Synthesis and Evaluation of Fault-Tolerant Quantum Computer Architectures

by

Andrew W. Cross

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

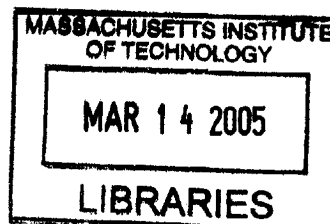
February 2005

© Massachusetts Institute of Technology 2005. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
January 28th, 2005

Certified by . . . .
Isaac L. Chuang
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
MAR 1 4 2005
LIBRARIES

BARKER

Room 14-0551
77 Massachusetts Avenue
Cambridge, MA 02139
Ph: 617.253.2800
Email: docs@mit.edu
http://libraries.mit.edu/docs

DISCLAIMER OF QUALITY

Due to the condition of the original material, there are unavoidable flaws in this reproduction. We have made every effort possible to provide you with the best copy available. If you are dissatisfied with this product and find it unusable, please contact Document Services as soon as possible.

Thank you.

The images contained in this document are of the best quality available.

The Archives copy contains grayscale images only. This is the best version available.

Synthesis and Evaluation of Fault-Tolerant Quantum Computer Architectures

by

Andrew W. Cross

Submitted to the Department of Electrical Engineering and Computer Science
on January 28th, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Fault-tolerance is the cornerstone of practical, large-scale quantum computing, pushed into its prominent position with heroic theoretical efforts. The fault-tolerance threshold, which is the component failure probability below which arbitrarily reliable quantum computation becomes possible, is one standard quality measure of fault-tolerant designs based on recursive simulation. However, there is a gulf between theoretical achievements and the physical reality and complexity of envisioned quantum computing systems. This thesis takes a step toward bridging that gap. We develop a new experimental method for estimating fault-tolerance thresholds that applies to realistic models of quantum computer architectures, and demonstrate this technique numerically. We clarify a central problem for experimental approaches to fault-tolerance evaluation – namely, distinguishing between potentially optimistic pseudo-thresholds and actual thresholds that determine scalability. Next, we create a system architecture model for the trapped-ion quantum computer, discuss potential layouts, and numerically estimate the fault-tolerance threshold for this system when it is constrained to a local layout. Finally, we place the problem of evaluation and synthesis of fault-tolerant quantum computers into a broader framework by considering a software architecture for quantum computer design.

Thesis Supervisor: Isaac L. Chuang
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I am ever grateful to Isaac Chuang for taking me as a student in his quantum computing group. His love of science and clarity of expression are always an encouragement. He has taught me to question more deeply than I had before, and to think carefully about the right questions, as they are at least half the problem. I am thankful for the opportunities he has created for me.

Tzvetan Metodiev contributed many layout and circuit synthesis tools I have used to construct recursively simulated circuits. He and Darshan Thaker also contributed the Planar object of the quantum architecture simulator and related code. It has been great fun traveling to see Setso and Darshan, and I have enjoyed weekly phone conversations with Setso.

Krysta Svore read early drafts of my thesis and her comments improved its clarity. Her prior work has also inspired the observations about pseudo-thresholds. I admire and have learned a lot from her way of working.

Ken Brown has always graciously answered my questions, however off-topic, and taught me to tell a story when I talk about my research (I am still learning).

I have had the wonderful opportunity to work with Al Aho, who tells it like it is. He has been a great encouragement, and I can't thank him enough for setting aside time to meet with me and read this thesis.

I am thankful to know Aram Harrow, Rob Clark, Jarek Labaziewicz, and Andrew Houck. Conversations with them have made my stay with the group very enjoyable, and I have appreciated their comments and advice.

Terri Yu helped me transition into the group. Thank you Terri.

My friends and family have always been very patient and supportive, even when work might keep me from the phone or busy over holidays. Thank you for understanding.

Abby, thank you for your love and support. I'm sorry I couldn't take your suggestion to title this "Faults I Have Tolerated."

This work was funded by an NDSEG graduate fellowship.

Contents

1 Introduction **25**

1.1 Background 25

1.2 Organization and Overview 27

2 Efficient Simulation of Quantum Error Correction Circuits **31**

2.1 Quantum Error Correction Circuits 32

2.2 Stabilizer Formalism 36

2.2.1 The Pauli Group, Stabilizers, and Stabilizer States 36

2.2.2 The Clifford Group and Stabilizer Circuits 39

2.2.3 The Gottesman-Knill Theorem and the $\mathcal{C}_k$ Hierarchy 40

3 Fault-Tolerance Thresholds **43**

3.1 Introduction 43

3.2 Triple Modular Redundancy (TMR) 44

3.3 Quantum Fault-Tolerance 49

3.3.1 Definitions for Quantum Fault-Tolerance 49

3.3.2 Quantum Triple Modular Redundancy 54

4 Fault-Tolerance Thresholds are Experimentally Observable **59**

4.1 Introduction 60

4.2 Criteria for Scalable Fault-Tolerance 61

4.2.1 DiVincenzo Criteria 62

4.2.2 Fault-Tolerance Criteria 63

4.3 Releasing Common Assumptions 65
4.3.1 Leakage and qubit loss 65
4.3.2 Ancilla Preparation 66
4.3.3 Locality and geometric constraints 66
4.4 Experimentally Estimating Thresholds 67
4.4.1 Experimental Method 68
4.4.2 Classical Example 69
4.4.3 Evaluating quantum fault-tolerance 71
4.4.4 Quantum Examples 72
4.5 The Pseudo-threshold problem 77
4.5.1 Classical Pseudo-thresholds 77
4.5.2 Quantum Pseudo-thresholds 85
4.6 Conclusion 89

5 Threshold for Trapped-Ion Quantum Computers 91

5.1 Trapped-Ion Quantum Information Processing 92
5.1.1 Ion-qubits and RF Traps 92
5.1.2 Gates and Measurement 94
5.1.3 Moving Ions between Traps 98
5.1.4 Noise and other imperfections 100
5.1.5 Revisiting the DiVincenzo criteria 100
5.2 Ion-Trap System Architecture Model 101
5.2.1 Stabilizer Simulation Model 101
5.2.2 Model implementation summaries 105
5.3 Quantum Computer Layouts 107
5.3.1 Turing machine layouts 108
5.3.2 H-Tree layouts 109
5.3.3 Ancilla-supported layouts 113
5.4 Thresholds for Trapped-Ions 120
5.4.1 $[[3,1,1]]$ bit-flip code 121

5.4.2 [[7, 1, 3]] quantum code 123
5.5 Conclusion 126

6 A Software Architecture for Quantum Computer Design Tools 127

6.1 Quantum Computer Organization 128
6.2 Design Flow 130
6.2.1 Quantum assembly language 133
6.2.2 Quantum physical operations language 134
6.3 Simulation Software Implementations 137
6.3.1 Ion-trap Simulator (ITSIM) 137
6.3.2 Quantum Architecture Simulator (ARQ) 143
6.4 Conclusion 147

7 Conclusion 149

A Notation 153

B Ion-Trap Simulator (ITSIM) Source Code 155

B.1 Module Dependencies 155
B.2 Source Listing 156
B.2.1 aqc.pt 156
B.2.2 bundle.py 157
B.2.3 cell.py 159
B.2.4 chain.py 160
B.2.5 chp.c 162
B.2.6 chp.h 171
B.2.7 chp.i 172
B.2.8 control.py 173
B.2.9 grid.py 178
B.2.10 ion.py 179
B.2.11 iontrap.py 180
B.2.12 llparse.py 181

B.2.13 logger.py 183
B.2.14 parse.py 183
B.2.15 physics.py 184
B.2.16 polparse.py 184
B.2.17 propagationMethod.py 187
B.2.18 pureMethod.py 187
B.2.19 stabilizerMethod.py 188

C Quantum Architecture (ARQ) Simulator Source Code 191

C.1 Module Dependencies 191
C.2 Source Listing 192
C.2.1 CGMachine.cc 192
C.2.2 CGMachine.He 194
C.2.3 ITMachine.cc 194
C.2.4 ITMachine.hh 196
C.2.5 Machine.cc 196
C.2.6 Machine.hh 201
C.2.7 NoisyCGMachine.cc 201
C.2.8 NoisyCGMachine.hh 203
C.2.9 NoisyITMachine.cc 203
C.2.10 NoisyITMachine.hh 206
C.2.11 NoisyPCGMachine.cc 206
C.2.12 NoisyPCGMachine.hh 208
C.2.13 Op.cc 208
C.2.14 Op.hh 209
C.2.15 PCGMachine.cc 209
C.2.16 PCGMachine.hh 211
C.2.17 Planar.cc 211
C.2.18 Planar.hh 212
C.2.19 QState.cc 212

