1	+ a0 a1	-1

The four computational basis operators $P_k = (-1)(1 \pm \mathbf{a0})(1 \pm \mathbf{a1})$ and the row-decode operators $R_k = (1 \pm \mathbf{a0})(1 \pm \mathbf{a1})$ are *irreversible* measurement operators. The process is irreversible because a *multiplicative inverse does not exist* for $(1 \pm \mathbf{a0})$ and $(1 \pm \mathbf{a1})$. This means our classical definition of a one-to-one mapping is insufficient to formally express reversibility and therefore not identical to the quantum definition for unitary transforms because of multiplicative cancellation. This is consistent with the conventional definition of unitary matrix operators used in Hilbert spaces.

7.2 Two-Qubit Operators

The gate and measurement operators for systems with two or more qubits are identical to the single qubit scenario, except that anticommutative rules must be applied to products of odd grade multivectors. The major similarity occurs when applying an operator to the *separable* standard and dual bases, a process that is identical to applying them to each of the individual qubits and represents two distinct co-exclusions. Another major difference is that the product of two idempotent operators from two qubits forms an expanded cyclic definition of an idempotent operator. As will be demonstrated, the geometric product of combined *separable* states masks multiple indistinguishable pairs of states, but two classical bits' worth of information can still be extracted. Conversely, the *entangled* Bell and magic basis states contain a loss of one bit's worth of phase information. As a result, these basis states no longer represent two separable co-exclusions, but rather only one entangled classical bit's worth of information. It is impossible to exit Bell/magic states without erasing the entangled state, since no multiplicative inverse exists. This argument makes it clear why the answer and the end states are distinct due to a measurement because the answer is not a valid qubit state and Bell states can be exited by forcibly resetting the coupled states.

7.2.1 Control-Not Gate for Q_2

An important gate for two qubits is the conditional inversion behavior called the *control-not* gate. The key to defining the operator for the control-not gate is to realize that imposing a *condition* on the system state effectively *reorients the state to the even grade direct basis*.

The direct basis is different from taking a measurement since it is reversible. The new system state must mathematically reflect the new imposed perspective by unconditionally applying the operator to any possible starting state.

Remember, the control-not conditionally inverts the *data* qubit when the *control* qubit is *True*. For the derivation, first start with two interacting qubits in classical states $A_0 B_0$ and place them both in the Pauli basis $A_0 B_0 \mathbf{P}_A \mathbf{P}_B = \mathbf{a1} \mathbf{b1}$. Remember that for qubit A , the Pauli basis transforms respectively the two classical states $\{A_1, A_0\}$ into $\pm \mathbf{a1}$ and the superposition states $\{A_+, A_-\}$ into $\pm \mathbf{a0}$. Assign the role of “control” to qubit A and the role of “data” to qubit B . As Table 7.6 demonstrates, the *CNOT* behavior conditionally inverts qubit B only for the bottom two rows of the table, where the qubit state $\mathbf{a1} = +1$. This final output equation for the desired $CNOT_{AB}$ output is synthesized using the *gag.pl* tool or, in this case, simply by inspection.

Table 7.6: Desired Control-Not Operator $X = +\mathbf{a1}$ where $(\mathbf{a1} \mathbf{b1}) X = -\mathbf{b1}$

a1	b1	(a1 b1)	Desired CNOT
-	-	+	+
-	+	-	-
+	-	-	+
+	+	+	-
gag.pl "a1,b1" "+0,-1,+2,-3" → -b1			

Control qubit A in both phases
(a1 b0)(+a1) = -b0
(a1 b1)(+a1) = -b1
(a0 b0)(+a1) = -a0 a1 b0
(a0 b1)(+a1) = -a0 a1 b1
CNOT Operator $X = +\mathbf{a1}$

The desired operator is X such that $(\mathbf{a1\ b1})(X) = -\mathbf{b1}$ is simply the vector $X = +\mathbf{a1}$, because $(\mathbf{a1\ b1})(+\mathbf{a1}) = -(\mathbf{a1\ a1\ b1}) = -\mathbf{b1}$. The right side of Table 7.6 applies this operator to the four possible cases where qubit A is both in the classical phase $\{\mathbf{a1\ b0}, \mathbf{a1\ b1}\}$ (top two rows) and in the superposition phase $\{\mathbf{a0\ b0}, \mathbf{a0\ b1}\}$ (bottom two rows). The operator $+\mathbf{a1}$ *conditionally inverts* qubit B state when A is classical = 1, thereby representing the CNOT gate for the Pauli basis.

Multiplying state A by the odd grade vector operator $(\mathbf{a1})$ converts the state to the direct basis $\{\pm 1, \pm \mathbf{S}_A\}$ which effectively orients the state from the perspective of $+\mathbf{a1} = +1$. Since this right multiplication vector operator is of odd grade, an extra non-commutative inversion may be implemented, which must be accounted for! When the control bit is not in a classical state, the resulting trivector $(-\mathbf{a0\ a1\ bx})$ encodes this fact. If the roles of the two qubits are reversed, then Pauli $CNOT_{BA} = -\mathbf{b1}$ because $(\mathbf{a1\ b1})(-\mathbf{b1}) = -\mathbf{a1}$.

This analysis used the Pauli basis, but the conditionalize operator can also be converted to work in the standard basis by inverting the Pauli basis operator $(\mathbf{P}_A)^{-1}(\mathbf{a1}) = (1 + \mathbf{S}_A) \mathbf{a1} = (\mathbf{a0} - \mathbf{a1}) = A_0$, which is an odd grade state A_0 in the standard basis. Using the *ga.pl* tool to validate all combinations of the standard and dual basis states shown in Table 7.7, the converted operator $CNOT_{AB} = (\mathbf{a0} - \mathbf{a1}) = A_0 = [0 + - 0]$ works as expected for the control in all classical states, as listed in the middle column of the table.

The right-most column of Table 7.7 shows the cases when control A is in superposition states A_{\pm} , and the resulting trivector co-occurrences $\mathbf{S}_A B = (\pm \mathbf{a0\ a1\ b0} \pm \mathbf{a0\ a1\ b0})$ are consistent

but unintuitive. Since $(CNOT)^2 = -1$, then \mathbf{B} and \mathbf{M} also act like $CNOT$ because operators

$\mathbf{B}^2 = \mathbf{I}$ and $\mathbf{M}^2 = \mathbf{I}$ also invert half of the row states for each application.

Table 7.7: Summary of Operator $CNOT_{AB} = A_0$ for $\mathcal{Q}_2 = \{A, B\}$

Data B is	Control A is Classical	Control A is <i>Superposed</i>
Classical	$(A_0 B_0) CNOT_{AB} = B_0$	$(A_- B_0) CNOT_{AB} = S_A B_0$
	$(A_0 B_1) CNOT_{AB} = B_1$	$(A_- B_1) CNOT_{AB} = S_A B_1$
	$(A_+ B_0) CNOT_{AB} = -B_0 = B_1$	$(A_+ B_0) CNOT_{AB} = -S_A B_0 = S_A B_1$
	$(A_+ B_1) CNOT_{AB} = -B_1 = B_0$	$(A_+ B_1) CNOT_{AB} = -S_A B_1 = S_A B_0$
<i>Superposed</i>	$(A_0 B_-) CNOT_{AB} = B_-$	$(A_- B_-) CNOT_{AB} = S_A B_-$
	$(A_0 B_+) CNOT_{AB} = B_+$	$(A_- B_+) CNOT_{AB} = S_A B_+$
	$(A_+ B_-) CNOT_{AB} = -B_- = B_+$	$(A_+ B_-) CNOT_{AB} = -S_A B_- = S_A B_+$
	$(A_+ B_+) CNOT_{AB} = -B_+ = B_-$	$(A_+ B_+) CNOT_{AB} = -S_A B_+ = S_A B_-$

```
ga.pl quiet vector "(-a0 - a1)(-b0 - b1)(a0 - a1)" = [-00++00- +00--00+]
ga.pl quiet vector "(-a0 - a1)(-b0 + b1)(a0 - a1)" = [0+-00+-0 0+-00-+0]
ga.pl quiet vector "(-a0 - a1)(+b0 - b1)(a0 - a1)" = [0+-00-+0 0-+00+-0]
ga.pl quiet vector "(-a0 - a1)(+b0 + b1)(a0 - a1)" = [+00--00+ -00++00-]...
```

		Table Output as Row for $A B CNOT$															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I n p u t	0	-		+		+		-		+		-		-		+	
	1		-	+			+	-			+	-			-	+	
	2		+	-			-	+			-	+			+	-	
	3	+			-		-		+		-		+		+		-
	4	+			-		+		-		+		-		+		-
	5		+	-			+	-			+	-			+	-	
	6		-	+			-	+			-	+			-	+	
R o w s	7	-		+		-		+		-		+		-		+	
	8	-			+		-		+		-		+		-		+
	9		-	+			-	+			-	+			-	+	
	10		+	-			+	-			+	-			+	-	
	11	+			-		+		-		+		-		+		-
AB	12	+			-		-		+		-		+		+		-
	13		+	-			-	+			-	+			+	-	
	14		-	+			+	-			+	-			-	+	
	15	-			+		+		-		+		-		-		+

Figure 7.1: Full Matrix Decode of $CNOT_{AB}$ gate output for $\mathcal{Q}_2 = \{A, B\}$

The CNOT operator appears to have all the required properties even though the algebraic form looks much different from the matrix format used in \mathbf{H}_4 . Therefore, to frame this control-not gate in a similar fashion to \mathbf{H}_4 , it is necessary to recast the derivation using the computational basis projectors. It helps that the solution is already in hand. Fortunately, Table 7.7 is fully expanded in Figure 7.1 into a matrix containing the vector notation for each possible starting input case multiplied times $CNOT$:

Obviously this matrix possesses significant redundancy: if the key rows 5, 6, 9 and 10 (which define the classical cases for \mathcal{Q}_2) are isolated, the result appears in Table 7.8. This same result can also be obtained analytically by using the facts that $1/A_0 = A_1$ and $1/B_0 = B_1$. For example, the specific equation $A_1 B_1 CNOT_{AB} = (+1)(B_0)$ can be solved for the operator $CNOT_{AB} = B_0 A_0 B_0 = A_1 B_0 B_0 = A_0$ and the other cases are shown in Table 7.8. Notice that the product of (+1) represents qubit A in the reversible direct basis state of (+1).

Table 7.8: Row by Row Operator Solutions for $CNOT$ for $\mathcal{Q}_2 = \{A, B\}$

	Inputs		$A B CNOT_{AB}$		$CNOT_{AB} =$	$A B CNOT_{BA}$		$CNOT_{BA} =$
	$A B$	$\mathbf{a0 a1 b0 b1}$	B	5 6 9 10		A	5 6 9 10	
5	$A_1 B_1$	- + - +	$B_0 = [+ - + -]$	$B_0 A_0 B_0 = A_0$	$A_0 = [- - + +]$	$= B_0 A_0 A_0 = B_1$		
6	$A_1 B_0$	- + + -	$B_1 = [- + - +]$	$B_1 A_0 B_0 = A_0$	$A_1 = [+ + - -]$	$= B_1 A_0 A_1 = B_1$		
9	$A_0 B_1$	+ - - +	$B_1 = [- + - +]$	$B_0 A_1 B_0 = A_0$	$A_1 = [+ + - -]$	$= B_0 A_1 A_1 = B_1$		
10	$A_0 B_0$	+ - + -	$B_0 = [+ - + -]$	$B_1 A_1 B_0 = A_0$	$A_0 = [- - + +]$	$= B_1 A_1 A_0 = B_1$		

This result shows that independent of starting state, the $CNOT$ operator is always the same expression. The *above procedure* has not been formalized for arbitrary multivectors X , where different individual starting states require different equation solutions to produce the desired

result. The complex conjugate operator is the simplest example that *cannot be written* as an unconditional geometric product, and instead uses the reverse operator.

7.2.2 Control-Hadamard Gate for \mathcal{Q}_2

An inversion operator is always a 180° phase shift and a Hadamard gate is half that phase amount. Since the control-not gate exists, perhaps a control-Hadamard gate also exists whose square is the control-not operator. The *gasolve.pl* tool immediately found two operator occurrences: $CHAD_{AB} = (-1 + A_0)$ and inverse $(1 + A_1)$ (see Figure 7.2) whose squares are $(-1 + A_0)^2 = A_0 = CNOT_{AB}$ and the 4th power is the inverter $(-1 + A_0)^4 = A_0^2 = -1$.

The role-reversed operator is $CHAD_{BA} = (1 + B_0)$ where $(1 + B_0)^2 = -B_0 = B_1$. Both CHAD operators appear to *concurrently* change to the *direct* basis and perform an *inversion* as seen in Figure 7.2.

```
gasolve.pl "a0,a1" "(X)(X)" "(a0 - a1)"
Found Match for X = - 1 + a0 - a1 in (X)(X) = + a0 - a1
Found Match for X = + 1 - a0 + a1 in (X)(X) = + a0 - a1
Attempted 80 with 2 found.
ga.pl "(a0 - a1)(b0 - b1)(-1 + a0 - a1)(-1 + a0 - a1)"
+ b0 - b1 <= correctly does NOT invert
ga.pl "(- a0 + a1)(b0 - b1)(-1 + a0 - a1)(-1 + a0 - a1)"
- b0 + b1 <= correctly DOES invert
```

Figure 7.2: Control-Hadamard Operator $CHAD_{AB}$ for $\mathcal{Q}_1 = \{A\}$ and $\mathcal{Q}_2 = \{A, B\}$

These relationships suggest the *CNOT* operator has a 90° phase relationship to the unconditional inverter, and the control-Hadamard gate a 45° phase relationship. A similar 45° relationship exists between the Pauli spin and the Hadamard operators because $(\pm \mathbf{P}_A)^2 = \mathbf{S}_A$ and $(\pm \mathbf{P}_A)^4 = (\mathbf{S}_A)^2 = -1$. This helps explain why the Pauli spin rotates the diagonal bases

by 45° from the off-diagonal to vertical/horizontal axis. The *CHAD* also exists for \mathbf{B} and \mathbf{M} operators because *gasolve.pl* found the solutions $\sqrt{\mathbf{B}} = (+1 + \mathbf{a0 a1} + \mathbf{b0 b1} - \mathbf{a0 a1 b0 b1}) = \mathbf{B}^2 + \mathbf{B} = \mathbf{I}^- + \mathbf{B} = \mathbf{B}(\mathbf{B} + 1)$, which has the same sparse form as $(-1 + A_0)$ and $\sqrt{\mathbf{M}} = (+1 + \mathbf{a0 a1} - \mathbf{b0 b1} + \mathbf{a0 a1 b0 b1}) = \mathbf{M}^2 + \mathbf{M} = \mathbf{I}^- + \mathbf{M}$. Each of these operators has at least eight other square roots, each of which contains a scalar, five bivectors, and a 4-vector. The \mathbf{Q}_2 geometric product operators are summarized in Table 7.9.

Table 7.9: Operator Summary for $\mathbf{Q}_2 = \{A, B\}$

Gate/Operator	Separable State ($A B$)	Inseparable State $AB\mathbf{B} = \mathbf{B}_i$ or $AB\mathbf{M} = \mathbf{M}_i$
Hadamard 90°	$A_0 B_0 (\mathbf{S}_A \mathbf{S}_B) = A_+ B_+$	$\mathbf{B}_i (\mathbf{B}) = \mathbf{B}_{i+1}$ or $\mathbf{M}_i (\mathbf{M}) = \mathbf{M}_{i+1}$
Inverter 180°	$AB(\mathbf{S}_A^2 \text{ or } \mathbf{S}_B^2) = -AB$	$\mathbf{B}_i (\mathbf{B})^2 = \mathbf{B}_{i+2}$ or $\mathbf{M}_i (\mathbf{M})^2 = \mathbf{M}_{i+2}$
Control-Had $_{AB}$	$AB(-1 + A_0)^2 = AB A_0$	$\sqrt{\mathbf{B}} = \mathbf{I}^- + \mathbf{B}$ or $\sqrt{\mathbf{M}} = \mathbf{I}^- + \mathbf{M}$
Control-Not $_{AB}$	$AB A_0 = \pm B$ or $\pm \mathbf{S}_A B$	\mathbf{B} or \mathbf{M} since $\mathbf{B}^2 = \mathbf{I}^-$ and $\mathbf{M}^2 = \mathbf{I}^-$
Computational	$ABC_{\pm\pm\pm\pm} = \mathbf{I}^\pm$ or random	$(\mathbf{B}_i \text{ or } \mathbf{M}_i) C_{\pm\pm\pm\pm} = \mathbf{I}^\pm$ or random
W/Operators $\mathbf{B} = (\mathbf{S}_A + \mathbf{S}_B)$, $\mathbf{M} = (\mathbf{S}_A - \mathbf{S}_B)$ and $C_{\pm\pm\pm\pm} = (1 \pm \mathbf{a0})(1 \pm \mathbf{a1})(1 \pm \mathbf{b0})(1 \pm \mathbf{b1})$		

The entangled Bell or magic basis \mathbf{B}_i or \mathbf{M}_i can be utilized as operator states equivalent to the direct basis. Since every Bell and magic state has the state pair complements in the same basis, selecting the right basis should produce the equivalent of the direct basis in \mathbf{Q}_1 . If the basis matches with the operator, then the result is either $\mathbf{B}_i \mathbf{B}_i = \mathbf{I}^+$ or $\mathbf{B}_i \mathbf{B}_{i+2} = \mathbf{I}^-$; otherwise the outcome results in concurrent spinor expression $(\pm \mathbf{S}_A \pm \mathbf{S}_B)$, which when measured is observed as 50% random values of ± 1 .

7.2.3 Two qubit measurement

Measurement for the separable standard and dual bases are the same as for individual qubits, which assumes known basis for each qubit. Measurement works the same with R_k operators.

7.2.4 Computational Basis for Two Qubits

Just as for one qubit, an individual row can be selected for \mathcal{Q}_2 by taking each separate qubit operator $R_{A0} = (1 - \mathbf{a0})(1 - \mathbf{a1}) = [+000]$ and $R_{B0} = (1 - \mathbf{b0})(1 - \mathbf{b1}) = [+000]$ and forming their geometric product $R_{A0} R_{B0} = R_{AB0} = (1 - \mathbf{a0})(1 - \mathbf{a1})(1 - \mathbf{b0})(1 - \mathbf{b1}) = [+0000000 00000000]$. Likewise, the various projection operators P_k are formed by the relationship $P_k = -R_k$ and are irreversible. Since *all* terms of form $(1 \pm \mathbf{x})$ have no multiplicative inverse, this means that $\det(1 \pm \mathbf{x}) = 0$. This also means that, for any \mathcal{Q}_q , since $\det(X)\det(Y) = \det(XY) = 0$ then also $\det(P_k) = 0$ and $\det(R_k) = 0$.

The scaling of P_k and R_k for any number of qubits works the same as for one qubit, except that the product of individual idempotent operators $(-1) R_{Ak} R_{Bk} = P_{ABk}$ does not produce a larger idempotent term P_{ABk} because of non-commutativity. For example, the projector equation $P_{AB0} = (-1)(1 - \mathbf{a0})(1 - \mathbf{a1})(1 - \mathbf{b0})(1 - \mathbf{b1}) = [-0000000 00000000]$ but that expression squared $P_{AB0}P_{AB0} = [-0+0+0+0 00000000] \neq P_{AB0}$ but instead $(P_{AB0})^4 = [-0000000 00000000]$. Likewise, for $R_{ABCO} = (R_{ABCO})^6 = [+0000000 0...0 0...0 0...0 0...0 0...0 0...0 00000000]$ and $R_{ABCD0} = (R_{ABCD0})^8 = [+0000000 00000000 \text{ and thirty more of } 00000000]$.

The conclusion of this result is that geometric products of idempotent operators produce new cyclic operators that have idempotent-like properties that return back to themselves when $(X)^n = X$, except with a higher power where $n = 2q$, instead of always $n = 2$. This is related to

the fact that the overall size of the space grows exponentially larger, resulting in more degrees of internal freedom. Also depending on the size of n , notice that sometimes $(\pm 1)^2 = +1$ and other times $(\pm 1)^2 = -1$. This proposed expanded cyclic definition of idempotency is interesting because these products still maintain the other important properties of logic AND decode of states and $\det(XY) = 0$.

Computing these products reveals that some sparse invariants \mathbf{I}^- are idempotent: $(P_{AB0})^3 = -1 + \mathbf{a0} + \mathbf{b1} - \mathbf{a0} \mathbf{b1} = [-0-0-0-0 \ 00000000]$, $(R_{ABC0})^5 = -1 + \mathbf{a0} + \mathbf{c1} - \mathbf{a0} \mathbf{c1} = [-0-0-0-0 \ -0-0-0-0 \ -0-0-0-0 \ -0-0-0-0 \ 00000000 \ 00000000 \ 00000000 \ 00000000]$ and $(R_{ABCD0})^7 = -1 + \mathbf{a0} + \mathbf{d1} - \mathbf{a0} \mathbf{d1} = [16 \text{ of } -0-0-0-0 \text{ and } 16 \text{ of } 00000000]$. What this shows is that some idempotent operators $(-1)(1-\mathbf{a0})(1-\mathbf{b1}) = -1 + \mathbf{a0} + \mathbf{b1} - \mathbf{a0} \mathbf{b1} = [-000]$, when embedded into large spaces, are still idempotent even though their values are effectively spread throughout the space due to the effect of don't cares for other unused inputs.

The problem with the projection operators P_k not being exactly idempotent is that the corresponding E_k are not exactly eigenvectors for \mathbf{Q}_2 . Many *other* solutions were found while using *gasolve.pl* to look for eigenvectors E_k where $(E_k)^2 = 1$. Even though the $P_k = -R_k$ are not exactly idempotent and the terms $E_0 = R_0 - 1 = [0\text{-----} \ \text{-----}]$ still produce the matching filter expressions $P_0 = [-0000000 \ 00000000]$ and $R_0 = [+0000000 \ 00000000]$. Due to larger phase space, it is easy to show $(E_k)^6 = +1$. Also the R_k produce valid measurement answers even for Bell states. For example, $A_0 B_0 \mathbf{B} = \mathbf{B}_0 = R_1 - R_2 + R_4 - R_7 - R_8 + R_{11} - R_{13} + R_{14}$. Measuring valid states in \mathbf{B}_0 produce invariant answers $\mathbf{B}_0 R_1 = R_7 + R_{13}$ and $\mathbf{B}_0 R_2 = P_4 + P_{14}$, but invalid states produce random answers $\mathbf{B}_0 R_0 = \underline{P}_6 + \underline{R}_{12}$.

7.3 Three-Qubit Operators

All operators for separable states for one and two qubits apply as well to three or more qubits. The four phase derivatives of the concurrent Hadamard operator for three spinors form the equivalent of the Bell and magic basis ($\mathbf{S}_A \pm \mathbf{S}_B \pm \mathbf{S}_C$) plus two additional sets. Additionally, the Fredkin and Toffoli gates (from Section 3.4) can finally be attempted, using the same design procedure developed for the control-not gate.

7.3.1 Toffoli Gate in Q_3

The same procedure used to define the control-not gate was applied to the Toffoli gate. The process for three qubits utilized the following steps:

1. Choose start state $A_1 B_1 C_1$ and convert to Pauli basis: $A_1 B_1 C_1 \mathbf{P}_A \mathbf{P}_B \mathbf{P}_C = \mathbf{a1 b1 c1}$.
2. Recreate state `gag.pl "a1,b1,c1" "-0,+1,2,-3,+4,-5,-6,+7" = a1 b1 c1`
3. Invert the last two bits to create the desired end state

```
gag.pl "a1,b1,c1" "-0,+1,2,-3,+4,-5,+6,-7"
➔ + c1 + a1 c1 + b1 c1 - a1 b1 c1
```

4. Solve for the operator to transition from the start state to the end state:

```
gasolve.pl "a1,b1,c1" "(a1 b1 c1)(X)" "c1 +a1 c1 +b1 c1 -a1 b1 c1"
Found Match for X = - 1 + a1 - b1 - a1 b1
```

5. Factor this by hand $(-1 + \mathbf{a1} - \mathbf{b1} - \mathbf{a1 b1}) = (1 + \mathbf{b1})(-1 + \mathbf{a1})$ and verify

```
Input expression is (1 + b1)(-1 + a1)
INPUTS: a1 b1 | - 1 + a1 - b1 - a1 b1 | OUTPUT
*****
ROW 00: - - | - - + - | +
ROW 01: - + | - - - + | +
ROW 02: + - | - + + + | -
ROW 03: + + | - + - - | +
*****
Row counts for outputs of ZERO=0, PLUS=3, MINUS=1 for TOTAL=4 rows.
```

6. The square of this potential Toffoli Operator T should be unity or $TT = +1$

```
ga.pl table "(1 + b1)(-1 + a1)(1 + b1)(-1 + a1)"
Input expression is (1 + b1)(-1 + a1)(1 + b1)(-1 + a1)
INPUTS: a1 b1 | - 1 + a1 - b1 - a1 b1 | OUTPUT
```

```

*****
ROW 00: - - | - - + - | +
ROW 01: - + | - - - + | +
ROW 02: + - | - + + + | -
ROW 03: + + | - + - - | +
*****
Row counts for outputs of ZERO=0, PLUS=3, MINUS=1 for TOTAL=4 rows.

```

Unfortunately, the result produced by using this process does not appear to be correct because the operator it found happens to be idempotent, which means multiple applications of it produce *no* net effect. Translating this operator back into the standard basis still leaves nothing more than Pauli inverse operator behavior for multiple applications. Consequently, this design approach will not work for idempotent operators. Moreover, the search space is too large with $q=3$ or $n=6$ to search the entire space using the *gasolve.pl* tool, as occurred for two qubits.

Another approach utilized the Pauli basis to shrink the search space to the size of $n = 3$, looking at only the classical cases and then searching that space for operators where $X^2 = +1$. Those solutions would be equivalent to eigenvectors for qutrits and possible candidates for Toffoli and Fredkin Gates. Starting with $A_1 B_1 C_1 P_A P_B P_C = \mathbf{a1 b1 c1}$ and running the equation *gasolve.pl* "a1 b1 c1" "(X)(X)" "1" produced 92 possibilities out of 6560, many of which had already been found for a single qubit. The only new ones include all the vector and sign combinations of two basic forms $(\pm \mathbf{a1 b1} \pm \mathbf{a1 c1}) = \pm \mathbf{a1}(\pm \mathbf{b1} \pm \mathbf{c1})$ and sum of the other known eigenvectors $(\pm \mathbf{a1} \pm \mathbf{b1} \pm \mathbf{c1} \pm \mathbf{a1 c1} \pm \mathbf{b1 c1})$. The analysis of these two operators indicates they do not represent a Fredkin (or Toffoli) gates because the operator $(\mathbf{a1 b1} + \mathbf{a1 c1})$ is $[-00++00-]$. So $(\mathbf{a1 b1 c1})(\mathbf{a1 b1} + \mathbf{a1 c1}) = (\mathbf{b1} - \mathbf{c1}) = [0+-00+-0]$ which isolates the important case when $b1$ and $c1$ are different but does not conditionalize it.

The another attempt assumes $A = (\pm \mathbf{a0} \pm \mathbf{a1})$ then $(A)^2 = -1$ and $(A)^{-1} = -A$. So the Toffoli operator multivector T must satisfy the equation $A_I B_I C_I T = (+1)(+1)C_0$ because both control qubits are rotated to direct basis and the data qubit inverts, while other cases are the identity.

$$A_I B_I C_0 T = (+1)(+1) C_I \text{ (or alternatively the inverse: } A_I B_I C_I T = (+1)(+1)C_0) \quad (7.1)$$

To solve for T for each term, left multiply both sides by $1/X_I = X_0$ then simplify $C_I C_I = -1$

$$T = C_0 B_0 A_0 C_I = -A_0 B_0 C_0 C_I = A_0 B_0 \text{ and also remember } A_I B_I = A_0 B_0 \quad (7.2)$$

If the four classical decode cases $\{-A_I B_I, -A_I B_0, -A_0 B_I, A_0 B_0\}$ are each solved for T , then the simultaneous solutions must be respectively $\{-A_0 B_0, A_0 B_0, A_0 B_0, A_0 B_0\} = [-++++]A_0 B_0$.

These simultaneous solutions could be represented in matrices, but the operator must be applied conditionally to the isolated solutions, which means that the states must be discernible from each other. This conditional application of operators is a formal problem for implementing arbitrary logic in quantum systems, and may be related to not finding an operator that *unconditionally* performs the Toffoli operation. This is another case where conditional operators may be required to implement a particular operator and this particular operator cannot be expressed as a geometric product of multivectors.

The final attempt succeeded by remembering the Toffoli operator is called the control-control-not. The approach is to *concurrently* take the control-not operator from two different control bits for register $A B C$ then $CNOT_{AC} = A_I$ and $CNOT_{BC} = B_0$. The co-occurrence is $TOF_{AB} = CNOT_{AC} + CNOT_{BC} = -\mathbf{a0} + \mathbf{a1} + \mathbf{b0} - \mathbf{b1}$ and this is already looking good because $TOF^2 = 1!$ This promising result can now be applied to a real state:

$$\begin{aligned} A_0 B_0 C_0 TOF_{AB} &= \mathbf{a0} \mathbf{c0} - \mathbf{a0} \mathbf{c1} - \mathbf{a1} \mathbf{c0} + \mathbf{a1} \mathbf{c1} + \mathbf{b0} \mathbf{c0} - \mathbf{b0} \mathbf{c1} - \mathbf{b1} \mathbf{c0} + \mathbf{b1} \mathbf{c1} = \\ &= [00000+-0 0-+00000 0+-00-+0 00000+-0 0-+00000 0+-00-+0 00000+-0 0-+00000] \quad (7.3) \end{aligned}$$

The states for $A_0 B_0 C_0 TOF_{AB}$ are analyzed in Table 7.10 and this produces the result that for the four states when qubits A and B are both classical, the output orientation *is the same* as qubit C for control states A_0 and B_0 and *is inversion* of qubit C for control states A_1 and B_1 !

Table 7.10 Valid State Rows for $A_0 B_0 C_0 TOF_{AB}$ in Q_3

Row _k	State Combinations						Active States	$A_0 B_0 C_0 (TOF_{AB})$	
	a0	a1	b0	b1	c0	c1			
R_{21}	-	+	-	+	-	+	$A_1 B_1 \& C_1$	-	Inverted
R_{22}	-	+	-	+	+	-	$A_1 B_1 \& C_0$	+	
R_{41}	+	-	+	-	-	+	$A_0 B_0 \& C_1$	+	Unitary
R_{42}	+	-	+	-	+	-	$A_0 B_0 \& C_0$	-	
8 rows	$A_{\text{classical}}$		$B_{\text{superpose}}$		$C_{\text{classical}}$		$A_c B_s \& C_c$	\pm	Mixed states
8 rows	$A_{\text{superpose}}$		$B_{\text{classical}}$		$C_{\text{classical}}$		$A_s B_c \& C_c$	\pm	
44 rows	<i>All cases not covered above</i>						<i>otherwise</i>	0	Invalid

7.3.2 Fredkin Gate in Q_3

The Fredkin gate design (from Section 3.4) was also attempted, but a problem occurred.

First, assume three qubits in Pauli basis: $A_0 B_0 C_0 P_A P_B P_C = - \mathbf{a1} \mathbf{b1} \mathbf{c1}$ and remember the goal is to swap the state of qubits B and C when qubit A has an active low value. This is shown in Figure 7.3 as rows 0-3, but only rows 1-2 are classically detectible.

```

Input expression is (- a1 b1 c1)
INPUTS: a1 b1 c1 | - a1 b1 c1 | OUTPUT
*****
ROW 00: - - - | + | +
ROW 01: - - + | - | -
ROW 02: - + - | - | -
ROW 03: - + + | + | +
*****
ROW 04: + - - | - | -
ROW 05: + - + | + | +
ROW 06: + + - | + | +
ROW 07: + + + | - | -
*****
Row counts for outputs of ZERO=0, PLUS=4, MINUS=4 for TOTAL=8 rows.