C.2.20 QState.hh 220
C.2.21 arq.cc 221
C.2.22 commandLine.cc 222
C.2.23 commandLine.hh 223
C.2.24 globals.hh 223
C.2.25 htable.c 223
C.2.26 htable.h 225
C.2.27 interactive.cc 225
C.2.28 interactive.hh 228
C.2.29 loglevels.hh 228
C.2.30 lookupa.c 228
C.2.31 lookupa.h 230
C.2.32 noise.cc 230
C.2.33 noise.hh 232
C.2.34 phash.c 232
C.2.35 phash.h 232
C.2.36 planarops.h 233
C.2.37 planarops.c 233
C.2.38 standard.h 236
C.2.39 utilities.cc 236
C.2.40 utilities.hh 237

List of Figures

2-1 The quantum circuit shown encodes a single qubit $|\psi\rangle$ in a subspace of a 3-qubit Hilbert space. The resulting subspace is protected against errors that X-rotate a single physical qubit (i.e., errors of the form $U = e^{-i\theta X}$ for some θ). 33

2-2 This quantum circuit couples a 3-qubit quantum codeword to two ancillary qubits. Upon measurement, these ancillary qubits reveal the parity of adjacent bits and simultaneously project the quantum codeword onto a state with that parity. The measurement results locate bit-flip errors which can then be corrected by classically controlled correction operations. For example, the 3-qubit input state $|\psi\rangle = \sqrt{\epsilon}(\alpha|010\rangle + \beta|101\rangle) + \sqrt{1-\epsilon}(\alpha|000\rangle + \beta 111)$ leads to parity measurement 11 with probability ϵ and parity measurement 00 with probability $1-\epsilon$. These measurements indicate no error with probability $1-\epsilon$ and a bit-flip on the second qubit with probability ϵ. . . . 33

2-3 This quantum network maps the all-zero state $|0000000\rangle$ to the logical zero basis state of the Steane code (equation 2.6). The 3 Hadamard gates split the state into a superposition of 2^3 codewords, while the 9 controlled-NOT gates create the appropriate codewords of the dual code $C^{\perp}$. The logical zero basis state is an essential reference state in quantum error-correction circuits, often used to extract an error syndrome from the data without collapsing the state of the data to a state outside of the code subspace. 35

2-4 A recovery network for the 7-qubit Steane code. The input state $|q_0\rangle$ is the encoded data to be corrected, and the input states $|a_0\rangle$ and $|a_1\rangle$ are encoded logical zero states. The first ancilla state $|a_0\rangle$ determines the bit-flip error location, and the second ancilla state $|a_1\rangle$ determines the phase-flip error location. Each gate and measurement is a block of 7 operations applied qubit-wise. The "Decode Syndrome" block is a classical circuit that applies two parity check matrices to the 14 measurement outcomes, outputting a pair of syndromes. The syndromes classically control a recovery operation R that applies an appropriate bit-flip and/or phase-flip to the data. 35

3-1 The fault-tolerant classical wire (or identity gate) expands the original faulty wire in triplicate and inserts three fan-outs and three majority voting gates "•" to amplify the correct signal. Failures can occur in locations marked by "×". Failures in the leftmost BSC are *gate failures* and failures in the rightmost BSC are *voter failures.* Any single gate or majority voter failure won't flip a majority of the output wires. Assume that the final value is read by a very reliable process, so that only a majority of the output wires need to carry the correct value. . 45

3-2 The fault-tolerant wire is even more reliable below threshold when recursively simulated. Each wire and majority gate in Figure 3-1 is expanded in triplicate and each failure location replaced by Figure 3-1 itself. The resulting fault-tolerant wire can tolerate 3 or fewer failures distributed in any way. 47

3-3 Transversal quantum gates are gates that can be applied bit-wise to transform the encoded input qubit(s) to the proper encoded output state. This figure shows transversal CNOT and Hadamard gates. Transversal gates are fundamental fault-tolerant gates because they limit the spread of errors by restricting interaction in a very simple way. All of the Clifford group gates (i.e. $\mathcal{C}_2$) are transversal for the CSS codes. 51

4-1 This circuit detects if $|\bar{\psi}\rangle$ has leaked from the computational Hilbert space. If this is the case, then the CNOT gates will act like the identity on qubit a_0 and the measurement outcome will be 0. Otherwise, the measurement outcome will be 1. If leakage occurs, the damaged qubit is reinitialized to $|0\rangle$. 65

4-2 The failure probability of a fault-tolerant classical wire computed using Monte-Carlo simulation and evaluation of products of Markov chain transition matrices. The gate locations and voter locations both fail with probability p_0 initially (i.e., $\gamma_g = \gamma_m = p_0$. This plot shows the reliability of a single fault-tolerant wire ($T = 1$) and gives crossing points $p_{0,1} \approx 0.226$, $p_{1,2} \approx 0.164$, and $p_{2,3} \approx 0.153$. 70

4-3 The failure probability of a fault-tolerant classical wire computed using Monte-Carlo simulation and evaluation of products of Markov chain transition matrices, like Figure 4-2. This plot shows the reliability of a single fault-tolerant wire computed by composing $T = 2$ fault-tolerant wires, then dividing the resulting failure probability by $T = 2$ to give an average failure probability. The crossing points $p_{1,2}$ and $p_{2,3}$ have shifted down by about 0.02 due to propagating errors. 70

4-4 This figure shows the $p_1(p)$, $p_2(p)$, and $p_3(p)$ curves for the fault-tolerant recovery network of the 3-qubit bit-flip code. The points are numerically computed data, and the solid lines are polynomial fits to that data. The solid black line is the line $p_f/T = p_0$ included for reference. Independent bit-flip noise occurs prior to every gate in the network with probability p_0. The vertical axis is the average failure probability the L-simulated recoveries, computed by composing $T = 32$ recoveries and dividing the failure probability by $T = 32$. The numerically estimated threshold is $p_{th} \approx 0.035$. 74

4-5 The $p_1(p)$ and $p_2(p)$ curves for the [[7, 1, 3]] fault-tolerant recovery network [Pre01]. Each point is numerically computed data, and the solid lines are polynomial fits to that data. The solid black line $p_f/T = p_0$ is included for reference. The numerically estimated threshold is $p_{th} \approx 7.2 \times 10^{-4}$. 76

4-6 The construction for a fault-tolerant majority voter is identical to a fault-tolerant wire in Figure 3-1 except the first wire is replaced by a voter. If input errors are ignored, the wire and voter have the same failure probability. However, input errors cause the wire and voter to behave differently in recursive simulations. 80

4-7 This sequence of graphs shows the image of the rectangle $R = [0, \frac{1}{2}] \times [0, \frac{1}{2}]$ under several iterations of the probability map given by equation 4.36. In particular, this rectangle is an invariant set, meaning $\vec{\gamma}(R) \subseteq R$, and the origin and the upper right hand corner are attracting fixed points. 82

4-8 This plot characterizes the fixed points of the probability map given by equation 4.36. Green circles denote attractive fixed points, and red circles denote saddle points. The white arrows are eigenvectors of the linearized system at each fixed point. Ingoing arrows correspond to associated eigenvalues with magnitude less than 1 and outgoing arrows correspond to associated eigenvalues with magnitude greater than 1. 84

5-1 Schematic of the ion trap used in ion shuttling experiments at NIST Boulder [RBKD+02], courtesy of David Wineland. Individual ions are trapped near electrodes 2 and 4. Ions can be moved by adjusting the static potentials on electrodes 1, 3, and 5. 93

5-2 Energy levels of the atom-oscillator system together with carrier-driven transitions ($\omega_L = \omega_0$). Carrier-driven transitions rotate the qubit to desired superposition states. 96