```

Figure 7.3: Fredkin Gate States for Q_3 in Pauli Basis

Figure 7.3 illustrates the problem: independent of the states of B and C , the output state cannot distinguish between rows 01 and 02. This condition exists because the only observable change due to this gate requires that the data qubits have opposite values and then both are simultaneously and conditionally inverted. These two simultaneous inversions are equivalent to multiplying the overall system state by $(-1)(-1) = +1$. Therefore, this results in *no* net effect to the overall system state product except that the control qubit is constrained. A Fredkin gate solution F is $A_0 B_0 C_1 F = (+1) B_1 C_0$, but the product of the two states is identical $B_0 C_1 = B_1 C_0 = \mathbf{b0 c0 -b0 c1 -b1 c0 +b1 c1} = [00000+-0 0-+00000]$. At the current time no Fredkin operator has been designed using \mathcal{Q}_3 due to the unresolved problem of general operator design or due to two outputs. This solution space is too large to search.

7.3.3 Computational Basis for Three Qubits

The computational basis operators $R_k = (1 \pm \mathbf{a0})(1 \pm \mathbf{a1})(1 \pm \mathbf{b0})(1 \pm \mathbf{b1})(1 \pm \mathbf{c0})(1 \pm \mathbf{c1})$, $P_k = -R_k$ and their relationships to $E_k = R_k - 1$ continue to work for \mathcal{Q}_3 but $(E_k)^{80} = +1$. For example, $R_0 = [+0000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000]$ and $E_0 = [0----- ----- ----- ----- ----- ----- ----- -----]$.

7.3.4 Bell Basis for Three Qubits

The Bell/magic operators and basis have previously been shown to work for any \mathcal{Q}_q . For three qubits, the four equivalent concurrent Hadamard operators are $(\mathbf{a0 a1} \pm \mathbf{b0 b1} \pm \mathbf{c0 c1})$. Recursively applying them from a starting state $A_0 B_0 C_0$ produces four separate sets of

positive/negative co-exclusion states as seen in Figure 7.4, using the vector output mode of the *ga.pl* tool. It is now becoming very clear why the vector notation was adopted early.

```

Generator polynomial for 5 iterations is (a0 a1 - b0 b1 - c0 c1) ← generator --
[---+00+ +00+--- 0--0--+ -++-0--0 0--0--+ -++-0--0 -+++00+ +00+---]
Equation is (a0 - a1)(b0 - b1)(c0 - c1) times Generator ** n (with n = 1-5)
0 [00000000 00000000 00000+0 0--+0000 00000-+0 0+-00000 00000000 00000000]
1 [00000+0 0--+0000 0+-0-00+ +00-0+-0 0+-0+00- -00+0-+0 00000-+0 0+-00000]
2 [0+-0+00- -00+0-+0 -00+0000 0000+00- +00-0000 0000-00+ 0+-0-00+ +00-0+-0]
3 [00000-+0 0+-00000 0+-0+00- -00+0-+0 0+-0-00+ +00-0+-0 00000+0 0-+00000]
4 [0+-0-00+ +00-0+-0 +00-0000 0000-00+ -00+0000 0000+00- 0+-0+00- -00+0-+0]
5 [00000+0 0--+0000 0+-0-00+ +00-0+-0 0+-0+00- -00+0-+0 00000-+0 0+-00000]

Generator polynomial for 5 iterations is (a0 a1 - b0 b1 + c0 c1) ← generator +-
[+++0+0+ 0+0+--- -00-+++ +---+00- -00-+++ +---+00- +---+0+0 0+0+---]
Equation is (a0 - a1)(b0 - b1)(c0 - c1) times Generator ** n (with n = 1-5)
0 [00000000 00000000 00000+0 0--+0000 00000-+0 0+-00000 00000000 00000000]
1 [00000+0 0--+0000 0+-0+00- -00+0-+0 0+-0-00+ +00-0-+0 00000-+0 0+-00000]
2 [0+-0-00+ +00-0-+0 +00-0000 0000-00+ -00+0000 0000+00- 0+-0+00- -00+0-+0]
3 [00000-+0 0+-00000 0+-0-00+ +00-0-+0 0+-0+00- -00+0-+0 00000+0 0-+00000]
4 [0+-0+00- -00+0-+0 -00+0000 0000+00- +00-0000 0000-00+ 0+-0-00+ +00-0-+0]
5 [00000+0 0--+0000 0+-0+00- -00+0-+0 0+-0-00+ +00-0-+0 00000-+0 0+-00000]

Generator polynomial for 5 iterations is (a0 a1 + b0 b1 - c0 c1) ← generator +-
[+00+--- -+++00+ ---+0--0 0--0--+ -+++0--0 0--0--+ +00+--- -+++00+]
Equation is (a0 - a1)(b0 - b1)(c0 - c1) times Generator ** n (with n = 1-5)
0 [00000000 00000000 00000+0 0--+0000 00000-+0 0+-00000 00000000 00000000]
1 [00000+0 0--+0000 0+-0-00+ +00-0-+0 0+-0+00- -00+0-+0 00000-+0 0+-00000]
2 [0+-0+00- -00+0-+0 +00-0000 0000-00+ -00+0000 0000+00- 0+-0-00+ +00-0-+0]
3 [00000-+0 0+-00000 0+-0+00- -00+0-+0 0+-0-00+ +00-0-+0 00000+0 0-+00000]
4 [0+-0-00+ +00-0-+0 -00+0000 0000+00- +00-0000 0000-00+ 0+-0+00- -00+0-+0]
5 [00000+0 0--+0000 0+-0-00+ +00-0-+0 0+-0+00- -00+0-+0 00000-+0 0+-00000]

Generator polynomial for 5 iterations is (a0 a1 + b0 b1 + c0 c1) ← generator ++
[0+0+--- +---+0+0 +---+00- -00-+++ +---+00- -00-+++ 0+0+--- +---+0+0]
Equation is (a0 - a1)(b0 - b1)(c0 - c1) times Generator ** n (with n = 1-5)
0 [00000000 00000000 00000+0 0--+0000 00000-+0 0+-00000 00000000 00000000]
1 [00000+0 0--+0000 0+-0+00- -00+0-+0 0+-0-00+ +00-0+-0 00000-+0 0+-00000]
2 [0+-0-00+ +00-0-+0 -00+0000 0000+00- +00-0000 0000-00+ 0+-0+00- -00+0-+0]
3 [00000-+0 0+-00000 0+-0-00+ +00-0-+0 0+-0+00- -00+0-+0 00000+0 0-+00000]
4 [0+-0+00- -00+0-+0 +00-0000 0000-00+ -00+0000 0000+00- 0+-0-00+ +00-0-+0]
5 [00000+0 0--+0000 0+-0+00- -00+0-+0 0+-0-00+ +00-0+-0 00000-+0 0+-00000]

```

Figure 7.4: Entangled Concurrent Hadamard States for \mathbb{Q}_3

Each set of states acts like a single co-exclusion which represents only one classical bit for the three entangled qubits, thereby indicating that two bits are erased. These higher order Bell/magic states for \mathbb{Q}_q are generated using concurrent Hadamard gates just as with the two qubit Bell/magic states. Remember these states also act as the direct basis operators.

7.3.5 Three-qubit measurement

The same R_k and P_k rules as two qubits apply for three or more qubits. Also $(E_k)^{80} = +1$.

7.3.6 Example Quantum Computation

The following sequence of steps is shown for using a Toffoli gate computation in \mathcal{Q}_3 . All of the steps use an algebraic notation and have been validated using the *ga.pl* tool.

1. Force the qubits to a known classical states by making a measurement, thereby erasing all

the information content: $A B C (1+\mathbf{a0})(1-\mathbf{a1})(1+\mathbf{b0})(1-\mathbf{b1})(1+\mathbf{c0})(1-\mathbf{c1}) \Rightarrow A_0 B_0 C_0 =$
 $R_{21} + P_{22} + P_{25} + R_{26} + P_{37} + R_{38} + R_{41} + P_{42}$

2. Assign specific values to qubits: $A_0 B_0 C_0 \Rightarrow A_0 B_0 C_0$ or $A_0 B_0 C_0 (\mathbf{S}_A \mathbf{S}_B)^2 \Rightarrow A_1 B_1 C_0$

3. Apply the Toffoli operator to constrain the valid states $A B C_0 TOF_{AB} \Rightarrow P_{21} + R_{22} + R_{41}$
 $+ P_{42} +$ sixteen other rows for the superposition cases.

4. Take a measurement using projection operator returns an answer that is a sparse

invariant: $A B C_0 TOF_{AB} P_{21} \Rightarrow R_9 + R_{11} + R_{17} + R_{19} + R_{25} + R_{27} + R_{41} + R_{43} \Rightarrow \mathbf{I}^+$ and

other case $A B C_0 TOF_{AB} P_{42} \Rightarrow P_{20} + P_{22} + P_{36} + P_{38} + P_{24} + P_{44} + P_{46} + P_{52} + P_{56} \Rightarrow \mathbf{I}^-$.

Other projection operators return random values due to a mixture of P_k and R_k values.

The original goal of this dissertation was to include a description of quantum algorithms that outperformed their classical counterparts, such as Grover's search algorithm or Quantum Fourier Transform (QFT) used by Shor's algorithm. Unfortunately, the reality of the schedule did not permit this. The basic ground work for qubits, ebits, entanglement, Fourier basis, concurrency and measurement have however been developed, which should enable this work at a later time. This concludes the chapter on quantum computation.

CHAPTER 8

SUMMARY AND CONCLUSIONS

8.1 Summary of Contributions

Many novel contributions from several different topic areas resulted from this effort. The following accomplishments are related exclusively to geometric algebra and logic tools.

- Demonstrated that Boolean algebra operators can be represented as orthonormal vectors in linear algebras such as Galois Field- GF(2) and geometric algebra \mathbf{G}_n using only the values $\{-1, 0, +1\}$ and addition and multiplication operators. These algebraic representations are universal or Boolean complete. For \mathbf{G}_n , multiplication is the XNOR operator, so for orthonormal vectors \mathbf{e}_i then $\mathbf{e}_i = 1/\mathbf{e}_i$. Addition is both NAND-like and NOR-like.
- Adopted and validated the interpretation of addition as concurrency (or co-occurrence), and multiplication for state interaction and operator implementation. Mutually exclusive states are properly represented as additive inverses, $\mathbf{x} + \bar{\mathbf{x}} = 0$, where the value of “0” is assigned the special meaning of “does not occur.”
- Demonstrated that the linear matrix operators formed by only the input state vectors are not closed for all Boolean logic operators unless the size of the vector space is increased to include all product combinations of the input vector set as additional

linearly independent degrees of freedom. This set of $N=2^n$ elements can be generated

as the sum of n orthonormal input vectors \mathbf{e}_i using $\prod_{i=0}^{n-1} (1 + \mathbf{e}_i)$.

- Demonstrated that the $N=2^n$ binary combinations of the input vector set for \mathbf{G}_n represent linearly independent states and are equivalent to the grids of a logic decode Karnaugh map. Any particular linearly independent decode state can be expressed as the form $\prod_{i=0}^{n-1} (-1)(1 \pm \mathbf{e}_i)$. Additive combinations hereof produce the final multivector state. These linearly independent states represent the idempotent projectors P_k for \mathbf{G}_2 that correspond to the pair-wise orthogonal eigenmultivectors E_k where $E_k P_k = P_k E_k = P_k$.
- Demonstrated that $(P_k)^{-1}$ does not exist, which indicates that $\det(P_k) = 0$ even though the exact analytical form for the determinant of a multivector was not derived.
- Demonstrated for \mathbf{G}_2 that the eigenmultivectors $E_k = (\pm \mathbf{a}_0 \pm \mathbf{a}_1 \pm \mathbf{a}_0 \mathbf{a}_1)$ are simply the directed major diagonals of a cube of length $\sqrt{N-1}$ which form the equally spaced corners of dual tetrahedrons. The faces of these tetrahedrons represent the projection operators P_k with unitarity constraint $\sum_k P_k = +1$ and their inverses R_k where $\sum_k R_k = -1$. For any \mathbf{G}_n then $R_k = -P_k$ and $E_k = R_k - 1$. Showed that $(E_k)^2 = 1$.
- Demonstrated that the total number of unique multivectors for \mathbf{G}_n is 3^N , which is the ternary count enumerations $\{0, -, +\}$ over the $N=2^n$ independent n -vector terms.

- Demonstrated that any state/operator S in \mathbf{G}_n can be expressed using values I_k as the

matrix diagonals $\begin{bmatrix} I_0 & & & \\ & I_1 & & \\ & & \dots & \\ & & & I_k \end{bmatrix}$ written as a vector $[I_0 \ I_1 \ \dots \ I_k]$ or alternatively

$$S = \sum_k I_k P_k . \text{ For } \mathbf{G}_1 \text{ and } \mathbf{G}_2 \text{ the } I_k \text{ values are eigenvalues and the } E_k \text{ are}$$

eigenmultivectors.

- Demonstrated in \mathbf{G}_2 that for operators $\square \in \{+, gp\}$ the following relationships hold:

$$[a \ b \ c \ d] \square [w \ x \ y \ z] = [a \square w \ b \square x \ c \square y \ d \square z] \text{ for n-vectors. Multivectors}$$

can also be represented in this fashion but multiplication of multivectors must have precedence over n-vector addition in order to handle anti-commutative inversions correctly.

- Demonstrated that geometric algebra rules and the above principles can be refined into tools that enable the design of state and operator behaviors. The tools *ga.pl*, *gag.pl*, *gandg.pl* and *gasolve.pl* were created and validated for this effort. The output vectors $[R_0 \ R_1 \ R_2 \ \dots]$ can be generated for any equation using the *ga.pl* tool.

The next accomplishments are related to both geometric algebra and one qubit. These results demonstrate that geometric algebra can faithfully represent one qubit.

- Demonstrated that a qubit can be expressed as the concurrent sum of two orthonormal vectors $\{\mathbf{a0}, \mathbf{a1}\}$ for \mathbf{G}_2 . This multivector $A = (\pm \mathbf{a0} \pm \mathbf{a1})$ represents a co-occurrence

and defines a *quantum geometric algebra* for one qubit $\mathcal{Q}_1 = \mathbf{G}_2 = \mathbf{H}_2$. There are 81 possible states or operators for \mathcal{Q}_1 and they were all examined in detail.

- Demonstrated that the standard basis containing two classical states is represented as $A_{0/1} = (\pm \mathbf{a0} \mp \mathbf{a1})$ and the dual basis with two alternate phase superposition states as $A_{+/-} = (\pm \mathbf{a0} \pm \mathbf{a1})$. The spinor $\mathbf{S}_A = (\mathbf{a0} \mathbf{a1})$ (also Hadamard operator or pseudoscalar \mathbf{I}_A) switches between the two mutually exclusive phases while the inversion operator flips between states within the same phase. The spinor \mathbf{S}_A is equivalent to $\mathbf{S}_A = \sqrt{-1}$ since $(\mathbf{S}_A)^2 = -1$, or $\mathbf{S}_A = \sqrt{NOT}$.
- Demonstrated that the reversible basis operators for \mathcal{Q}_1 starting from the standard basis are: the Dual basis operator \mathbf{S}_A , the Pauli basis operator $(-1 + \mathbf{S}_A)$, the circular basis operators $\mathbf{a0}$ or $\mathbf{a1}$, and the measurement operators $A_{0/1}$ or $A_{+/-}$. All of the reversible basis states map either to all even or to all odd grade planes. The non-reversible computational basis operators are any of the combinations $C_{\pm\pm} = (1 \pm \mathbf{a0})(1 \pm \mathbf{a1})$. The computational operators do not have multiplicative inverses implying that $\det(C_{\pm\pm})=0$, which makes them irreversible.
- Measurement using basis operators produces constants of ± 1 or multivectors with 50/50 percent random values. Some operators also result in sparse invariant values signified by \mathbf{I}^+ (equivalent to sparse +1) and \mathbf{I}^- (equivalent to sparse -1) such that $(\mathbf{I}^-)^2 = (\mathbf{I}^+)^2 = \mathbf{I}^+$. Individual states with value = 0 effectively cancel due to destructive interference and *do not occur*.

- Demonstrated that twelve non-constant idempotent operators (where $X^2 = X$) exist for \mathcal{Q}_1 : $I^+_0 = (-1 \pm \mathbf{a0})$, $I^+_1 = (-1 \pm \mathbf{a1})$, and the eight combinations $(-1 \pm \mathbf{a0} \pm \mathbf{a1} \pm \mathbf{a0} \mathbf{a1})$, each of which has one of the other four I^+ operators as a factor. The inverses (or square roots) of these operators all have the idempotent property $X^2 = -X = X + X$. In addition, the invariant inversion operators $I^+_0 = (1 \pm \mathbf{a0})$, $I^+_1 = (1 \pm \mathbf{a1})$ both have the property $(I^-)^2 = I^+$. Multiple equations for I^+ and I^- exist with opposite phases.

Other results are related to both quantum geometric algebra and two qubits \mathcal{Q}_2 .

- Demonstrated that multiple qubits can be expressed as the product of single qubits $A B = (\pm \mathbf{a0} \pm \mathbf{a1})(\pm \mathbf{b0} \pm \mathbf{b1}) = \pm \mathbf{a0} \mathbf{b0} \pm \mathbf{a1} \mathbf{b0} \pm \mathbf{a0} \mathbf{b1} \pm \mathbf{a1} \mathbf{b1}$ thereby defining $\mathcal{Q}_2 = \mathbf{G}_4 = \mathbf{H}_4$. There are 43,046,721 possible states or operators for \mathcal{Q}_2 and several important identities were found by exhaustively searching with *gasolve.pl* (in less than five days). The \mathcal{Q}_n algebra may be related to Symplectic Clifford algebras.
- Demonstrated that the geometric product for \mathcal{Q}_q replaces the tensor product \otimes of \mathbf{H}_N since the normal multiplication of co-occurrences naturally generates all the product combinations.
- Demonstrated that if A, B, C are in standard or dual basis with $A B X = C$ then, using left multiplication, $X = (-B)(-A)C = BAC = -ABC$ because $A^2 = -1$ or $(A)(-A) = 1$ or $A^{-1} = -A$. For vectors where $1/\mathbf{x} = \mathbf{x}$ then with $\mathbf{a} \mathbf{b} \mathbf{x} = \mathbf{c}$ using left multiplication also $\mathbf{x} = \mathbf{b} \mathbf{a} \mathbf{c} = -\mathbf{a} \mathbf{b} \mathbf{c}$. However, this relationship depends on the number and order of terms.

- Exhaustively demonstrated that the concurrent Hadamard operator ($\mathbf{S}_A + \mathbf{S}_B$) does not have a multiplicative inverse, rendering it irreversible and states that can never be exited. Recursively applying this operator generates the four Bell states \mathbf{B}_i so that the concurrent Hadamard operator defines the Bell operator $\mathbf{B} = (\mathbf{S}_A + \mathbf{S}_B)$. Applying the operator $-\mathbf{B}$ cyclically generates the Bell states in the opposite order.
- Demonstrated that recursively applying the operator ($\mathbf{S}_A - \mathbf{S}_B$) cyclically generates the four magic states \mathbf{M}_i so the concurrent Hadamard *difference* defines the magic operator $\mathbf{M} = (\mathbf{S}_A - \mathbf{S}_B)$.
- Demonstrated that Bell and magic states each represent only one co-exclusion, rather than two, due to the erasure of one bit of information. Also $(\mathbf{B})^2 = \mathbf{I}^-$ and alternate phase $(\mathbf{M})^2 = \mathbf{I}^-$ where $(\mathbf{I}^-)^2 = (\mathbf{I}^+)^2 = \mathbf{I}^+$. The following square roots are also defined: $\sqrt{\mathbf{B}} = \mathbf{B}^2 + \mathbf{B} = \mathbf{I}^- + \mathbf{B}$ and $\sqrt{\mathbf{M}} = \mathbf{M}^2 + \mathbf{M} = \mathbf{I}^- + \mathbf{M}$.
- Demonstrated that the Bell and magic operators erase state information due to multiplicative cancellation of some states: $\mathbf{B}(-1 - \mathbf{S}_A \mathbf{S}_B) = \mathbf{M}(-1 + \mathbf{S}_A \mathbf{S}_B) = 0$, which means these states “cannot occur” in the result. These operators are idempotent regarding erasure.
- Demonstrated that Bell and magic operators can act as their own measurement operators, producing results that represent the sparse invariant constant values \mathbf{I}^\pm .
- Demonstrated that the control-Hadamard operator has definition $CHAD_{AB} = (-1 + A_0) = (-1 + \mathbf{a0} - \mathbf{a1})$ and control-not is $CNOT_{AB} = A_0 = (\mathbf{a0} - \mathbf{a1})$ with properties

$(CHAD_{AB})^2 = CNOT_{AB}$ and $(CNOT)^2 = -1$. The $\sqrt{\mathbf{B}}$ and $\sqrt{\mathbf{M}}$ represent the control-Hadamard operators for Bell/magic states.

- Demonstrated that the idempotent operators P_k for \mathbf{Q}_1 where $(P_k)^2 = P_k$, when multiplied together for $\mathbf{Q}_{q=2}$ form cyclic operators where $(P_k)^{2q} = P_k$. Also showed for $\mathbf{Q}_{q=2}$ that $(E_k)^6 = 1$.

Other results are related to quantum geometric algebra and three or more qubits.

- Demonstrated that in general $\mathbf{Q}_q = \mathbf{G}_{n=2q} = \mathbf{H}_N$ where $q = n/2$ is the number of qubits in the quantum register, n is the number of orthonormal vectors, and $N=2^n$ is the size of overall generated vector space. There are $(43046721)^4 = 3.4 \times 10^{30}$ possible states or operators for \mathbf{Q}_3 , but very few of them were examined.
- Demonstrated that all combinations of $(\mathbf{S}_A \pm \mathbf{S}_B \pm \mathbf{S}_C \pm \dots \pm \mathbf{S}_Z)$ produce orthogonal phase encodings equivalent to the \mathbf{Q}_2 Bell and magic states, with $(q-1)^2$ variations for \mathbf{Q}_q . It still must be proven that these states do not have multiplicative inverses.
- Proved that the concurrent Hadamard transform $(\mathbf{S}_A + \mathbf{S}_B + \mathbf{S}_C + \dots + \mathbf{S}_Z)$ has the property $(\mathbf{S}_A + \mathbf{S}_B + \mathbf{S}_C + \dots + \mathbf{S}_Z)^2 = \mathbf{I}^-$ for any number of qubits q . The only valid states are those with a single qubit in the superposition state with all other qubit states in classical phases.
- Showed the Toffoli operator T is equivalent to the concurrent control-not and $T^2 = 1$
- Was unable to design the Fredkin operators in \mathbf{Q}_3 , which may require use of an inner product, a space larger than \mathbf{Q}_3 , or some other general linear operator mechanism.

These reversible gates are primarily concerned with implementing classical logic gates within quantum systems and so represent a somewhat different technological tack from the remainder of this work.

- Demonstrated that the idempotent operators P_k for \mathcal{Q}_1 where $(P_k)^2 = P_k$, when multiplied together for $\mathcal{Q}_{q=3,4}$ form cyclic operators $(R_k)^{2q} = R_k$ where $R_k = -P_k$. Also showed for $\mathcal{Q}_{q=3}$ that $(E_k)^{80} = 1$ but did not find the eigenvector power for $\mathcal{Q}_{q=4}$.

8.2 Conclusions

This dissertation focused primarily on the representation of, and tools for, a quantum geometric algebra \mathcal{Q}_n , where a single qubit is defined as \mathcal{Q}_1 . Applying Manthey's co-occurrence and co-exclusion perspective to this novel symmetric representation of qubits led to a consistent and meaningful interpretation when deriving new states and operators. All the relevant quantum computing concepts related to a qubit were described using these new geometric algebra definitions and framework. The new tools helped to deal automatically with the complexities of geometric algebra and enabled the automatic simplification and interpretation of complex geometric algebra expressions. The majority of the fundamental mathematical properties defining quantum computing were derived using the above infrastructure, and without using the traditional, unintuitive, complex-valued, matrix notation of Hilbert spaces. It appears that \mathcal{Q}_1 does faithfully represent the state, operators, properties, and measurement principles of a single qubit.

The behavior of multiple qubits was also explored and the geometric product was shown to be equivalent to the tensor product, thereby defining the quantum geometric algebra \mathcal{Q}_q as a quantum register of size q qubits. An ebit formed by entangling any number of qubits was simply derived using the recursive concurrent Hadamard operator and produced the appropriate Bell and magic states. In spite of the growth of the size of the \mathcal{Q}_q , the tools were able to perform some computations for >10 qubits and exhaustively prove some properties for two qubits. All of the well known operators for one and two qubits were derived.

8.3 Future Effort

The definition of a quantum geometric algebra for representing qubits and ebits, along with the corresponding prototyping tools, is obviously just the beginning. Many of the remaining mathematical holes in the formal definition in \mathcal{Q}_q need to be explored (by more formally trained mathematicians) such as determinants, multiplicative inverses of multivectors, Fredkin gate, Quantum Fourier Transforms, density operators, super-operators and probability densities, to name just a few. Also the next generation of tools could give more functionality and greater capacity for larger systems.

APPENDIX A

FREDKIN GATE USING GALOIS FIELD(2)

Fredkin gate is a universal reversible gate with three inputs [c b a] and three outputs [C B A] written as basis vectors using values {0,1}. When the control line c = 1 the other two inputs simply pass through to the corresponding output, so A = a and B = b. When the control line c = 0, then the two inputs cross to the other output, so A = b and B = a. The control line C always just passes through, so C = c. This gate is reversible where inputs and outputs can be swapped. Below is the Fredkin Gate truth table and gate symbol.

Table A.1 Fredkin Gate Logic

c b a	C B A	Observable?	<i>Logic relationships</i>	<i>Fredkin Gate Symbol</i>
0 0 0	0 0 0	same	AND/OR	
0 0 1	0 1 0	visible	XOR, AND/OR	
0 1 0	0 0 1	visible	NOT, XOR	
0 1 1	0 1 1	same	same	
1 0 0	1 0 0	same	same	
1 0 1	1 0 1	same	XOR	
1 1 0	1 1 0	same	NOT, XOR, AND/OR	
1 1 1	1 1 1	same	AND/OR	

A Fredkin gate is universal and can emulate any logic gates using the following rules.

When a = 0, b = 1 then A = !c and B = c, which is a NOT/DUP gate.

When c = b then A = a AND b, and B = a OR b which is an AND/OR gate.

When a = !b then A = a XOR b, and B = a XNOR b which is an XOR/XNOR gate.

Matrix representation of Fredkin Gate using Galois Fields

Finally enough groundwork is in place to write the definitions of a reversible gate using a matrix notation. This paper uses the mathematical notion where square braces represent vectors and matrices. Below is the derivation of matrix notation for a Fredkin gate. The binary values of {0, 1} are only allowed values.

When the control line $c = 1$, then the transformation is just the identity matrix as below:

$$[C \ B \ A] = [c \ b \ a] * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (\text{A.1})$$

When the control line $c = 0$, then the transformation is the "a" swap "b" matrix as below:

$$[C \ B \ A] = [c \ b \ a] * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad (\text{A.2})$$

Combining these two results produces the following operator written as a conditional result.

Notice how even at this early stage, the conditional terms based on "c" inside the matrix makes questionable the goal of a linear result.

$$[C \ B \ A] = [c \ b \ a] * \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & !c \\ 0 & !c & c \end{bmatrix} \quad (\text{A.3})$$

Now remove the constant columns and rows due to $C = c$ and rewrite to get:

$$[B \ A] = [b \ a] * \left(c * \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + !c * \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right) \quad (\text{A.4})$$

In GF(2) due to XOR nature of addition, can rewrite using !c = c + 1:

$$[B \ A] = [b \ a] * \left(c * \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + c * \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} + 1 * \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right) \quad (\text{A.5})$$

Now combine for the common product term "c":

$$[B \ A] = [b \ a] * \left(c * \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right) \quad (\text{A.6})$$

Then multiply completely out to show the final formula in linear matrix notation:

$$[B \ A] = [bc \ ac] * \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + [b \ a] * \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad (\text{where } bc = b * c = b * c) \quad (\text{A.7})$$

Notice how the original input basis dimensions of "a" and "b" have increased to include some new input basis terms formed by their product with input term "c". Combining this entire basis set into a single basis vector produces the following matrix operation for the two switched outputs of the Fredkin gate.

$$[B \ A] = [bc \ ac \ b \ a] * \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \quad (\text{A.8})$$

Now by including the original term C = c the final matrix notation is produced with all the additional basis terms, which are each products of some combinations of the original input basis terms.

$$[C \ B \ A] = [bc \ ac \ c \ b \ a]^* \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad (\text{A.9})$$

These additional terms are required in order to express Fredkin gates in a linear matrix format using GF(2). As will be shown later in general all input terms and all product combinations of input basis (plus the number 1) are required to be included in the new basis set for expressing any Boolean equation as reversible universal gates (such as Fredkin or Toffoli gates) in matrix notation. This similar result will resurface again later using Geometric Algebra, so this exercise is important to introduce in the simpler GF(2).

The individual output terms from equation (A.9) can be written out using Galois field notation:

$$A = b c + a c + b$$

$$B = b c + a c + a \quad (\text{A.10})$$

$$C = c$$

Now the usual Boolean logic operations can be double checked using the symmetrical equations (A.10). The following result shows for primitive Boolean expressions that the increase in number of dimension can be proportional to number of logic “AND” plus number of logic “OR” terms in the overall equation. Both the individual equations and matrix format are shown for each logic operation.

For NOT gate where $b = 1$ and $a = 0$ are substituted into equations (A.10) then:

$$\begin{aligned} A &= 1*c + 0*c + 1 = c + 1 = \text{NOT } c \\ B &= 1*c + 0*c + 0 = c = \text{DUP } c \end{aligned} \tag{A.11}$$

$$[C \ B \ A] = [1 \ c \ b \ a] * \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = [c \ b] * \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \text{ (where } b = 1)$$

For AND/OR gate where $c = b$ then: (with $b*b = b$ and $b+b = 0$)

$$\begin{aligned} A &= b*b + a*b + b = b + b + a*b = a*b = a \text{ AND } b \quad (\text{depends on } a*b) \\ B &= b*b + a*b + a = a + b + a*b = a \text{ OR } b \quad (\text{depends on } a*b) \end{aligned} \tag{A.12}$$

$$[C \ B \ A] = [ab \ b \ a] * \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

For XOR gate where $a = !b = b + 1$ then: [rewrite $b*c + b = b*(c+1) = b*!c$]

(the $a = !b$ is created with an extra NOT gate)

$$\begin{aligned} A &= b*c + !b*c + b = b*!c + !b*c = b + c = b \text{ XOR } c \\ B &= b*c + !b*c + !b = b*!c + !b*c + 1 = b + c + 1 = b \text{ XNOR } c \end{aligned} \tag{A.13}$$

$$[C \ B \ A] = [1 \ c \ b] * \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \quad (\text{matrix represents two gates})$$

This result shows that logic "AND" and logic "OR" both depend on the equivalent local product of "a b" as an addition basis vector to represent arbitrary Boolean equations in linear matrix format. This property is related to the property of reversibility, since "AND" and

"OR" throw away information about the inputs unless additional bits are carried through to other outputs of the computation. These new basis terms required for matrix format, are truly linearly independent terms for these operators, otherwise the overall result would not be linear for "AND" and "OR" gates.

It turns out the "a b" product term is just another way to look at the conditional control variable "c" in equation (A.1). This is understandable in vector terms if the AND gate case is analyzed in equations (A.12). The "a b" product term is required to make the four input states be independent of the other states {0, a, b}. This is true because the "a b" product is dependent on **both** inputs a and b, so can not be linearly independent on both simultaneously. By thinking about this geometrically, then the number of dimensions must be expanded to deal linearly with this codependency.