5-3 Energy levels of the atom-oscillator system together with red-driven transitions. Red-driven transitions change the populations of the qubit energy levels and the oscillator energy levels simultaneously. The oscillator loses one quanta of vibrational energy in a transition on the red sideband. 97

5-4 Energy levels of the atom-oscillator system together with blue-driven transitions. Blue-driven transitions change the populations of the qubit energy levels and the oscillator energy levels simultaneously. The oscillator gains one quanta of vibrational energy in a transition on the blue sideband. 97

5-5 A sample layout of a QCCD ion trap architecture is specified by a trap matrix. Each non-empty location in the trap matrix can contain an electrode or a single ion. 102

5-6 This is one possible layout for ion-trap quantum computing based on a Turing machine with a finite circular tape. The blue squares represent non-conductive substrate that supports electrodes represented by gray rectangles. Black squares are empty space in which ions can be trapped. Each ion is drawn as a colored sphere. Green spheres are ions that hold data while the blue sphere is an ion used for sympathetic cooling. The green line represents a beam of laser light striking an ion in the accumulation region, the region where quantum logic operations take place. 109

5-7 This H-tree fractal is generated from the central "H" by scaling and translation. As an ion-trap layout, the gray region should be interpreted as a large channel with many smaller rows and columns of traps. The white region should be interpreted as substrate. A 6-simulated quantum error-correction circuit on the H-tree looks like this figure. . 110

5-8 The basic unit of the H-tree layout for the 3-qubit bit-flip code is constructed from a trap with 6 channels that branch from a central channel. Three of these channels store data qubits, shown in green, and three of these channels store ancillary qubits, shown in red. Ancilla and data qubits each share their channels with a sympathetic cooling ion shown in blue. The ancilla are prepared first using the central channel, then move transversally to extract an error syndrome from the data. The data is stationary for the entire error-correction process and only moves to interact with other data qubits. 111

5-9 An ion-trap layout for a 3-simulated 3-qubit quantum memory. The solid circles represent ions and the cyan squares represent substrate. Black regions of the layout represent free space in which ions can be trapped and moved. Green circles are data ions, red circles are ancilla ions, and blue circles are sympathetic cooling ions. 112

5-10 An alternate view of Figure 5-9. 112

5-11 A single logical qubit and its ancilla support structure reside in a plane for a quantum computer built from a 3-dimensional lattice of ions. The left column of green circles represents the data qubits in an L-codeblock. The remaining columns of red and blue circles represent ancilla qubits. Each column of red ancilla qubits is a 1-superblock and each column of blue ancilla qubits is a 2-superblock. The pattern continues – for each q-superblock with $q > r > 0$ there are M_r r-superblocks supporting that q-superblock. 115

5-12 This is the same structure shown in Figure 5-11 when $M_k = 1$ for all k. The data codeblock is on the left. Each box of qubits to the right of the data is called a *q-chunk* – one q-superblock to correct data and the rest to correct ancilla in the box. There are L chunks to correct data encoded at level-L. 116

5-13 This is a 1-supported L-block for which data and ancilla are interleaved. M_1 red level 1 ancilla follow each of the n green data qubits. These nM_1 level 1 ancilla are used in level 1 error-correction of the n data qubits, though the data qubits are part of a larger L-codeblock. There are $n_v(M_1 + 1)$ gray verification qubits used to prepare up to $(M_1 + 1)$ 1-codeblocks, the total number contained in the 1-supported L-block if the data qubits need to be prepared as well. 117

5-14 This is also a 1-supported L-block, as in Figure 5-13, except data is placed adjacent to the ancilla and verifiers. Compared to the interleaved block, this block has a lower ancilla preparation failure rate due to movement but a higher data-ancilla interaction failure rates due to movement. 118

5-15 This is an interleaved 2-supported L-block. Notice that Figure 5-13 is contained within this figure n times for the green data qubits and n times for the blue level 2 ancilla qubits. A total of $n_v(M_2 + 1)n$ gray verifiers V_1 appear at the end of the block to verify the blue level 2 ancilla qubits. These V_1 verifiers have their own verification qubits V_0 because the V_1 verifiers themselves must be prepared into a logical zero state and verified. 118

5-16 This is an adjacent 2-supported L-block. There are the same number of qubits as in Figure 5-15. The qubits have different locations but the same roles, and they are labeled as before. 119

5-17 This figure shows how the failure probability of an L-simulated fault-tolerant recovery network for the $[[3,1,1]]$ code depends on both gate and movement failure probabilities under bit-flip noise. The blue surface is the average failure probability of a recovery network for a 0-concatenated code, and the red surface is the average failure probability of a recovery network for a 1-concatenated code (i.e., a 1-simulated recovery network). The "gate" axis is the failure probability $p_g = p_1 = p_2$ and the "move" axis is the failure probability p_t (see Table 5.2). There is a region where recursive simulation improves reliability of the recovery network – the region where the red curve passes beneath the blue curve. The neighborhood of the crossing manifold is colored gray; these points can be identified with neither the red nor the blue surface within the standard error. 122

5-18 This is Figure 5-17 viewed from above. This plot emphasizes the set of reliable parameters in the lower left. These parameters lie approximately within the triangle given by $(0.005, 0.0005)$, $(0.005, 0.0027)$, $(0.03, 0.0005)$. The neighborhood of the crossing manifold is colored gray; these points can be identified with neither the red nor the blue surface within the standard error. 123

5-19 This figure shows how the failure probability of an L-simulated fault-tolerant recovery network for the [[7, 1, 3]] quantum code depends on both gate and movement failure probabilities under depolarizing noise. The blue surface is the average failure probability of a recovery network for a 0-concatenated code, and the red surface is the average failure probability of a recovery network for a 1-concatenated code (i.e., a 1-simulated recovery network). The "gate" axis is the failure probability $p_g = p_1 = p_2$ and the "move" axis is the failure probability p_t (see Table 5.2). There is a region where the red surface passes beneath the blue surface. Recursive simulation improves reliability of the recovery network in this region. Grey portions of both surfaces represent interpolated data that could be attributed to neither the blue nor the red surface within the standard error. 124

5-20 This is Figure 5-19 viewed from above. This plot emphasizes the region of reliable parameters which are shown in blue. 125

6-1 The four-phase design flow accommodates most approaches to quantum computer system design. Each phase maps between languages representing quantum circuits with varying degrees of technology dependence. 132

6-2 The ITSIM module dependence diagram (MDD) includes 10 different components. Lines connecting the modules signify inclusion, instantiation, or some other dependence. See appendix B for a more detailed MDD. 139

6-3 Snapshot of the ITSIM graphical display showing an H-tree layout. Qubits are ions represented by spheres, and gates are applied using laser pulses, represented by lines. The qubits can move within the black regions of the figure and are prohibited from moving into the substrate which is drawn using light squares. In the right window the simulator displays feedback regarding the current operations, noise induced failures, and estimated execution time. 143

6-4 The ARQ virtual quantum machine hierarchy contains 7 machines. The *classical control* (`cc`) machine lies at the bottom of the hierarchy and implements conditions and branching. The *Clifford group* (`cg`) machine includes the classical control machine and adds measurement, CX, CZ, H, S, X, Y, and Z gates. The *planar Clifford group* (`pcg`) machine adds a layout and the accompanying movement and placement instructions. Finally, the *Clifford ion trap* (`cit`) machine adds ion-trap specific instructions such as cooling. Each machine has a noisy equivalent (`ncg, npcg, ncit`). 145

B-1 The full ITSIM module dependence diagram. Arrows represent Python `import` dependence. The module at the arrow tail imports the module at the arrow head. 155

C-1 This diagram shows ARQ source dependencies. Each source file at an arrow tail includes the source file at the arrow head. 191

List of Tables

4.1 This table lists sources of error in the quantum TMR circuit. The first column is the error source which is a subset of locations in the circuit. The second column is the failure probability δ associated with an individual error event. The third column is the number of locations N in the error source. 87

4.2 This table lists the location thresholds and pseudothresholds for the quantum TMR circuit. Wait errors have the lowest location threshold, and CNOT gates have the largest difference between thresholds and pseudothresholds. 89

5.1 Summary of the trapped-ion system model as implemented in the IT-SIM simulation tool. The upper half of the table lists important points regarding the trap layout, ion chaining, heating, and memory error model. The lower half of the table lists the available operations and their adjustable parameters. 106

5.2 Summary of the trapped-ion system model as implemented in the ARQ simulation tool. The ARQ tool does not model the ion-trap in as much detail as the ITSIM tool, but the simplified model allows us to explore larger systems at higher levels of recursive simulation. 107

5.3 Summary of physical parameters for ion trap system models. The current experimental values of movement and splitting parameters are taken from [RBKD+02, CSB+04]. The current experimental values of gate parameters are taken from [DMW+98] and [DDV+03]. Sympathetic cooling times can be found in [BDS+03]. Projected parameter values are motivated by [WH00]. 108

Chapter 1

Introduction

1.1 Background

There appears to be mounting evidence that quantum computers are more powerful than their classical counterparts. Shor's historic factoring algorithm can factor composite integers in polynomial time, meaning that quantum computers have the potential to break public-key cryptosystems [Sho94]. While Shor's result provides the most significant algorithmic motivation for studying quantum computation at present, quantum search algorithms also offer modest improvements over their classical counterparts [Gro97]. Quantum simulation algorithms may also offer advantages over classical simulation techniques. Finally, significant algorithmic speedups have been found for other problems ([HRTS03, Hal02] and references therein).