APPENDIX B

PERL SOURCE CODE FOR GA.PL TOOL

```
#!/usr/bin/perl
#!/usr/local/bin/perl
#usage: ga.pl <table|zero|vector|inner|outer> <expression>
#Geometric algebra parsing, normalization, products and simplification
#routines supporting non-commutative products where  $a*a = 1$  and  $a*b=-b*a$ 
#and mod 3 arithmetic with  $2=-1$ 

@myargs = @ARGV;

$hush = 0;
$verbose = 0; #for debugging only
$nosimplify = 0;
$evaluate_table = 0;
$printfunction = 0;
$showzeros = 0;
$innerproduct = 0;
$outterproduct = 1; #default is really the geom prod unless next flag set
$term_squared_zero = 0; #test since out prod  $a^a=0$  but  $a^a^b$  is still = b
$term_squared=1; #controls if  $a*a=1$  (default) or  $a*a=-1$  (using minus flag)
$time_basis = ""; # indicate time=token variable

%dimnames = ();
%all_terms = ();
$use_all_terms;
$right_hand_rule = 0; #don't set this unless experimentation

$first_term = "";
$last_term = "";

while ($arg = $myargs[0]) { #must put all control parameters first
  if ($arg =~ /simplif/i) {
    $nosimplify = "ON";
    print "ENABLED parameter: no simplification\n" if $verbose;
    shift @myargs;
    next;
  } elsif ($arg =~ /table/i) {
    $evaluate_table = "ON";
    print "ENABLED parameter: table\n" if $verbose;
    shift @myargs;
    next;
  } elsif ($arg =~ /verbose/i) {
    $verbose = "ON";
    print "ENABLED parameter: verbose\n" if $verbose;
    shift @myargs;
    next;
  }
}
```

```

} elseif ($arg =~ /function/i) {
    $evaluate_table = "ON";
    $printfunction = "ON";
    print "ENABLED parameters: table and show function\n" if $verbose;
    shift @myargs;
    next;
} elseif ($arg =~ /zero/i) {
    $evaluate_table = "ON";
    $showzeros = "ON";
    print "ENABLED parameters: table and show zeros\n" if $verbose;
    shift @myargs;
    next;
} elseif ($arg =~ /vector/i) {
    $evaluate_table = "ON";
    $vector_result = "ON";
    print "ENABLED parameters: zeros & show vector result\n" if $verbose;
    shift @myargs;
    next;
} elseif ($arg =~ /quiet/i) {
    $hush = "ON";
    print "ENABLED parameter: quiet\n" if $verbose;
    shift @myargs;
    next;
} elseif ($arg =~ /left/i) {
    $right_hand_rule = 0;
    print "DISABLED parameter: right\n" if $verbose;
    shift @myargs;
    next;
} elseif ($arg =~ /right/i) {
    $right_hand_rule = 1;
    print "ENABLED parameter: right\n" if $verbose;
    shift @myargs;
    next;
} elseif ($arg =~ /minus/i) {
    $term_squared = -1;
    print "ENABLED parameter: minus\n" if $verbose;
    shift @myargs;
    next;
} elseif ($arg =~ /all/i) {
    $use_all_terms = 1;
    print "ENABLED parameter: all\n" if $verbose;
    shift @myargs;
    next;
} elseif ($arg =~ /outer/i) {
    $term_squared_zero = 1; #force computation of standalone outer product
    $outerproduct = 1;
    $innerproduct = 0;    print "ENABLED parameter: outer\n" if $verbose;
    shift @myargs;
    next;
} elseif ($arg =~ /inner/i) {
    $outerproduct = 0;
    $innerproduct = 1;
    print "ENABLED parameter: inner\n" if $verbose;
    shift @myargs;
    next;
} elseif ($arg =~ /geom/i) {
    $term_squared_zero = 1; #forces GP as sum of real inner & outer prod

```

```

    $outerproduct = 1;
    $innerproduct = 1;
    print "ENABLED parameter: geometric\n" if $verbose;
    shift @myargs;
    next;
  } elsif (($value) = ($arg =~ /time=(.+)/i)) {
    $time_basis = $value;
    print "ENABLED parameter: time=$value\n";
    shift @myargs;
    next;
  } else {
    last;
  }
}
if (scalar @myargs == 0) {
  #print "Please type in expression:\n";
  $inputterm = <STDIN>; #if no args passed then read from standard input
  chomp $inputterm;
  @myargs = ($inputterm);
  #print "FOUND =$inputterm=\n";
}

$sum_resultsp = scalar(@myargs) > 1;
$sum_results = "";
#default table input terms if expression simplifies to constant.
map { &all_terms($_) } @myargs;
if ($right_hand_rule){
  foreach $term (&sort_terms(keys %all_terms)) {
    $first_term = $term unless $first_term;
    $last_term = $term; #leaves last value set in variable
  }
}
#print "First=$first_term and Last=$last_term\n" $if $verbose;

print "Input expression is ", join(" + ", @myargs), "\n" unless $hush;
#all args are simplified and then summed together
while ($eqn = shift @myargs) {
  @prod_terms = &parse_products($eqn);
  if (scalar @prod_terms > 1) {
    $results = &tensor_products(@prod_terms);
    #print "Tensor Product of $eqn is:\n $results\n";
  } else {
    $results = join(" ", &simple_equ(&parse_equ($prod_terms[0])));
    #print "Eqn \"$prod_terms[0]\" in normalized form is:\n $results\n";
  }
  print "$results\n" unless $evaluate_table || $sum_resultsp;
  if ($sum_resultsp) {
    $sum_results .= " " . $results;
  } else {
    &evaluate_table($results);
  }
}
#then simply sum of separate partial results and print final results
if ($sum_resultsp) {
  $combined = join(" ", &simple_equ(&parse_equ($sum_results)));
  print "$combined\n" unless $evaluate_table;
}

```

```

    &evaluate_table($combined);
}

sub all_terms {
    my ($string) = @_;
    my ($term);
    $string =~ s/\+/ /g;
    $string =~ s/\-/ /g;
    $string =~ s\/\(/ /g;
    $string =~ s\/\)/ /g;
    $string =~ s/^ +//g;
    $string =~ s/ +$//g;
    foreach $term (grep { /\D/ } split(/ +/, $string)) {
        $all_terms{$term} = "INPUT";
    }
}

sub parse_equ { #first split up terms, then normalize and sort
    my($equstring) = @_;
    my(@rawterms, @terms);
    $equstring =~ s/^ +//;
    $equstring =~ s/ +$//;
    #add ":" before + or - to aid in splitting and force space after
    $equstring =~ s/([+-])/\:$1 /g;
    $equstring =~ s/ +/ /g; #remove redundant spaces
    (@rawterms) = split /:/, $equstring;
    #remove empty term caused by leading + or -
    shift(@rawterms) unless $rawterms[0];
    (@rawterms) = map { &normalize_token_order($_) } @rawterms;
    #sort terms based on # of terms & then alphabetic sort while ignoring sign
    return &sort_terms(@rawterms);
}

#WARNING: parsing does not take precedence of products over addition
#so a + b (d + e) == (a + b)(d + e)
sub parse_products { #parses products based on parens
    my($eqnstring) = @_;
    my(@prodterms, $newterm, $begin, $rest);
    #look for parens to indicate product terms are present
    if ($eqnstring =~ /[()]/ ) {
        #find terms within parenthesis and remove parens
        while (($newterm) = $eqnstring =~ /\((.+?)\)/ ){
            $begin = $`;
            $rest = $';
            # allow terms can be seperated by parens but not enclosed
            push(@prodterms, $begin) if $begin =~ /\w+/;
            push(@prodterms, $newterm);
            $eqnstring = $rest;
        }
        #allow nonparens at end
        push(@prodterms, $eqnstring) if $eqnstring =~ /\w+/;
        return @prodterms;
    } else {
        return ($eqnstring);
    }
}

sub tensor_products { #makes one or more tensor/ combination product

```

```

my(@productterms) = @_;
#print "here are product terms =", join("=", @productterms), "\n";
my($lastresult) = shift @productterms;
my(@resulterms, $eqnstring);
while ($eqnstring = shift @productterms) {
    (@resulterms) = &product_combinations ($lastresult, $eqnstring);
    $lastresult = join(" ", @resulterms);
}
return $lastresult
}

#expand product combinations using appropriate inner/outer product
sub product_combinations { #makes one tensor/combination product
    my($string1,$string2) = @_;
    my(@list1, @list2, $item1, $item2);
    my($sign, $product, $stop, @tokens);
    my($sign1, $sign2, @tokens1, @tokens2, @result);
    (@list1) = &parse_equ($string1);
    (@list2) = &parse_equ($string2);
    foreach $item1 (@list1) { #all combinations of list items
        foreach $item2 (@list2) {
            #allow inner, outer or both for geometric product
            ($sign1, @tokens1) = &sign_and_terms($item1);
            ($sign2, @tokens2) = &sign_and_terms($item2);
            if ($outerproduct) { #then concatenate tokens for outer product
                $sign = &sign_product($sign1,$sign2);
                @tokens = grep { $_ ne "1" } (@tokens1, @tokens2);
                $product = &normalize_token_order("$sign @tokens");
                push(@result, $product);
            }
            if ($innerproduct) {
                if (scalar @tokens1 > scalar @tokens2) { #shortest list in @t1
                    @temp = @tokens1;
                    @tokens1 = @tokens2;
                    @tokens2 = @temp;
                }
                #normalize order for each else final sign
                ($sign1, @tokens1) = &sign_and_terms(
                    &normalize_token_order("$sign1 @tokens1"));
                ($sign2, @tokens2) = &sign_and_terms(
                    &normalize_token_order("$sign2 @tokens2"));
                $sign = &sign_product($sign1,$sign2);
                #iterate from end while reducing grade
                foreach $vector (reverse @tokens1){
                    ($stop, $sign, @tokens2) =
                        &dot_product($sign, $vector, @tokens2);
                    last if $stop;
                }
                if (! $stop) {
                    $product = "$sign @tokens2";
                    push(@result, $product);
                }
            }
        }
    }
}
$bigresult = join(" ",@result);
#print "Terms are $bigresult\n";

```

```

    return &simply_equ(&parse_equ($bigresult));
    #return &parse_equ($bigresult);
}

sub dot_product {
    my($sign, $vector, @tokens) = @_;
    my($swaps) = 0;
    my($token, @scanned);
    #print "Dot product of \"$sign\" \"$vector\" for \"@tokens\"\\n";
    #if scalar (or empty) then result is 0
    return ("STOP", $sign) if ($vector eq "1");
    while ($token = shift @tokens) {
        if ($vector eq $token) {
            $flipsign = &oddp($swaps) ? "-" : "+";
            $sign = &sign_product($sign, $flipsign);
            return ("", $sign, @scanned, @tokens); #exclude this token
        }
        $swaps = $swaps + 1;
        push(@scanned, $token);
    }
    return ("STOP", $sign); #else not found so result is 0
}

sub sign_product {
    my($sign1, $sign2) = @_;
    my($newsign) = "+";
    if (($sign1 eq "-") || ($sign2 eq "-")) {
        $newsign = "-" unless ($sign1 eq $sign2);
    }
    return $newsign;
}

sub simply_equ { #this implements mod 3 arithmetic where 2 => -1;
    my(@terms) = @_;
    if ($nosimplify) {
        return &sort_terms(@terms);
    } else {
        my($term, $count, @tokens, $tokens, $final_count);
        my(@final_result, %sameterms);
        foreach $term (@terms){
            ($count, @tokens) = &number_and_terms($term);
            map { $dimnames{$_} = "YES" } @tokens;
            $tokens = join " ", @tokens;
            #print "Processing term =$term= into $count and $tokens\\n";
            $sameterms{$tokens} = $sameterms{$tokens} + $count; #incr or decr
        }
        foreach $term (keys %sameterms){
            $final_count = $sameterms{$term} % 3;
            if ($final_count) {
                push(@final_result, (($final_count == 1) ? "+" : "-") . " " . $term);
            }
        }
        return (0) unless @final_result;
        return &sort_terms(@final_result);
    }
}

$entries_printed = 0; #used to suppress multiple unnecessary dividers
$stars = "*****";

```

```

sub evaluate_table {
    my($equation) = @_;
    #only non-numeric variable names
    my(@dnames) = sort { $a cmp $b } (grep { /\D/ } keys %dimnames);
    if(! $hush) && $evaluate_table) {
        print "Using all INPUT KEYS of: ", join " ", (sort keys %all_terms),
            "\n" if ($use_all_terms || ! @dnames);
    }
    #use all terms from input expression if constant output expression.
    @dnames = sort keys %all_terms if ($use_all_terms || ! @dnames);
    my($dcount) = scalar @dnames;
    my($maxcount) = 2 ** $dcount;
    my($diterate, @sum_terms, $nonzero_rows, $plus_rows);
#print "FOUND unique terms @dnames from =",join("=",keys %dimnames),"\n";
    if ($evaluate_table && $dcount) {
        #disables right hand rule fixup for parse_equ, since already done
        #right_hand_rule = 0;
        @sum_terms = &build_exp($equation, @dnames);
        if ($vector_result) {
            print "[";
        } else {
            print "Logic table with $maxcount entries with columns INPUTS ",
                "| PRODUCTS | OUTPUT\n" if $verbose;
            print "INPUTS: @dnames | @sum_terms | OUTPUT\n";
            $entries_printed = 1; #controls row divider printing
        }
        while ($maxcount > $diterate ) {
            #print "ROW ", $diterate || "0", "\n";
            if ($value = &evaluate_row($diterate, $dcount)) {
                $nonzero_rows++;
                $plus_rows++ if ($value eq "+");
            }
            $diterate++;
        }
        if ($vector_result) {
            print "]\n";
        } else {
            print "$stars\n" if $entries_printed;
            print "Row counts for outputs of ZERO=", $maxcount - $nonzero_rows,
                ", PLUS=", $plus_rows || "0",
                ", MINUS=", ($nonzero_rows - $plus_rows) || "0",
                " for TOTAL=$maxcount rows.\n\n\n" if $nonzero_rows;
        }
    }
}

sub evaluate_row {
    my($rowcount, $width) = @_;
    my($original) = $rowcount || "0";
    my($result, @signs, @inputs);
    @inputs = &binary_explode($rowcount, $width); #returns values of 0,1,2
    @signs = &sign_conversion(@inputs);
    ($result,@products) = map {&sign_conversion($_)} &evaluate_exp(@inputs);
    if ($vector_result) {
        my($spacer) = "";
        if ($rowcount) {$spacer = " " if ($rowcount % 8) == 0;}
    }
}

```

```

    print "$spacer$result";
    return $result;
}
if (($original % 4) == 0) { #print dividers every four entries
    print "$stars\n" if $entries_printed;
    $entries_printed = 0; #reset count if just printed divider.
}
$original = "0$original" if $original < 10;
if ($showzeros || ($result ne 0)) {
    print "ROW $original: @signs | @products | $result\n";
    $entries_printed++; #increment enables printing of trailing divider
}
return $result;
}

sub build_exp { #build custom func on the fly for &evaluate_exp using eval
    my($equation, @dnames) = @_;
    my(@sum_terms);
    @sum_terms = &parse_equ($equation);
    #only non-numeric variable names
    $inputnames = join(",", map { "\$" . $_ } @dnames);
    $prodexpression = join(",", map { &build_prodterm($_) } @sum_terms);
    $function = <<EOFUNCTION;
sub evaluate_exp {
    my ($inputnames) = \@_;
    my (\@products) = ($prodexpression);
    return (&cadd(\@products), \@products);
}
EOFUNCTION
    print "BUILDING function:\n$function" if $printfunction;
    eval $function; ##redefine the evaluation function using eval
    return @sum_terms;
}

sub evaluate_exp { #default dummy func will be redefined by &build_exp
    #my($a0, $a1, $x0, $x1) = @_;
    #my(@products) = (&cmult("-", $a0, $x0), &cmult("-", $a0, $x1),
    #                &cmult("-", $a1, $x0), &cmult("-", $a1, $x1));
    #return &cadd(@products);
    return 1;
}

sub build_prodterm {
    my($arg) = @_;
    my($sign, @terms, @vars);
    ($sign, @terms) = &sign_and_terms($arg);
    #if non-numeric then create var, else leave as number
    @vars = map { /\D/ ? ("\$" . $_) : $_ } @terms;
    return "&cmult(\"$sign\", " . join(",", @vars) . ")";
}

#mod 3 mult used by evaluate_table and evaluate_row and evaluate_exp
sub cmult {
    my($msign, $first, @mterms) = @_;
    my($result) = $first;
    my($term);
    print "WARNING: cmult passed $result\n" if $result < 0 || $result > 3;