Of course, the path to realizing a quantum computer is not without obstacles. One of the major obstacles to realizing a working, large-scale quantum computing system is noise. Noise results from uncontrolled interactions with spurious fields, material defects, and even fluctuations in the vacuum itself. These uncontrolled interactions cause the sensitive quantum state of the computer to become correlated with the environment in a way that typically destroys the desired unitary evolution of a quantum computation. As information about the quantum state is lost to the environment, the state loses quantum coherence and begins to behave like a classical random variable. After a short amount of time, the quantum state is no longer useful

for quantum computation, and the computer gives the wrong answer.

Theoretical results by Calderbank, Shor, and Steane show that quantum states can be encoded using quantum error-correcting codes [Sho95, CS96, Ste96]. The encoded quantum state is more robust than the unencoded state in the presence of some kinds of noise. In fact, quantum computation can take place entirely with encoded quantum states, and quantum gates can be organized such that failures do not cause errors to spread to too many qubits [Sho96, Pre01, Got98b, ABO99]. This discovery makes the concept of a large-scale quantum computing system possible, for without fault-tolerance, practical large-scale quantum computing appears to be doomed.

One method of constructing fault-tolerant quantum circuits recursively replaces components with new components that are increasingly more reliable. The recursive construction applies stronger encoding while reducing the probability of a fatal failure. If the basic components are more reliable than some threshold reliability, this recursive construction produces an exponentially more reliable circuit for only a polynomial gate overhead. The threshold reliability, called the fault-tolerance threshold, is a crucial figure of merit for fault-tolerant circuit designs, because quantum gates that physicists construct in the laboratory must be at least this reliable to enable large-scale quantum computation.

The theoretical groundwork for fault-tolerant quantum computation has been put in place by a series of heroic theoretical discoveries. However, there is a gulf between these theoretical achievements and the physical reality and complexity of envisioned quantum computing systems. For example, locality constraints are among the most important constraints influencing the fault-tolerance threshold. Introducing locality constraints creates the potential for additional trade-offs between space, time, and reliability. Furthermore, locality constraints must be realistic in view of current trends in quantum information processing experiments.

This thesis takes a step toward bridging the gap between theory and practice. We develop new experimental techniques for evaluating quantum fault-tolerant designs, and demonstrate these techniques numerically by estimating fault-tolerance thresh-

olds for the $[[3,1,1]]$ and $[[7,1,3]]$ quantum codes. We identify a central problem for experimental evaluation of fault-tolerance, namely how to distinguish between thresholds and so-called pseudo-thresholds, and suggest directions for future research to solve this problem. The new experimental techniques apply to realistic models of quantum computer architectures, with multiple failure probability parameters and local gates. We create a system architecture model for trapped-ion quantum computers, the most promising candidate for future large-scale quantum information processing experiments. Using this system architecture model, we construct several potential layouts for the trapped-ion quantum computer, and estimate a fault-tolerance threshold for trapped-ion computing with the $[[3,1,1]]$ and $[[7,1,3]]$ codes. Finally, we place the problem of evaluation and synthesis of fault-tolerant quantum computers into a broader framework by considering a software architecture for quantum computer design. We create a design flow and implement the lowest levels of this design flow as concrete software tools, including the complete source code in the appendices of this thesis.

1.2 Organization and Overview

This section describes the thesis structure and briefly reviews the contents of each chapter. The thesis is organized into seven chapters: three introductory chapters, three chapters containing new results, and one concluding chapter. This chapter provides introduction and thesis organization. Chapter 2 contains examples of how quantum error-correction circuits can be efficiently simulated. Chapter 3 introduces fault-tolerance thresholds by example. Chapter 4 motivates the need for experimental methods to evaluate fault-tolerant designs and describes a new technique, demonstrated numerically, for estimating quantum fault-tolerance thresholds. This chapter gives examples of how pseudo-threshold behavior arises in even the simplest fault-tolerant systems, and identifies pseudo-threshold behavior as a central problem for experimental evaluation of quantum fault-tolerance. Chapter 5 creates a system model for the trapped-ion quantum computer within the framework of Chap-

ter 4, investigates several layouts for such a computer, and estimates fault-tolerance thresholds for computing on a local layout. Finally, Chapter 6 motivates the need for software tools for evaluation and synthesis of quantum computer architectures, such as the architecture studied in Chapter 5. Chapter 6 presents a design flow realized as a concrete set of software tools and describes simulation tools this author has developed within the design flow.

In more detail, these are the contents of each chapter:

- Chapter 2 reviews quantum error-correction circuits and a technique for efficiently simulating them. The first half of the chapter discusses circuits for 3-qubit and 7-qubit quantum error-correcting codes. These codes are canonical examples in the field and we use them in Chapters 3, 4, and 5. The second half of the chapter describes the stabilizer formalism as a means to simulate quantum error-correction circuits.

- Chapter 3 reviews fault-tolerance and the fault-tolerance threshold by means of classical and quantum fault-tolerant design examples.

- Chapter 4 reviews the typical assumptions behind threshold estimates and how some of these assumptions can be lifted in detailed system designs. The chapter describes an experimental approach to evaluating quantum fault-tolerant designs. It presents a new numerical technique that gives experimental estimates of the fault-tolerance threshold for a certain family of quantum computer system models. Finally, the chapter identifies the problem of distinguishing pseudo-thresholds from thresholds as central to future experimental fault-tolerance studies.

- Chapter 5 begins by reviewing the trapped-ion quantum information processor. The chapter begins with high-level concepts and experimental progress realizing building blocks for trapped-ion quantum computing. We distill these concepts into a concrete architectural model within the framework of Chapter 4. Finally, we estimate a fault-tolerance threshold for a particular trapped-ion quantum computer system model.

- Chapter 6 discusses a software architecture for quantum computing design tools and emphasizes the role of quantum computer architecture simulation tools in this framework. The chapter describes two concrete implementations of quantum computer architecture simulators and suggests future work that could improve these tools.
- Chapter 7 concludes the thesis by reviewing the main results, establishing relationships between the ideas in this thesis, and suggesting future research directions.

Chapter 2

Efficient Simulation of Quantum Error Correction Circuits

Quantum mechanics applies to both few-body and many-body systems. As long as the quantum system remains isolated, a wave function describes its state at all times. However, if the system cannot be *perfectly* isolated, interaction between the system and the surrounding environment induces a preferred basis to which all or part of the system irreversibly damps [Zur81, Zur82]. In practice, very good isolation may be possible, but perfect isolation is not because we must ultimately interact to learn something about the system. Hence, the state of a quantum computer always suffers from this fundamental source of noise, which is called decoherence.

Section 2.1 uses examples to introduce quantum error-correction circuits. These circuits can counter the effect of decoherence on a specific subspace of the quantum computer's state space, so they are of great practical importance [Sho95, CS96, Ste96]. Quantum error-correction is quite general because the fine details of the underlying noise process are less important than the overall strength and space-time correlation length. Quantum error-correction is also an engineered solution to the decoherence problem, in that codes can be *designed* for specific system-environment interactions.

Section 2.2 introduces a mathematical framework for efficient quantum simulation on a digital computer based on the Gottesman-Knill theorem [Got98a]. The stabilizer formalism describes how a particular subset of quantum states evolves under a

restricted set of quantum gates. This collection of states and gates is precisely the collection used in quantum error-correction circuits based on stabilizer codes [Got97].

2.1 Quantum Error Correction Circuits

This section reviews specific examples of quantum circuits for encoding quantum information and recovering that information after correctable errors occur. Further details can be found in primary sources [Sho95, CS96, Ste96] and textbooks [NC00].

Classical repetition codes provide a very simple set of quantum error-correction circuits that can either correct bit-flip errors in the computational basis $\{|0\rangle, |1\rangle\}$, or phase-flip errors in the conjugate basis

$$|0\rangle \longrightarrow (|0\rangle + |1\rangle)/\sqrt{2}, \tag{2.1}$$

$$|1\rangle \longrightarrow (|0\rangle - |1\rangle)/\sqrt{2}. \tag{2.2}$$

The encoding circuit shown in Figure 2-1 maps a single qubit state

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{2.3}$$

to a three qubit codeword

$$|\bar{\psi}\rangle = \alpha|000\rangle + \beta|111\rangle. \tag{2.4}$$

During quantum computation, encoders are rarely needed to encode unknown quantum states. Specific preparation circuits typically prepare well-defined reference states, such as basis states of a quantum code. For the repetition code, the logical zero basis state is $|0_L\rangle = |000\rangle$. The preparation circuit for this state is simply measurement followed by conditioned bit-flips.

The original state $|\bar{\psi}\rangle$ can be recovered if any one qubit is flipped. A network that measures the parity of adjacent qubits in the codeword, and nothing more, reveals what qubit has been flipped without determining anything about the coefficients α and β. The circuit in Figure 2-2 measures the parity of adjacent qubits, called

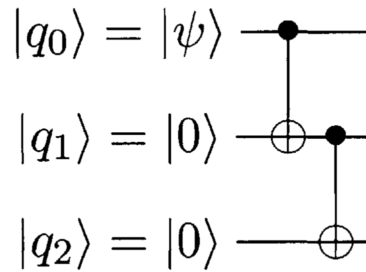


Figure 2-1: The quantum circuit shown encodes a single qubit $|\psi\rangle$ in a subspace of a 3-qubit Hilbert space. The resulting subspace is protected against errors that X-rotate a single physical qubit (i.e., errors of the form $U = e^{-i\theta X}$ for some θ).

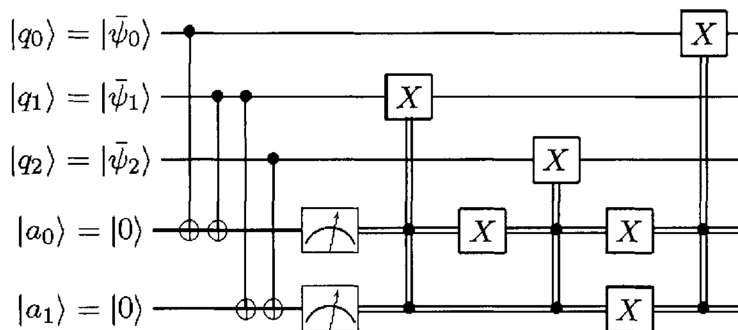


Figure 2-2: This quantum circuit couples a 3-qubit quantum codeword to two ancillary qubits. Upon measurement, these ancillary qubits reveal the parity of adjacent bits and simultaneously project the quantum codeword onto a state with that parity. The measurement results locate bit-flip errors which can then be corrected by classically controlled correction operations. For example, the 3-qubit input state $|\psi\rangle = \sqrt{\epsilon}(\alpha|010\rangle + \beta|101\rangle) + \sqrt{1-\epsilon}(\alpha|000\rangle + \beta 111)$ leads to parity measurement 11 with probability ϵ and parity measurement 00 with probability $1-\epsilon$. These measurements indicate no error with probability $1-\epsilon$ and a bit-flip on the second qubit with probability ϵ.

the *error syndrome*, and applies a classically controlled correction operation. This network is an example of a *recovery network.* The recovery network in Figure 2-2 behaves correctly even if the gates in the network are unreliable (in a very particular way, see Chapter 4), but the correction operation must be postponed until several syndromes have been collected.

The 3-qubit quantum error-correction network may be useful in early experiments to verify quantum error-correction and is a conceptually simple example. However, the 3-qubit redundancy code is unsatisfying because it only protects against failures

that are linear combinations of bit-flips and non-errors.

The 7-qubit Steane code (i.e., the $[[7,1,3]]$ quantum code) is the canonical example of a quantum code that can correct an arbitrary quantum error on a single qubit [Ste96]. The Steane code, based on a classical Hamming code C, distributes the quantum information of 1 qubit in non-local correlations of 7 qubits. Each codeword in the Hamming code C is annihilated by the parity check matrix

$$H = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}. \tag{2.5}$$

Figure 2-3 shows a circuit that prepares the logical zero state of the Steane code,

$$|0_L\rangle = \frac{1}{\sqrt{8}} \sum_{w \in C^{\perp}} |w\rangle \quad = \tfrac{1}{\sqrt{8}}(|0000000\rangle + |1010101\rangle + |0110011\rangle + |0001111\rangle \tag{2.6}$$

$$+|0111100\rangle + |1011010\rangle + |1100110\rangle + |1101001\rangle). \tag{2.7}$$

This state is an equally weighted superposition of codewords in the dual code $C^{\perp}$ [CS96], which contains all of the even codewords of C.

The recovery network for the Steane code is based on the same general concept as the 3-qubit recovery network [Ste96, Ste97]. Several ancillary qubits are coupled to the 7-qubit quantum codeword and measured. The measurement results relate to the parity checks in H that reveal an error syndrome. Figure 2-4 shows the explicit recovery network. The inputs at the left are 7-qubit codewords encoded using the Steane code. Each gate or measurement in this figure is a block of 7 gates or measurements acting qubit-wise. The classical "Decode Syndrome" circuit applies the parity check matrix to the measurement outcomes to determine the error syndrome. Classically controlled gates conditioned on the syndrome apply a recovery operation R consisting of no more than one bit and one phase flip.

The Steane code is an example of a general class of quantum codes called the Calderbank-Shor-Steane (CSS) codes [Sho95, CS96, Ste96]. These codes have desirable properties both as error-correcting codes and as codes for fault-tolerant quantum

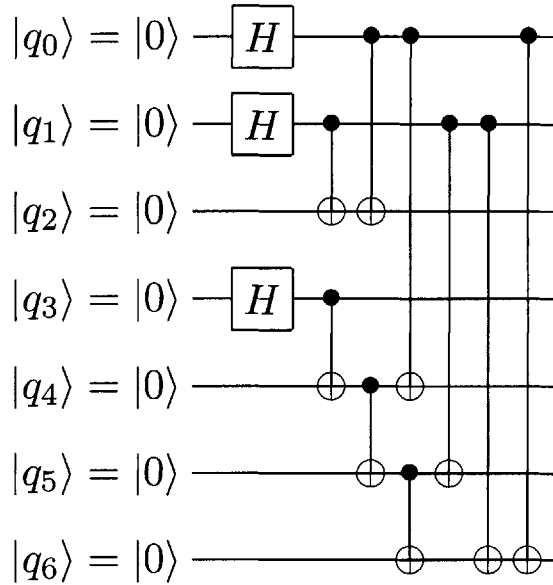


Figure 2-3: This quantum network maps the all-zero state $|0000000\rangle$ to the logical zero basis state of the Steane code (equation 2.6). The 3 Hadamard gates split the state into a superposition of 2^3 codewords, while the 9 controlled-NOT gates create the appropriate codewords of the dual code $C^{\perp}$. The logical zero basis state is an essential reference state in quantum error-correction circuits, often used to extract an error syndrome from the data without collapsing the state of the data to a state outside of the code subspace.