```

```

foreach $term (@mterms) {
    print "WARNING: cmult passed $term\n" if $term < 0 || $term > 3;
    $result = ($result * $term) % 3; #mod three multiplication
}
if ($msign eq "-") {
    $result=(($result == 1)? 2 : 1; #invert swaps between values 1 and 2=-1
}
#print "TRACE product of $msign $first @mterms gives $result\n";
return $result;
}
#mod 3 addition used by evaluate_table and evaluate_row and evaluate_exp
sub cadd {
    my(@addterms) = @_;
    my($result);
    my($term);
    foreach $term (@addterms) {
        print "WARNING: cadd passed $term\n" if $term < 0 || $term > 3;
        $result = ($result + $term) % 3; #mod three addition
    }
    #print "TRACE add of @addterms gives $result\n";
    return $result;
}
#converts +1 => + and 2 => - and leaves others alone for debug
sub sign_conversion {
    my(@terms) = @_;
    return map {if ($_ == 1){ "+" } elsif ($_ == 2) { "-" } else {$_ }}
                @terms;
}
sub binary_explode {
    my($integer, $width) = @_;
    my(@results);
    while ($width) {
        #values 0 and 1 converted to -1=2 and 1
        unshift (@results,($integer & 1) || 2);
        $integer = $integer >> 1;
        $width--;
    }
    return @results;
}
sub sort_terms {
    my(@terms) = @_;
    if ($nosimplify) {
        return @terms;
    } else {
        #fewest number of product terms as first sort criteria
        return sort { ($a =~ s/ / /g) <=> ($b =~ s/ / /g) ||
            #then alphabetically next criteria
            (substr($a,2) cmp substr($b,2)) ||
            #then sign as last criteria
            (substr($a,0,1) cmp substr($b,0,1))
        } @terms;
    }
}
sub normalize_token_order {
    my($tokenstring) = @_;

```

```

my($sign, @tokens, @results);
($sign, @tokens) = &number_and_terms($tokenstring);
my($count) = scalar(@tokens) - 1; #upper offset into array (zero based)
my($upper, $this, $next, $result, %termcount);
#print "Count = $count for @tokens\n";
if ($count) {
    #this is a bubble sort to guarantee adjacent swaps for non-assoc
    foreach $upper ($count .. 1) {
        foreach $this (0 .. $upper-1) {
            $next = $this + 1;
            $result = $tokens[$this] cmp $tokens[$next];
            #print "With sign=$sign comparing $tokens[$this] with ",
            #      "$tokens[$next] giving result $result";
            if ($result == 1){ #then swap 2 values so max is in $tokens[$next]
                #print " and toggling sign and values";
                my($min) = $tokens[$next];
                my($max) = $tokens[$this];
                $tokens[$this] = $min;
                $tokens[$next] = $max;
                if ($sign == 1){$sign = -1} else {$sign = 1}; #and swap sign value
            }
            #print ".\n";
        }
    }
}
#after sorting now can safely remove duplicate terms a*a=1
#without sign change (or -1 w/sign adjust)
%termcount = ();
#precount identical tokens & then include one copy for odd counts
grep { $termcount{$_}++; 0} @tokens;
#flip sign due to right hand rule. but not right yet so remove
# if ($right_hand_rule && &oddp($termcount{$first_term}) &&
#   &oddp($termcount{$last_term})) {
#   #print "Flipping Sign=$sign due to Right hand rule for @tokens\n";
#   $sign = $sign * -1;
# }
my ($result_is_zero) = 0;
if ($term_squared_zero) {
    @results = grep { $result_is_zero = 1 if $termcount{$_} > 1;
                    $termcount{$_} == 1 } @tokens;
    return 0 if $result_is_zero && (0 == scalar @results);
    if ($sign == 1) { $sign = "+" } else { $sign = "-" };
    return "@sign @results";
} else {
    @results =
        grep { my($odd) = &oddp($termcount{$_});
              $sign = &signadj($sign,$_, $termcount{$_});
              $termcount{$_} = 0;
              $odd}
            @tokens;
    @results = (1) unless @results;
    if ($sign == 1) { $sign = "+" } else { $sign = "-" };
    return "$sign @results";
}
}

sub signadj { #adjusts the sign when removing duplicate terms

```

```

my($oldsign, $token, $count) = @_;
my($local_term_squared) = $term_squared;
if ($time_basis) {
    if ($time_basis eq $token) {
        #then set as time like according to Univ Cambridge GA tutorial
        $local_term_squared = 1;
    } else {
        $local_term_squared = -1; #but set all others as space like
    }
}
if ($count == 1) {
    #default case when sorting but no dupls removed, so don't touch sign
    return $oldsign;
} elsif ($local_term_squared == 1) {
    return $oldsign;
} elsif ($local_term_squared == -1) {
    #divide count by two and flip sign that number of times
    $count = $count >> 1;
    while ($count) {
        $count--;
        $oldsign = $oldsign * -1;
    }
    return $oldsign;
} else {
    die "ERROR: invalid value for term_squared=$local_term_squared\n";
}
}

sub oddp { my($numb) = @_; return $numb % 2;}

sub number_and_terms {
    my($tokenstring) = @_;
    my($sign, @tokens);
    ($sign, @tokens) = split(/ +/, $tokenstring);
    if ($sign eq "+") { $sign = 1;
    } elsif ($sign eq "-") { $sign = -1;
    } else {unshift (@tokens, $sign); $sign = 1; #if no sign then assume pos
    }
    return ($sign, @tokens);
}

sub sign_and_terms {
    my($tokenstring) = @_;
    my($sign, @tokens);
    ($sign, @tokens) = split(/ +/, $tokenstring);
    if (($sign eq "+") || ($sign eq "-")) {
        $sign = $sign;
    } else {
        unshift (@tokens, $sign);
        $sign = "+"; #if no sign then assume positive
    }
    return ($sign, @tokens);
}

#end of file

```

APPENDIX C

PERL SOURCE CODE FOR GANDG.PL TOOL

This code is relatively short because it depends on shared file *permutation.pl*, which the source code can be found in Appendix F.

```
#!/usr/bin/perl
#!/usr/local/bin/perl

#Geometric algebra AND generator based on dimensions and permutation.pl

# usage: gandg.pl <dims>
#example: gandg.pl "a,b,c"
#example: gandg.pl "a,b,c,d,e"

require 'permutation.pl';

#this order is maintained through out
@dimterms = split /[;, ]/, shift @ARGV;

@sorterms = sort {$a cmp $b} @dimterms;
if (join(" ",@sorterms) ne join(" ", @dimterms)) {
    die("ERROR: Since input terms are not in expected sort order: " .
        "@sorterms\n");
}

print join " ", &generate_equ_for_maxstate(@dimterms), "\n\n";

#end of file
```

APPENDIX D

PERL SOURCE CODE FOR GAG.PL TOOL

```
#!/usr/bin/perl
#!/usr/local/bin/perl

#Geometric algebra generator for equations based on state numerical values
#Default is to set states to + but can all specify - result. All others
#are unassigned (left at 0).

# usage: gag.pl <dims> <stateset1> <stateset2> ... etc
#example: gag.pl "a,b,c" "4,5,-6 7"
#where statesets are lexicographical state number based on dimterm order
#example: gag.pl "a,b,c" "+,6" "-7"
#vector mode starts and ends with square brackets
#example: gag.pl "a,b,c" "[+--+ +-0+]"

require 'permutation.pl';
$verbose = 0; #set =1 for verbose debug printing
@dimterms = split /[ ,; ]/, shift @ARGV; #this order is maintained
through out
@revterms = reverse @dimterms;
$dims = scalar @dimterms;
$maxstate = (2 ** $dims) - 1;
$odd_arity = $dims % 2; #based on number of dims;

@sorterms = sort {$a cmp $b} @dimterms;
if (join(" ",@sorterms) ne join(" ", @dimterms)) {
    die("ERROR: Since input terms are not in expected sort order: " .
        "@sorterms\n");
}

@andterms = &generate_equ_for_maxstate(@dimterms);
%termhash;
%stateequations;
$stablecontrols = "quiet"; #suppresses the input eqn printing (can be huge)
my($laststate) = -1; #used in vector input mode

while (@ARGV) {
    $current_states = shift @ARGV;
    print "BEFORE $current_states\n" if $verbose;
    if ($vectormode = $current_states =~ /\[([.+])\]/) {
        $current_states = $1;
        die "ERROR: bad chars in vector mode only allow \"+-0 \"\n"
            if $current_states =~ /^[^+-0 ]/;
        $current_states =~ s/([+-0])/ $1 /g;#splitup tokens w/vector notation
        $current_states =~ s/ / /g; #remove unwanted
        $current_states =~ s/ $//g;
    }
```

```

}
if ($current_states =~ /verbose/) { $verbose = 1; }
#can pass thru table controls here.
if ($current_states =~ /table|zero|vector/) {
    $stablecontrols .= " " if $stablecontrols;
    $stablecontrols .= $current_states;
    next;
}
print "AFTER $current_states \n" if $verbose;
my($assignplus) = 1; #default sign value
@states = split(/[,; ]+/, $current_states);
print "split states ARE [@states]\n" if $verbose;
foreach $state (@states) {
    #manage the sign as either seperate token
    if ($state eq "+") {
        $assignplus = 1;
        $state = $laststate + 1;
    } elsif ($state eq "-") {
        $assignplus = 0;
        $state = $laststate + 1;
    } elsif ($state eq "0" && $vectormode){#skip state only in vector mode
        $laststate = $laststate + 1;
        next;
    }
    #or manage the sign as part of other number token
    } elsif (substr($state,0,1) eq "+") {
        $assignplus = 1;
        $state = substr($state,1) || ($laststate + 1);
    } elsif (substr($state,0,1) eq "-") {
        $assignplus = 0;
        $state = substr($state,1) || ($laststate + 1);
    }
    }
    if ($verbose){
    print "WARNING: State=$state is not in expected order ",
        $laststate + 1, "\n" if $state != ($laststate + 1);
    }
    $laststate = $state;
    die "WARNING: Bad state $_ exceeds max of $maxstate\n"
        unless (abs($state) <= $maxstate);
    %termhash = &binary_explode_hash($state,@revterms);# always overwrite
    $result = join(" ", map { &adjustterm($_, $assignplus) } @andterms);
    $stateequations{$state} = $result;
    print "State $state with sign ", $assignplus?"+": "-",
        " has equation $result\n" if $verbose;
}
}
}

$all_terms = join(" ", map({ "\"($_)\\"" } values %stateequations));
#use remote shell call to simplify equations.
print `./ga.pl $stablecontrols $all_terms`;

##*****
##subroutines are below here

#substitutes in for each vector the new sign value based on bin row count.
sub adjustterm {
    my ($term, $positive) = @_;
    my ($item, $sign, $tokenstring);

```

```

($sign, $tokenstring) = $term =~ /([+-]) (.+)/;
foreach $item (@dimterms){
  if ($termhash{$item} eq "-") {#toggle sign if token in string
    $sign = ($sign eq "+") ? "-" : "+" if $tokenstring =~ /$item/;
  }
}
#invert one last time if not positive
$sign = ($sign eq "+") ? "-" : "+" unless $positive;
return "$sign $tokenstring";
}

#determines which vectors need to be inverted based on integer row count
sub binary_explode_hash {
  my($integer, @revstates) = @_;
  my(%results);
  while (@revstates) {
    $results{shift @revstates} = ($integer & 1) ? "+" : "-";
    $integer = $integer >> 1;
  }
  return %results;
}

#end of file

```

APPENDIX E

PERL SOURCE CODE FOR GASOLVE.PL TOOL

```
#!/usr/bin/perl
#!/usr/local/bin/perl

#Geometric algebra solver that iterates through all possible multivectors

#  usage: gasolve.pl <dims> <eXpression> <result1><result2> ... etc

#example: gasolve.pl "a0,a1,b0,b1" "(a0 a1 + b0 b1) (X)" "1"
#tries to find the multiplicative inverse of (a0 a1 + b0 b1)

#example: gasolve.pl "a0,a1,b0,b1" "(X)(X)" "(a0 - a1)"
#tries to find the square root of cnot which should be chad

require 'permutation.pl';
require 'galib.pl'; #code identical to ga.pl except is imported

$bindir = ".";
$varname = "X"; #use capital X to indicate where to substitute

#this order is maintained through out
@dimterms = split /[ ,; ]/, shift @ARGV;
@revterms = reverse @dimterms;
$dims = scalar @dimterms;
$maxstate = (2 ** $dims) - 1;
$oodd_arity = $dims % 2; #based on number of dims;

@sorterms = sort { $a cmp $b } @dimterms;
if (join(" ",@sorterms) ne join(" ", @dimterms)) {
    die("ERROR: Since input terms are not in expected sort order: " .
        "@sorterms\n");
}

$initial_state = 0;
if (! ($ARGV[0] =~ /^[^+0]/)) { #if any chars other than +-0 then skip
    $initial_state = shift @ARGV;
    #print "STARING with state $initial_state\n";
}

$eqn = shift @ARGV;
$stablecontrols = "quiet"; #suppresses input eqn print which can be huge

if (grep /X/, @ARGV) {
    $X_in_answer = "TRUE";
    @answers = map {$_} @ARGV;
}
```

```

} else {
    @answers = map {&standardize_eqn($_)} @ARGV;
}

@andterms = map {substr($_,2)} &generate_equ_for_maxstate(@dimterms);

$max = $#andterms;
@trinary = map { ($initial_state ? substr($initial_state, $_, 1) : "0")
    || "0"} (0 .. $max);
print "Starting with state ", @trinary, " = ", &fetch_all_terms(), "\n"
    if $initial_state;

$totalcount = 0;
$matchcount = 0;
while ($next = &next_eqn_perm()) {
    my ($eqncopy) = $eqn;
    $totalcount++;
    $result = &standardize_eqn($eqncopy,$next);
    my (@custom_answers) = @answers;
    if ($X_in_answer) {
        @custom_answers = map {&standardize_eqn($_,$next)} @answers;
    }
    #print "Iteration \"$result\" for X = \"$next\" in $eqncopy matching
    #@custom_answers\n";
    @matches = grep {$_ eq $result} @custom_answers;
    if (@matches || 0 == @answers) {
        $matchcount++ if @matches;
        print "Found Match for X = $next in $eqn = $result\n";
    }
    if (($totalcount % 100000) == 0) {
        print "TRIED $totalcount: ", @trinary,
            " = $next and found $matchcount on " . `date`;
    }
}
print "Attempted $totalcount with $matchcount found.\n";

##*****
##subroutines are below here
sub next_eqn_perm {#get the next equation combinations by trinary counting
    foreach $index (0 .. $max) { #while in this loop trinary count
        my ($tristate) = $trinary[$index];
        if ($tristate eq "0") {
            $trinary[$index] = "-";
            last;
        } elsif ($tristate eq "-") {
            $trinary[$index] = "+";
            last;
        } elsif ($tristate eq "+") {
            $trinary[$index] = "0"; #this is carry into next bit
            # die "Max Carry hit for $totalcount\n" if ($index == $max);
        } else {
            die "Illegal value found in index=$index\n";
        }
    }
    #then collect all non-zero terms
    return join "", map {&fetch_term($_)} (0 .. $#trinary)
}

```

```

sub fetch_all_terms { #then collect all non-zero terms
    return join "", map {&fetch_term($_)} (0 .. $#trinary)
}

#conditionally collects the Ith term with the appropriate sign
sub fetch_term {
    my ($index)= @_;
    my ($andterm) = $andterms[$index];
    my ($prefix) = $trinary[$index];
    my ($endfix) = ($index == $max) ? "" : " ";
    if ($prefix eq "0") {
        return "";
    } else {
        return $prefix . " " . $andterm . $endfix; #for + and - terms
    }
}

sub standardize_eqn {
    my($equation, $customize) = @_;
    if ($customize) { #customize result each time equation is passed.
        $equation =~ s/$varname/$customize/g;
    }
    # $equation = `$bindir\ga.pl quiet \"$equation\"`; chomp $equation;
    $equation = &ga_evaluate("quiet", $equation);
    return $equation;
}

#end of file