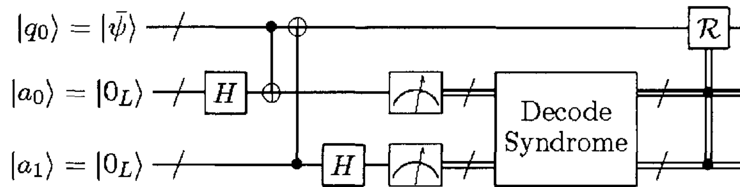


Figure 2-4: A recovery network for the 7-qubit Steane code. The input state $|q_0\rangle$ is the encoded data to be corrected, and the input states $|a_0\rangle$ and $|a_1\rangle$ are encoded logical zero states. The first ancilla state $|a_0\rangle$ determines the bit-flip error location, and the second ancilla state $|a_1\rangle$ determines the phase-flip error location. Each gate and measurement is a block of 7 operations applied qubit-wise. The "Decode Syndrome" block is a classical circuit that applies two parity check matrices to the 14 measurement outcomes, outputting a pair of syndromes. The syndromes classically control a recovery operation R that applies an appropriate bit-flip and/or phase-flip to the data.

circuit construction, as will be discussed later in Chapter 4.

2.2 Stabilizer Formalism

It is well known that no algorithm has been found to simulate general quantum systems in polynomial time on a classical computer. However, efficient simulation algorithms exist when the set of quantum states or gates is restricted [ABO96, Got98a, Vid03, VHP04]. Section 2.1 reviewed quantum error-correction circuits based on two CSS codes to illustrate that these circuits involve a very particular set of quantum gates and quantum states, specifically the Clifford group gates and stabilizer states. Quantum error-correction circuits can be efficiently simulated by using the stabilizer formalism [Got98a, AG04], which we will now review. The section assumes some basic familiarity with the theory of finite groups [Art91].

Subsection 2.2.1 reviews the n-qubit Pauli group and states stabilized by the commutative subgroups of the Pauli group. This set of states is the set involved in quantum error-correction (with stabilizer codes). Subsection 2.2.2 reviews the Clifford group of quantum gates and how these gates transform stabilizer states. We review the concept of a stabilizer circuit. Finally, subsection 2.2.3 states the well known Gottesman-Knill theorem and reviews the $\mathcal{C}_k$ hierarchy of groups.

2.2.1 The Pauli Group, Stabilizers, and Stabilizer States

The *Pauli group* is the group of tensor products of bit, phase, and bit-phase flips of n-qubits. More carefully, the single qubit Pauli group $\mathcal{G}$ is the group generated by the Pauli operators X and Z, as well as iI. Formally,

$$\mathcal{G} \equiv \langle X, Z, iI \rangle = \{1, i, -1, -i\} \otimes \{I, X, Y, Z\}. \tag{2.8}$$

Notice that $\mathcal{G}$ contains proper subgroups isomorphic to the quaternions, but is not itself the quaternion group. The n-qubit Pauli group $\mathcal{G}^{\otimes n}$ consists of all 4^{n+1} elements that are tensor products of n Pauli operators and four possible phases, written in

shorthand,

$$\mathcal{G}^{\otimes n} = \{1, i, -1, -i\} \otimes \{I, X, Y, Z\} \otimes \cdots \otimes \{I, X, Y, Z\}. \tag{2.9}$$

Because individual Pauli operators are composed according to $XY = iZ$, it is easy to see that $\mathcal{G}^{\otimes n}$ is a group. Typically elements of these groups are written in a shorthand that omits the tensor product symbol, so that $X \otimes X$ and XX represent the same element. The intended operation, group multiplication or tensor product, will be clear from context.

The stabilizer formalism uses a particular set of quantum states called *stabilizer states.* Before these can be defined, let us review the concept of a *stabilizer* S. A stabilizer S is an abelian subgroup of the Pauli group $\mathcal{G}^{\otimes n}$ that does not contain $-I$. Because the Pauli group and its subgroups are finite, S is usually expressed by a list of its generators $\{g_i\}$ for $i \in \mathbb{N}_m$. The generators of S must commute, and m commuting elements generate a group of size 2^m. Because elements of $\mathcal{G}^{\otimes n}$ either commute or anticommute, and $-I \notin S$, a stabilizer $S \subseteq \mathcal{G}^{\otimes n}$ cannot be generated by more than n elements. Let $\hat{S}$ denote a minimal generating set of S.

Suppose you are given a stabilizer $S \subseteq \mathcal{G}^{\otimes n}$ on n-qubits generated by m elements, i.e., $S = \langle g_i, i \in \mathbb{N}_m \rangle$. Take the first of these generators, g_1, and consider the subspace of all $+1$ eigenvectors of g_1 within the n-qubit Hilbert space,

$$\mathcal{H}_{\{g_1\}} = \left\{ |\psi\rangle \in \mathcal{H}^{\otimes n} \mid g_1|\psi\rangle = |\psi\rangle \right\}. \tag{2.10}$$

$\mathcal{H}_{\{g_1\}}$ is the 2^{n-1}-dimensional subspace *stabilized* by g_1. Continuing in this way, define the subspace $\mathcal{H}_S$ stabilized by S,

$$\mathcal{H}_S = \left\{ |\psi\rangle \in \mathcal{H}^{\otimes n} \mid g_i|\psi\rangle = |\psi\rangle \ \forall i \in \mathbb{N}_m \right\}. \tag{2.11}$$

Note that $\mathcal{H}_S$ is a 2^{n-m} dimensional subspace.

The n-qubit stabilizer state $|\psi_S\rangle$ is the two-dimensional subspace of the n-qubit Hilbert space stabilized by S. There is a one-to-one correspondence between stabi-

lizer states and stabilizer subgroups; hence, there are $2^{\mathcal{O}(n^2)}$ n-qubit stabilizer states [AG04]. Such an S must have n generators. Though the generators are not unique, the fact that each state can be represented by n elements of the Pauli group is the first hint that these states may be easy to manipulate with a digital computer.

Just to make the discussion concrete, consider the two-qubit "cat" state

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle). \tag{2.12}$$

To express this state using a stabilizer, we need to find two elements of $\mathcal{G}^{\otimes 2}$ that commute and are independent. Because $XX|00\rangle = |11\rangle$, this is a good first choice. Note also that $ZZ|11\rangle = (-1)^2|11\rangle$, so choose $S = \langle XX, ZZ\rangle$. The complete stabilizer is the group of products of these generators,

$$S = \{I, XX, ZZ, -YY\}. \tag{2.13}$$

Finally, there is a useful group homomorphism that makes stabilizer states easy to manipulate on a digital computer. Consider

$$\phi : \mathcal{G}^{\otimes n} \longrightarrow \mathbb{Z}_2^{\otimes 2n}, \tag{2.14}$$

defined by

$$\phi(g) = b_x(g) \otimes b_z(g), \tag{2.15}$$

where $b_x(g)$ (resp. $b_z(g)$) maps an n-qubit Pauli operator to an n-bit string with ones wherever the ith term of g is X (resp. Z) or Y. This is a homomorphism because $\mathcal{G}$ is isomorphic to $\mathbb{Z}_2 \otimes \mathbb{Z}_2$ modulo phase, due to the multiplication rule $XY = iZ$ (i.e., you can apply one of the standard homomorphism theorems). Hence, elements of the Pauli group can be represented as length $2n$ binary strings under binary addition, provided phase can be ignored. Under the homomorphism, a stabilizer state is represented by a table of n binary strings, each of length $2n$, rather than n Pauli operators. This polynomial storage requirement is substantially better than storing 2^n complex

numbers to describe a general quantum state. In practical implementations, the phase is kept and a more complicated composition rule is used, rather than direct binary addition [DM03].

2.2.2 The Clifford Group and Stabilizer Circuits

This section reviews the Clifford group, a subgroup of quantum gates that have a high degree of symmetry in their action on stabilizer states. The *Clifford group*, written $\mathcal{C}_2$, is a subgroup of special unitary matrices

$$\mathcal{C}_2 = \mathcal{N}(\mathcal{G}^{\otimes n}) = \{U \in SU(2^n) \mid UgU^\dagger \in \mathcal{G}^{\otimes n}\ \forall g \in \mathcal{G}^{\otimes n}\} \tag{2.16}$$

that permute the elements of the Pauli group under conjugation. All Clifford groups will be labeled $\mathcal{C}_2$, leaving the number of qubits n to be determined from context. Note that $\mathcal{G}^{\otimes n}$ is properly contained in the Clifford group. The Clifford group is the normalizer of the Pauli group, so it is sometimes referred to as the Normalizer group. The key fact to observe is that Clifford group gates will transform stabilizer generators into stabilizer generators, hence stabilizer states evolve naturally under Clifford group gates.

For example, consider a single qubit stabilizer state $|\psi_S\rangle$. We may choose any Pauli group element $g \neq -I$ to generate S. For concreteness, choose $S = \langle Z \rangle$, which corresponds to $|\psi_S\rangle = |0\rangle$. Suppose we apply the Hadamard gate H to this state. The new state $H|\psi_S\rangle$ will be stabilized by $S' = HSH^\dagger$ because $H|\psi_S\rangle = HgH^\dagger H|\psi_S\rangle$. Note that $HZH = X$ and $HXH = Z$, so $H \in \mathcal{C}_2$. Hence, S' will also be an abelian subgroup of the Pauli group, and $|\psi_{S'}\rangle = H|\psi_S\rangle$ will also be a stabilizer state. Specifically, $S' = \langle X \rangle$ which has $+1$ eigenvalue $(|0\rangle + |1\rangle)/\sqrt{2}$.

Several gates can be shown to be Clifford group gates. The controlled-NOT gate acts on a control and a target qubit, from left to right, and has the following action

by conjugation:

$$XI \leftrightarrow XX, \tag{2.17}$$

$$IZ \leftrightarrow ZZ. \tag{2.18}$$

One interpretation of these equations is that "bit-flips" propagate from control to target and "phase-flips" propagate from target to control. The unlisted transformations act like the identity. Note that equations 2.17 and 2.18 imply that XZ maps to $-YY$, so some transformations require a phase change. The Hadamard gate has already been discussed, and we know it is a Clifford group gate. Finally, the $\pi/2$-phase gate S satisfies $SXS^\dagger = Y$, $SYS^\dagger = -X$, and $SZS^\dagger = Z$. These three gates, H, $CNOT$, and S, can generate any automorphism of the n-qubit Pauli group, meaning that any Clifford group gate can be expressed as a *stabilizer circuit* constructed from only these three gates.

2.2.3 The Gottesman-Knill Theorem and the $\mathcal{C}_k$ Hierarchy

Measurement of a Pauli operator M also transforms a stabilizer to stabilizer. Define the projectors $P_k = \frac{1}{2}(I + (-1)^k M)$. The post-measurement state is one of $P_k|\psi_S\rangle$, depending on the measurement outcome k. The probabilities of each outcome are

$$p(k) = \mathrm{Tr}(P_k|\psi_S\rangle\langle\psi_S|), \tag{2.19}$$

where the n-qubit stabilizer state can be written as a product of projectors,

$$\rho_S = |\psi_S\rangle\langle\psi_S| \tag{2.20}$$

$$= \prod_{g\in\hat{S}} \frac{1}{2}(I+g) \tag{2.21}$$

$$= \frac{1}{2^n}\prod_{g\in\hat{S}}(I+g). \tag{2.22}$$

There are two cases to consider because Pauli operators either commute or an-

ticommute with each other: either $\{M, g\} = 0$ for one or more generators g, or $[M, g] = 0$ for all generators g. For the first case, notice that $p(k) = \frac{1}{2}$ because elements of the Pauli group are traceless, and $M \notin S$, so products Mg are not proportional to the identity. The post-measurement state $|\psi_{S_k}\rangle$ is stabilized by $(-1)^k M$ as well as any $g \in S$ such that $[M, g] = 0$. Any $g \in S$ such that $\{M, g\} = 0$ can be replaced by gg', where $g' \in S$ also satisfies $\{M, g'\} = 0$, $g' \neq g$ for all g.

In the second case, $(-1)^k M \in S$, so the measurement outcome is deterministic and the post-measurement state is stabilized by $S_k = S$. In this case, the generators of S must be put into a standard form using Gauss-Jordan elimination to algorithmically determine the measurement outcome k.

Single qubit projective measurements in the computational basis are a special case of Pauli operator measurements, because a projective measurement corresponds to a measurement of Z on a particular qubit. Therefore, projective measurements can be resolved by updating the generators and computing the outcome using $M = Z_i$ for the ith qubit. The Gaussian elimination step may take $O(n^3)$ time when the outcome is deterministic. However, by storing slightly more information about the stabilizer, measurement outcomes can be resolved in $O(n^2)$ time for this special case [AG04]. By using an ancilla qubit, measurement circuits for general Pauli operators can be constructed and simulated in $O(n^2)$ time.

The Gottesman-Knill Theorem [Got98a] states that stabilizer circuits can be efficiently simulated on a digital computer:

> *Any quantum computer initially in a stabilizer state and performing only, a) Clifford group gates, b) Measurements of Pauli group operators, and c) Clifford group gates conditioned on classical bits which may be the results of earlier measurements, can be simulated in polynomial time on a probabilistic classical computer.*

The essential idea behind this theorem is to represent states by their stabilizer generators and update these stabilizer generators during the course of the simulation. Using equation 2.14, stabilizer generators are stored in a binary table. Clifford group

gates, particularly H, $CNOT$, and S, and projective measurements update this table in a straightforward manner [AG04].

Groups used in the stabilizer formalism fit within a general hierarchy of groups that reoccurs in the study of fault-tolerant quantum computation. The $\mathcal{C}_k$ hierarchy of groups [Got97] is a useful framework in which to view the Pauli and Clifford groups. The group $\mathcal{C}_1$ is the n-qubit Pauli group $\mathcal{G}^{\otimes n}$. The remaining groups in the hierarchy are defined recursively,

$$\mathcal{C}_k \equiv \left\{ U \in SU(2^n) \mid UVU^\dagger \in \mathcal{C}_{k-1} \ \forall V \in \mathcal{C}_{k-1} \right\}. \tag{2.23}$$

Notice that $\mathcal{C}_2$ is the Clifford group. The Clifford group gates are not universal for quantum computation, and may not even be universal for classical computation [AG04], but become a discrete universal set of quantum gates with the addition of a single gate from $\mathcal{C}_3 - \mathcal{C}_2$. Both the Toffoli and $\pi/8$-gate are in $\mathcal{C}_3$, for example.

Chapter 3

Fault-Tolerance Thresholds

This chapter reviews classical and quantum fault-tolerance through concrete examples. Section 3.1 introduces fault-tolerance in broad terms. Section 3.2 considers a canonical classical approach to fault-tolerance. We specifically review the fault-tolerant classical wire and set out to bound its fault-tolerance threshold. Section 3.3 reviews quantum fault-tolerance, then presents a quantum fault-tolerant wire and discusses its threshold. The quantum fault-tolerant wire, which has a rather high threshold, is useful for applications as a quantum memory.

3.1 Introduction

Von Neumann introduced the first fault-tolerant construction in 1956 [von56] in the context of neural networks. The concepts he introduced were thought to be necessary for early computer systems, whose vacuum tubes and switches frequently failed, and for proper human brain function, where individual neurons misfire. Subsequently, Winograd and Cohen [WC63] placed fault-tolerant computation in the context of information theory and reliable communication. Recent work on fault-tolerance treats reliability as a fungible physical resource that can be exchanged for space, time, or power [Imp03].

The transistors within modern computer systems are exceedingly reliable, so fault-tolerant design is unnecessary for most computing applications. However, as men-

tioned in Chapter 2, quantum systems of any scale suffer from fundamental noise, called decoherence, that damps their state to an element of a preferred basis, making the state useless for quantum computation [Zur81, Zur82]. Nevertheless, fault-tolerant constructions akin to Von Neumann's can make quantum computing systems exponentially more reliable with only polynomial gate overhead, provided that basic gates are more reliable than some threshold [Sho96, Pre01, KLZ98, Got98b, ABO99].

All of the examples of fault-tolerant quantum error-correction in Chapter 2 require ancillary reference states for reliably extracting error syndromes, and such ancilla states are often highly entangled. Hence, in addition to being of practical interest for quantum computation, fault-tolerant quantum gates are of fundamental physical interest as well. This is because they require preparation and transformation of highly nontrivial entangled quantum states, which are another state of matter that has only recently been appreciated.