```

APPENDIX F

PERL SOURCE CODE FOR SHARED PERMUTATION.PL FILE

```
#!/usr/local/bin/perl
#For inputs (a,b,c,etc) produces the equivalent of (1+a)(1+b)(1+c)etc

sub generate_equ_for_maxstate { #only callable function in this file
    my (@terms) = @_; #ordered list of vector term names
    my ($usesign, $found, @allpermutations);
    my ($expect) = 2 ** (scalar(@terms)); #number of expected terms
    $usesign = (scalar(@terms) % 2) ? "-" : "+"; #odd_arity use - sign
    #print "TERMS are @terms\n";
    @allpermutations = map { &generate_permutation($_, $usesign, @terms) }
        (1 .. scalar(@terms));
    unshift(@allpermutations, "$usesign 1");
    $found = scalar(@allpermutations);
    print "WARNING: Expected $expect permutation count=$found for @terms\n"
        unless $expect == $found ;
    return @allpermutations;
}

sub generate_permutation {#really generating combinations not permutations
    my ($bycount, $sign, @theterms) = @_;
    my (@results);
    my ($maxgap, $gapindex, $gather, $term);
    #print "PERMUTATION: bycount=$bycount, sign=$sign, @theterms\n";
    $sign = $sign . " " if $sign =~ /[+-]/;
    if ($bycount == 1) {
        return map { $sign . $_ } @theterms;
    } elsif ($bycount > 2) {
        my($first, @rest, @subterms);
        $maxgap = scalar(@theterms) - $bycount + 1;
        while ($gapindex < $maxgap) {
            ($first, @rest) = @theterms;
            @subterms = &generate_permutation($bycount-1,"", @rest);
            #print "SUBTERMS for $bycount, $sign, @theterms were =",
            #      join("=", @subterms), "=\n";
            @results = (@results, map { "$sign$first $_" } @subterms);
            shift @theterms;
            $gapindex++;
        }
        return @results;
    } else {
        while (@theterms) {
            $gapindex = 0;
            $maxgap = scalar(@theterms) - $bycount + 1;
            #print "MAXGAP = $maxgap @theterms\n";
```

```

    while ($gapindex < $maxgap) {
        $term = $theterms[0];
        $gather = 1;
        while ($gather < $bycount) {
#print "COUNTS are max= $maxgap gap= $gapindex gather= $gather $term\n";
            $term .= " " . $theterms[$gapindex + $gather];
            $gather++;
        }
        unshift(@results, $sign . $term);
        $gapindex++;
    }
    shift @theterms; #shorten the iteration list/array
}
#print " produced @results\n";
return reverse @results;
}
}

1; #return true when used with require and end of file

```

APPENDIX G

PERL SOURCE CODE FOR SHARED GALIB.PL FILE

This file is a shared version of *ga.pl* from Appendix B, so only the top level functions are changed. This file only takes args passed thru function interface, and not thru UNIX pipes.

```
#!/usr/bin/perl
#!/usr/local/bin/perl
#Geometric algebra parsing, normalization, products and simplification
#routines supporting non-commutative products where  $a*a = 1$  and  $a*b=-b*a$ 
#and mod 3 arithmetic with  $2=-1$ 
#Only the top two functions are intended to be callable by other programs

sub init_ga_globals {
    my (@myargs) = @_ ;
    my ($arg);
    $hush = 0;
    $verbose = 0; #for debugging only
    $nosimplify = 0;
    $evaluate_table = 0;
    $printfunction = 0;
    $showzeros = 0;
    $outerproduct = 1; #set both for geometric product
    $innerproduct = 1;
    $term_squared=1; #controls if  $a*a=1$  (default) or  $a*a=-1$  (w/minus flag)
    $time_basis = ""; # indicate time=token variable
    %dimnames = ();
    %all_terms = ();
    $right_hand_rule = 0; #don't set this unless experimentation
    $first_term = "";
    $last_term = "";
    while ($arg = $myargs[0]) { #must put all control parameters first
        print "DEBUGGING parsing of $arg\n" if $verbose;
        if ($arg =~ /simplif/i) {
            $nosimplify = "ON";
            print "ENABLED parameter: no simplification\n" if $verbose;
            shift @myargs;
            next;
        } elsif ($arg =~ /table/i) {
            $evaluate_table = "ON";
            print "ENABLED parameter: table\n" if $verbose;
            shift @myargs;
            next;
        } elsif ($arg =~ /function/i) {
            $evaluate_table = "ON";
            $printfunction = "ON";
        }
    }
}
```

```

    print "ENABLED parameters: table & show function\n" if $verbose;
    shift @myargs;
    next;
} elsif ($arg =~ /zero/i) {
    $evaluate_table = "ON";
    $showzeros = "ON";
    print "ENABLED parameters: table and show zeros\n" if $verbose;
    shift @myargs;
    next;
} elsif ($arg =~ /vector/i) {
    $evaluate_table = "ON";
    $vector_result = "ON";
    print "ENABLED params: zeros & show vector result\n" if $verbose;
    shift @myargs;
    next;
} elsif ($arg =~ /quiet/i) {
    $hush = "ON";
    print "ENABLED parameter: quiet\n" if $verbose;
    shift @myargs;
    next;
} elsif ($arg =~ /left/i) {
    $right_hand_rule = 0;
    print "DISABLED parameter: right\n" if $verbose;
    shift @myargs;
    next;
} elsif ($arg =~ /right/i) {
    $right_hand_rule = 1;
    print "ENABLED parameter: right\n" if $verbose;
    shift @myargs;
    next;
} elsif ($arg =~ /minus/i) {
    $term_squared = -1;
    print "ENABLED parameter: minus\n" if $verbose;
    shift @myargs;
    next;
} elsif ($arg =~ /all/i) {
    $use_all_terms = 1;
    print "ENABLED parameter: all\n" if $verbose;
    shift @myargs;
    next;
} elsif ($arg =~ /outer/i) {
    $outerproduct = 1;
    $innerproduct = 0;
    print "ENABLED parameter: outer\n" if $verbose;
    shift @myargs;
    next;
} elsif ($arg =~ /inner/i) {
    $outerproduct = 0;
    $innerproduct = 1;
    print "ENABLED parameter: inner\n" if $verbose;
    shift @myargs;
    next;
} elsif ($arg =~ /geom/i) {
    $outerproduct = 1;
    $innerproduct = 1;
    print "ENABLED parameter: geometric\n" if $verbose;
    shift @myargs;

```

```

        next;
    } elsif (($value) = ($arg =~ /time=(.+)/i)) {
        $time_basis = $value;
        print "ENABLED parameter: time=$value\n";
        shift @myargs;
        next;
    } else {
        return @myargs;
    }
}
return @myargs;
}
sub ga_evaluate { ##this is the main routine that all args are passed.
my(@myeqns);
my($eqn);
@myeqns = &init_ga_globals(@_);
$sum_resultsp = scalar(@myeqns) > 1;
$sum_results = "";
print "MYEQNS are @myeqns\n" if $verbose;
#default table input terms if expression simplifies to constant.
map { &all_terms($_) } @myeqns;
if ($right_hand_rule){
    foreach $term (&sort_terms(keys %all_terms)) {
        $first_term = $term unless $first_term;
        $last_term = $term; #leaves last value set in variable
    }
}
print "First=$first_term and Last=$last_term\n" if $verbose;
print "Input expression is ", join(" + ", @myeqns), "\n" unless $hush;
#all args are simplified and then summed together
while ($eqn = shift @myeqns) {
    @prod_terms = &parse_products($eqn);
    if (scalar @prod_terms > 1) {
        $results = &tensor_products(@prod_terms);
        #print "Tensor Product of $eqn is:\n $results\n";
    } else {
        $results = join(" ", &simple_equ(&parse_equ($prod_terms[0])));
        #print "Eqn \"$prod_terms[0]\" in normalized form is:\n $results\n";
    }
    return $results unless $evaluate_table || $sum_resultsp;
    if ($sum_resultsp) {
        $sum_results .= " " . $results;
    } else {
        &evaluate_table($results); #this is never executed
    }
}
#then simply sum of separate partial results and print final results
if ($sum_resultsp) {
    $combined = join(" ", &simple_equ(&parse_equ($sum_results)));
    return $combined unless $evaluate_table;
    &evaluate_table($combined); #this is never executed
}
}
#the remainder of the file is the same as ga.pl starting with definition
#of function all_terms()
1; #return true and also end of file

```

APPENDIX H

SUMMARY OF NOTATION AND BASES

Geometric algebra notation and properties with orthonormal vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$, etc: \mathbf{G}_n

Geometric algebra: $\mathbf{G}_2 = \text{span}\{\mathbf{a}, \mathbf{b}\}$, $\mathbf{G}_3 = \text{span}\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$, $\mathbf{G}_4 = \text{span}\{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$, etc

Co-occurrence: $\mathbf{a} + \mathbf{b}$ means \mathbf{a} concurrent with \mathbf{b} . $\mathbf{a} + \bar{\mathbf{a}} = 0$, where 0 means cannot occur.

Geometric product $\mathbf{a b}$: $\mathbf{a b} = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \wedge \mathbf{b}$ and $-\mathbf{b a} = \mathbf{a} \cdot \mathbf{b} - \mathbf{b} \wedge \mathbf{a}$

Inner product $\mathbf{a} \cdot \mathbf{b}$: vectors are orthogonal $\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a} = 0$ and self collinear $\mathbf{a} \cdot \mathbf{a} = \mathbf{b} \cdot \mathbf{b} = 1$.

Outer Product $\mathbf{a} \wedge \mathbf{b}$: bivector orientation is anti-commutative $\mathbf{a} \wedge \mathbf{b} = -\mathbf{b} \wedge \mathbf{a}$.

Spinor and Pseudoscalar: $I = (\mathbf{a b})$ where $(\mathbf{a b})(\mathbf{a b}) = -\mathbf{a a b b} = -1$ or $(\mathbf{a b}) = \sqrt{-1}$

Multivector: $A = \langle A \rangle_0 + \langle A \rangle_1 + \dots = \sum_r \langle A \rangle_r$, $\langle A \rangle_0 = \text{scalars}$, $\langle A \rangle_1 = \text{vectors}$, $\langle A \rangle_2 = \text{bivectors}$

For any 1-vector \mathbf{x} : since $\mathbf{x}^2 = 1$, so $\mathbf{x} = \mathbf{x}^{-1}$. For $X = (\pm 1 \pm \mathbf{x})$: X^{-1} is undefined, so X is singular.

Projectors $P_k = -R_k$: $R_0 = C_{--} = (1 - \mathbf{a})(1 - \mathbf{b}) = [+000]$, $R_1 = C_{-+} = [0+00]$, $R_2 = C_{+-} = [00+0]$, $R_3 = C_{++} = [000+]$

Eigenvectors: In \mathbf{G}_2 , $E_k^2 = +1$, $E_k R_k = R_k$, $P_k^2 = P_k$, then $E_k = (\pm \mathbf{a} \pm \mathbf{b} \pm \mathbf{a b}) = R_k - 1$, $\sum P_k = +1$

Qubit notation and properties: $\mathbf{Q}_1 = \mathbf{G}_2$

Qubit A: is defined for $\mathbf{Q}_1 = \mathbf{G}_2 = \text{span}\{\mathbf{a0}, \mathbf{a1}\}$. Spinor $\mathbf{S}_A = \mathbf{a0 a1}$, Pauli spin $\mathbf{P}_A = (-1 + \mathbf{S}_A)$

Qubit states $A = (\pm \mathbf{a0} \pm \mathbf{a1})$: $A_0 = (+\mathbf{a0} - \mathbf{a1})$, $A_1 = (-\mathbf{a0} + \mathbf{a1})$, $A_+ = (+\mathbf{a0} + \mathbf{a1})$, $A_- = (-\mathbf{a0} - \mathbf{a1})$,

Qubit properties: $A_0 = -A_1$, $A_+ = -A_-$ and also $1/A_0 = -A_0 = A_1$, $1/A_+ = -A_+ = A_-$

Phases: Classical: $\{A_0, A_1\} = (\pm \mathbf{a0} \mp \mathbf{a1})$, $A_0 A_1 = 1$, *Superposition:* $\{A_+, A_-\} = (\pm \mathbf{a0} \pm \mathbf{a1})$, $A_+ A_- = 1$

Hadamard: $A_{0/1} \mathbf{S}_A = A_{\pm}$, $A_{\pm} \mathbf{S}_A = A_{1/0}$. *Inverter:* $A \mathbf{S}_A \mathbf{S}_A = -A$, *Pauli:* $A_{0/1} \mathbf{P}_A = \mp \mathbf{a1}$, $A_{\pm} \mathbf{P}_A = \pm \mathbf{a0}$.

Quantum register notation and properties: $\mathbf{Q}_q = \mathbf{G}_{n=2q}$

Quantum register AB: is defined as $\mathbf{Q}_2 = \mathbf{G}_4 = \text{span}\{\mathbf{a0}, \mathbf{a1}, \mathbf{b0}, \mathbf{b1}\}$, $\mathbf{S}_A, \mathbf{S}_B, \mathbf{P} = \mathbf{P}_A \mathbf{P}_B$

Geometric is Tensor Product: $AB = (\pm \mathbf{a0} \pm \mathbf{a1})(\pm \mathbf{b0} \pm \mathbf{b1}) = (\pm \mathbf{a0} \mathbf{b0} \pm \mathbf{a0} \mathbf{b1} \pm \mathbf{a1} \mathbf{b0} \pm \mathbf{a1} \mathbf{b1})$

Pseudoscalar: is a 4- vector $\mathbf{S}_A \mathbf{S}_B = (\mathbf{a0} \mathbf{a1} \mathbf{b0} \mathbf{b1})$ where $\text{reverse}(\mathbf{S}_A \mathbf{S}_B) = \mathbf{S}_A \mathbf{S}_B$

Pauli spin: $\mathbf{P} = \mathbf{P}_A \mathbf{P}_B = (-1 + \mathbf{S}_A)(-1 + \mathbf{S}_B) = (+1 - \mathbf{S}_A - \mathbf{S}_B + \mathbf{S}_A \mathbf{S}_B)$

Singlets: $A_0 B_0 \mathbf{P}_A \mathbf{P}_B = A_0 \mathbf{P}_A B_0 \mathbf{P}_B = \mathbf{a1} \mathbf{b1} = \mathbf{S}_{11}, \mathbf{S}_{00} = \mathbf{a0} \mathbf{b0}, \mathbf{S}_{01} = \mathbf{a0} \mathbf{b1}, \mathbf{S}_{10} = \mathbf{a1} \mathbf{b0}$

Sequential spinors: $A_0 B_0 (\mathbf{S}_A \mathbf{S}_B) = A_+ B_+ = (+ \mathbf{a0} \mathbf{b0} + \mathbf{a0} \mathbf{b1} + \mathbf{a1} \mathbf{b0} + \mathbf{a1} \mathbf{b1})$

Concurrent spinors: $A_0 B_0 (\mathbf{S}_A + \mathbf{S}_B) = (- \mathbf{a0} \mathbf{b0} + \mathbf{a1} \mathbf{b1}) = - \mathbf{S}_{00} + \mathbf{S}_{11} = \mathbf{B}_0$ Bell state = Φ^+

Bell Recursive Operator: $\mathbf{B} = (\mathbf{S}_A + \mathbf{S}_B)$ where $\mathbf{B}_{i+1} = \mathbf{B}_i \mathbf{B}$ for $i = \{0-3\}$, \mathbf{B}^{-1} is undefined.

Bell States: $\mathbf{B}_0 = (-\mathbf{S}_{00} + \mathbf{S}_{11}) = \Phi^+$, $\mathbf{B}_2 = (\mathbf{S}_{00} - \mathbf{S}_{11}) = \Phi^-$, $\mathbf{B}_1 = (\mathbf{S}_{01} + \mathbf{S}_{10}) = \Psi^+$, $\mathbf{B}_3 = (-\mathbf{S}_{01} - \mathbf{S}_{10}) = \Psi^-$

Magic Recursive Operator: $\mathbf{M} = (\mathbf{S}_A - \mathbf{S}_B)$, $\mathbf{M}_{i+1} = \mathbf{M}_i \mathbf{M}$ for $i = \{0-3\}$, \mathbf{M}^{-1} is undefined.

Magic States: $\mathbf{M}_0 = (\mathbf{S}_{01} - \mathbf{S}_{10})$, $\mathbf{M}_2 = (-\mathbf{S}_{01} + \mathbf{S}_{10})$, $\mathbf{M}_1 = (-\mathbf{S}_{00} - \mathbf{S}_{11})$, $\mathbf{M}_3 = (\mathbf{S}_{00} + \mathbf{S}_{11})$

Phase difference: $\mathbf{M}_3 = \mathbf{B}_0 - \mathbf{S}_{00} = \mathbf{B}_0 (\mathbf{S}_{01} + \mathbf{S}_{10}) = \mathbf{B}_0 \mathbf{B}_1$

Direction reversal: $\mathbf{B}_{i-1} = -\mathbf{B}_i \mathbf{B} = \mathbf{B}_i \mathbf{P}_A \mathbf{P}_B$ and $\mathbf{M}_{i-1} = -\mathbf{M}_i \mathbf{M} = \mathbf{M}_i \mathbf{P}_A (-1 - \mathbf{S}_B)$

Multiplicative cancellation: $\mathbf{M}_i \mathbf{B} = \mathbf{B}_i \mathbf{M} = 0$ and since $A_+ B_+ = \mathbf{M}_3 + \mathbf{B}_1$ then $A_+ B_+ \mathbf{B} = \mathbf{B}_2$

Sparse Invariants: $\mathbf{B}^2 = (1 - \mathbf{S}_A \mathbf{S}_B) = \mathbf{I}^-$, $\mathbf{M}^2 = (1 + \mathbf{S}_A \mathbf{S}_B) = \mathbf{I}^-$, and $\mathbf{B}^4 = \mathbf{B}^2 = \mathbf{I}^+ = (\mathbf{I}^\pm)^2$

Vector notation: $+1 = [++++ +++++ +++++ +++++]$, $-1 = [----- ----- ----- -----]$

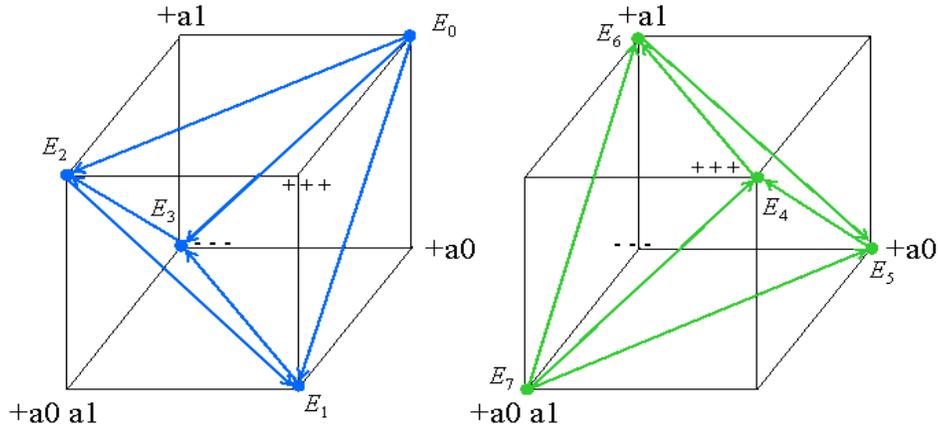
Sparse +1: $\mathbf{B}^4 = \mathbf{I}^+ = [0++0 +00+ +00+ 0++0]$, $\mathbf{M}^4 = \mathbf{I}^+ = [+00+ 0++0 0++0 +00+]$

Sparse -1: $\mathbf{B}^2 = \mathbf{I}^- = [0--0 -00- -00- 0--0]$, $\mathbf{M}^2 = \mathbf{I}^- = [-00- 0--0 0--0 -00-]$

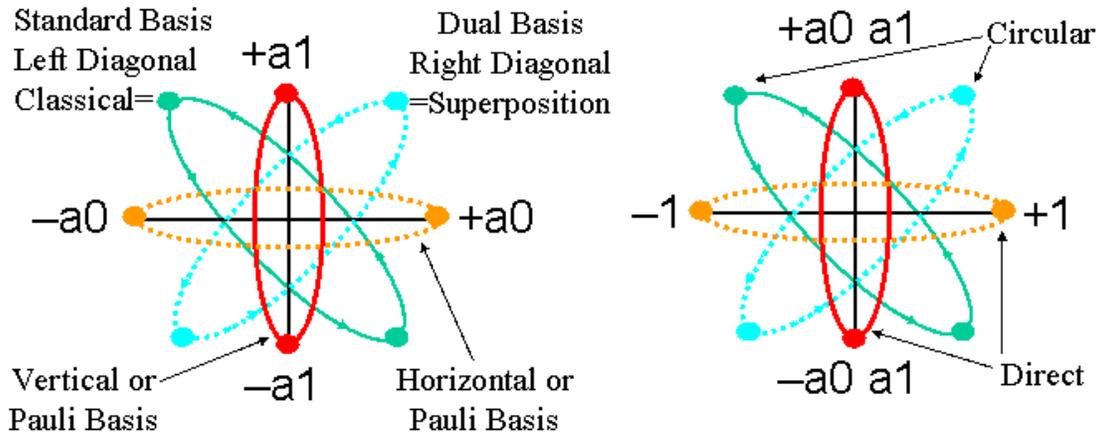
Control-Not: $CNOT_{AB} = A_0$ for $A B CNOT_{AB}$ and \mathbf{B} or \mathbf{M} since $\mathbf{B}^2 = \mathbf{I}^-$ and $\mathbf{M}^2 = \mathbf{I}^-$

Control-Hadamard: $CHAD_{AB} = \pm 1 + A_0$ and $\sqrt{\mathbf{B}} = \mathbf{I}^+ + \mathbf{B}$ or $\sqrt{\mathbf{M}} = \mathbf{I}^+ + \mathbf{M}$

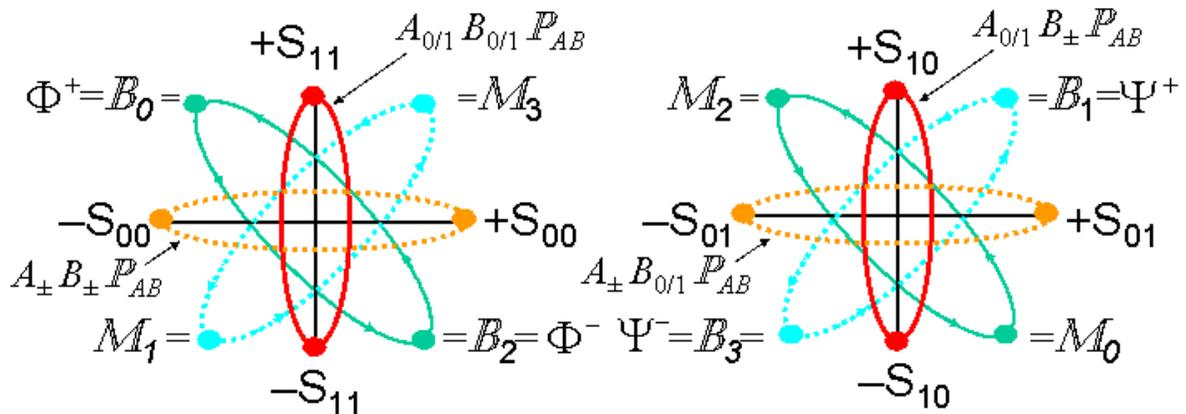
This page contains a recap of figures previously shown in chapters four, five, and six.



Projection Operators P_i form sides of dual tetrahedrons for $\mathbf{Q}_1 = A = (\pm\mathbf{a0}\pm\mathbf{a1})$



Geometric representation of reversible bases for $\mathbf{Q}_1 = A = (\pm\mathbf{a0}\pm\mathbf{a1})$



Geometric representation of Bell and Pauli bases for $\mathbf{Q}_2 = AB = (\pm\mathbf{a0}\pm\mathbf{a1})(\pm\mathbf{b0}\pm\mathbf{b1})$

REFERENCES

- [1] A. Barenco, C.H. Bennett, R. Cleve, D.P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin and H. Weinfurter. "Elementary gates for quantum computation". *Phys. Rev. A*. vol. 52, No.5, pp. 3457-3467, 1995.
<ftp://eve.physics.ox.ac.uk/Archive/Numbered/BBCDMSSW95/paper.ps>
- [2] J. D. Bekenstein and M. Schiffer. *Int. Journal of Mod. Phys.* **C1**, p 355, 1990.
- [3] J. Bell. "On the Einstein-Podolsky-Rosen Paradox". *Physics*. Vol. 1, pp. 195-200, 1964.
- [4] C. H. Bennett, F. Bessette, G. Brassard, L. Salvail, and J. Smolin. "Experimental quantum cryptography". *Journal of Cryptology*. vol. 5, no. 1, pp. 3-28, 1992. Also see paper by C. Bennett, G. Brassard, and A. Ekert. "Quantum cryptography". *Scientific American*. p. 50. October 1992 and online paper "What is Quantum Cryptography".
<http://www.qubit.org/intros/crypt.html>.
- [5] C. H. Bennett, G. Brassard, C. Crepeau, R. Jozsa, A. Peres, and W. Wootters.
"Teleporting an Unknown Quantum State via Dual Classical and EPR Channels".
Physical Review Letters. Vol. 70, pp. 1895-1899, 1993.
- [6] Charles Bennett. "Logical Reversibility of Computation". *IBM Journal of Research and Development*. Vol. 17, pp. 525-532, 1973.
- [7] C. D. Cantrell. *Modern Mathematical Methods for Physicists and Engineers*. Cambridge University Press, 2000.
- [8] D. Deutsch, A. Ekert. "Machines, Logic and Quantum Physics". Los Alamos e-print archive at <http://xxx.lanl.gov/abs/math.HO/9911150>, 1999.

- [9] C.J.L. Doran. Handouts for course “Physical Applications of Geometric Algebra”. see <http://www.mrao.cam.ac.uk/~clifford/ptIIIcourse/>. Handout for lecture 4 on Geometric Algebra and Quantum Mechanics, Section 2 on “Spinors and Multivectors”.
- [10] A. Ekert, et al. “Basic Concepts in Quantum Computation”. Oxford Quantum Computing web site at <http://www.qubit.org/intros/comp/houches.ps>.
- [11] Richard Feynman. “Simulating physics with computers”. *International Journal of Theoretical Physics*. Vol. 21, No. 6/7, pp. 467-488, 1982.
- [12] Ed Fredkin. “Digital Mechanics”. *Physica D*. pp. 254-270, 1990.
- [13] Edward Fredkin and Tommaso Toffoli. “Conservative Logic”. *International Journal of Theoretical Physics*. vol. 21, pp 219-253, 1982.
- [14] L. K. Grover. “A fast quantum mechanical algorithm for database search”. in *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*. Philadelphia, PA. pp. 22-24, May 1996. (New York: ACM 1996) pp. 212-219.
- [15] Jozef Gruska. *Quantum Computing*. McGraw-Hill, 1999. See book information on web site <http://www.mcgraw-hill.co.uk/gruska>.
- [16] David Hestenes. *New Foundations for Classical Mechanics (Second Edition)*. Kluwer Academic Press, 1999.
- [17] Pentti Kanerva. *Sparse Distributed Memory*. MIT Press, 1988.
- [18] E. Knill, R. Laflamme, R. Martinez, C.-H. Tseng. “An algorithmic benchmark for quantum information processing”. *Nature*. Letters to Editor (23 Mar 2000) vol. 404. pp. 368-370. and news release from LANL about the first seven-qubit quantum computer using NMR <http://www.lanl.gov/worldview/news/releases/archive/00-041.html> on March 22, 2000.

- [19] Rolf Landauer. "Information is Physical". *Proceedings of the Workshop on Physics and Computation, PhysComp92*. IEEE Computer Society Press. Los Alamitos, CA, 1992.
- [20] Rolf Landauer. "Information is Physical". *Physics Today*. Vol. 44, pp. 23-29, 1991.
- [21] J. Lasenby, A.N. Lasenby and C.J.L. Doran. "A unified mathematical language for physics and engineering in the 21st century". *Phil. Trans. R. Soc. Lond. A* **358**. pp. 21-39, 2000.
- [22] Nick Lawrence. "Corobs and Neurophysiology: A white paper". Available at <http://www.lt.com>, May 1998.
- [23] Michael Manthey. "A Combinatorial Bit Bang Leading to Quaternions". See paper number 9809033 on LANL Eprints server at <http://eprints.lanl.gov>, Sept 1998.
- [24] Norman Margolus. "Physics and Computation". *Ph.D. Dissertation MIT/LCS/TR-415*, March 1988.
- [25] Frank Mattern. "Quantum Computing Introduction". Paper found on his website <http://www.rommel.stw.uni-erlangen.de/~frank/informatik/QuantumComputing.pdf>, August 1999. (also see page 3).
- [26] Carver Mead. *Introduction to VLSI Systems*. 1980.
- [27] Doug Matzke. "Will Physical Scalability Sabotage Performance Gains?". *Computer Magazine*. Vol. 30, No. 9, pp 37-39, September 1997.
- [28] Doug Matzke. "Information is Protophysical". *Proceedings of the Workshop on Physics and Computation, PhysComp96*. New England Complex System Institute, 1996.
- [29] Teresa Meng, Sharad Malik (Editors). *Asynchronous Circuit Design for VLSI Signal Processing*. 1994.
- [30] C. Rorabaugh. *Error Coding Cookbook*. McGraw Hill. Chapter 2 on Galois Fields, 1996.

- [31] M. Schiffer. "The interplay between Gravitation and Information Theory". *Proc. of the Workshop on Physics and Computation, PhysComp92*. IEEE Computer Society Press, 1993.
- [32] M. Schiffer and J. D. Bekenstein. *Phys Rev.* **D39**, p 1109, 1989.
- [33] M. Schiffer. *Phys Rev.* **A43**, p 5337, 1991.
- [34] Peter Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In *Proceedings of 35th Annual Symposium on the Foundations of Computer Science*. IEEE Computer Society Press. Los Alamitos, CA. page 124, 1994.
- [35] John von Neumann, "Mathematical Foundations of Quantum Mechanics", translated from the German by Robert Beyer (Princeton University Press, 1955), pp. 249-254.
- [36] John Wheeler. "It From Bit". *Proceedings of the 3rd International Symposium on Foundations of Quantum Mechanics*. Tokyo, 1989.

VITA

Douglas J. Matzke was born in Green Bay, Wisconsin on April 29, 1953. After graduation from Southwest High School in 1971, he attended the University of Wisconsin in Madison, Wisconsin. He spent his junior year abroad studying at a private technical college in Monterrey, Mexico. During his senior year, he was the part time co-chairperson for a Student Competition On Relevant Engineering (SCORE) program, which allowed 50 college student teams to enter designs for the Energy Resource Alternatives (ERA) competition. He graduated in December 1975 with a Bachelor of Science in Electrical Engineering. In January 1976, he started working full time at Texas Instruments in Austin, Texas. While working full time for the Digital Systems Division at TI, he married and acquired a Masters of Science in Electrical Engineering at the University of Texas in Austin in 1980. He then moved to TI Dallas and soon joined the central research laboratories where he worked on the design of the DARPA funded Lisp chip, which won the ISSSCC Best paper award in February 1987. During the late 1980's he created a fast compiled-code, reversible, discrete-time simulation system and language which was built into an integrated CAD system called DROID. Primarily for his DROID efforts, he was elected to Senior Member of Technical Staff in 1990. His interests in limits of computation led to grant funding and becoming chairman of PhysComp92 and PhysComp94 workshops and later a paper in September 1997 in IEEE Computer. Because of his increasing interest in quantum computing, he started his EE Ph.D. in quantum computation in September 1999 at UT Dallas. He is working full time for Lawrence Technologies in Addison, Texas on high-dimensional software applications.