3.2 Triple Modular Redundancy (TMR)

Consider a *wire* that carries a signal representing a bit of information (i.e. a 0 or a 1) to another location in space. Similarly, we could consider a *memory* that carries a bit value forward in time. The wire is simply an identity map on a single bit that takes that bit to a new location, and it is an example of a *component* that maps binary strings to binary strings. Because the wire is not a composite object in our view, we say that the wire is a *basic component.*

Wires may not always be reliable. Model a *p-faulty identity gate* as a binary symmetric channel (BSC) with parameter p, meaning that the output value of the wire differs from the input value with probability p and is unchanged with probability $1-p$. We use the BSC acronym to refer to a single BSC or to tensor products of BSCs, and use "wire" and "identity gate" interchangeably.

A *faulty component* is a probabilistic map on binary strings whose input/output map is modeled by a transition matrix. If the faulty component's input and output differ, the component has *failed.* The probability p is the *failure probability.* If the

component fails, it introduces an *error*, which in this (classical) case is a bit-flip i.e. equivalent to a NOT gate.

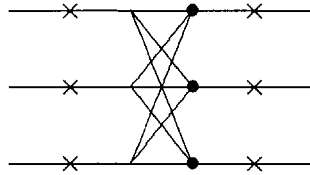


Figure 3-1: The fault-tolerant classical wire (or identity gate) expands the original faulty wire in triplicate and inserts three fan-outs and three majority voting gates "•" to amplify the correct signal. Failures can occur in locations marked by "×". Failures in the leftmost BSC are *gate failures* and failures in the rightmost BSC are *voter failures*. Any single gate or majority voter failure won't flip a majority of the output wires. Assume that the final value is read by a very reliable process, so that only a majority of the output wires need to carry the correct value.

The *fan-out gate* is a component that maps one binary input to three binary outputs that are copies of the input. The fan-out gate amplifies the input signal whether or not it is correct. For simplicity, assume that the fan-outs do not fail.

The *majority voting gate* is a component that maps three binary inputs a, b, c to one binary output $ab + bc + ca$, where addition is modulo 2. This gate outputs the dominant signal and decodes the 3-bit repetition code, i.e. the code that takes 0 to 000 and 1 to 111. A *p-faulty majority gate* is modeled as a majority voting gate followed by a binary symmetric channel with parameter p.

The circuit in Figure 3-1 is an example of a fault-tolerant wire. The input and output value is encoded using the simplest repetition code, which takes 0 to 000 and 1 to 111. The triplicated p-faulty wires are followed by triplicated fan-outs and triplicated p-faulty majority voting gates. The "×" symbols indicate binary symmetric channels that model wire or majority voter failure *locations*. Majority voting can correct errors introduced by a small number of failed components. This circuit is an example of triple modular redundancy (TMR).

The fault-tolerant wire is an example of a *composite component* because it is constructed from basic components. The fault-tolerant wire *fails* if the ideally decoded output differs from the input. To distinguish between composite component failure

and basic component failure, we call the former a *total failure* when looking at the composite component in isolation, as opposed to a basic failure. When basic components fail, they introduce errors. These errors propagate through the composite component in the form of bit-flips, specifically through fan-outs and incorrect majority voter outputs.

A basic component failure may not lead to a failure in the composite component. In fact, the essence of fault-tolerant design is the notion that a small number of basic component failures should not cause the composite component to fail. The fault-tolerant wire has the property that any isolated basic failure cannot cause a total failure.

Let $p_1(p)$ be the failure probability of a circuit acting on bits encoded once using TMR. By counting the number of ways the fault-tolerant wire can succeed, one can compute the exact probability of a total failure $p_1(p)$. Obviously, if 0 locations fail, that is a success. A single failure in any one of the 6 locations is also a success, since the majority gates restore the correct value. If two locations fail, the fault-tolerant wire can still succeed if those failures are distributed between the first and second binary symmetric channel, which can happen 9 different ways. Three errors will always lead to failure. Four, five, and six errors lead to success only if errors cancel. By symmetry, these situations occur in 9, 6, and 1 way(s), respectively. Summing terms of the form $ap^k(1-p)^{6-k}$ for integers a and k,

$$p_1(p) = 6p^2 - 4p^3 - 18p^4 + 24p^5 - 8p^6 \tag{3.1}$$

$$\leq 6p^2. \tag{3.2}$$

Notice that $p_1(p) \leq p$ if $p \leq 1/6$. Hence, for this analysis the fault-tolerant wire is more reliable than the basic wire when $p < 1/6$.

To make the fault-tolerant wire even more reliable, a different construction is required. Following Von Neumann [von56], replace each failure location in Figure 3-1 with Figure 3-1 itself and expand the majority gates to act on bundles of 3 wires at a time. This approach is called *recursive simulation* since the original wire has been

simulated by a fault-tolerant wire which itself can be simulated by replacing its wires with fault-tolerant wires.

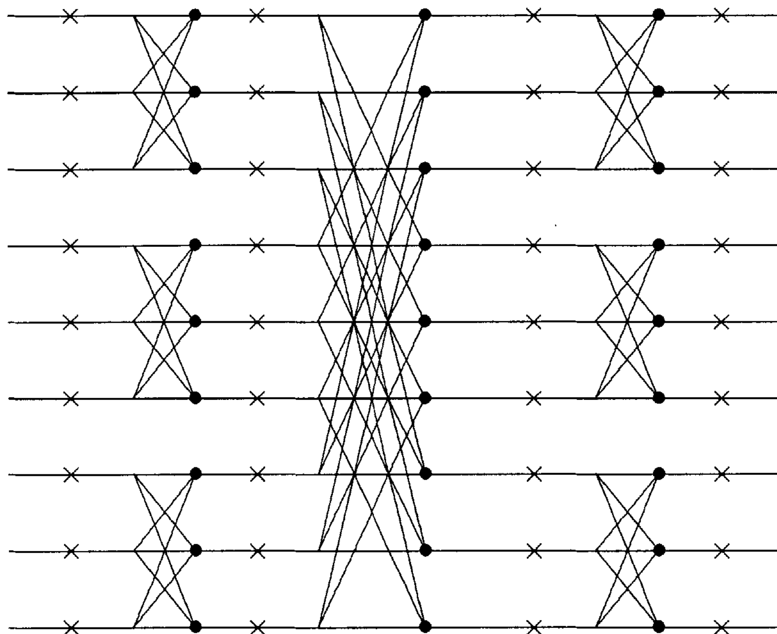


Figure 3-2: The fault-tolerant wire is even more reliable below threshold when recursively simulated. Each wire and majority gate in Figure 3-1 is expanded in triplicate and each failure location replaced by Figure 3-1 itself. The resulting fault-tolerant wire can tolerate 3 or fewer failures distributed in any way.

For example, Figure 3-2 shows the circuit resulting from recursive simulation of the original fault-tolerant wire in Figure 3-1. This circuit is 2-*simulated* because failure locations have been replaced twice by Figure 3-1, once for the original faulty wire, then again for the 6 locations in the composite wire. Similarly, Figure 3-1 is 1-simulated, and the original p-faulty wire is 0-simulated. The 2-simulated circuit is called a 2-*block*. The 2-block contains 6 1-blocks and 36 0-blocks. A 0-block is simply a failure location. Blocks are related to *rectangles*, another term used in the literature.

The 2-block can tolerate a total failure of any of its 6 1-blocks. These 1-blocks fail with probability less than or equal to $6p^2$, so the 2-block fails with approximate probability $6(6p^2)^2$. This observation leads us to approximate the failure probability

$p_L(p)$ of an L-simulated fault-tolerant wire as

$$p_L(p) \approx p_1^L(p) = \underbrace{(p_1 \circ \cdots \circ p_1)}_{\text{L times}}(p). \tag{3.3}$$

This one-parameter map $p_1(p)$ has a fixed point p_{th} that solves $p_1^L(p_{th}) = p_{th}$ for all L. For $p < p_{th}$, $\lim_{L\to\infty} p_1^L(p) = 0$. As long as $p_1^L(p) \geq p_L(p)$ for all p, $p_L(p)$ approaches 0 as well.

Recursive simulation produces a composite component that is increasingly reliable under some conditions, namely that all of the p-faulty basic components satisfy $0 < p < p_{th}$ for some nonzero *threshold failure probability* p_{th}. The threshold cannot depend on the number of gates in the composite component nor on the number of input and output bits of the component. If it did depend on these quantities in the limit of large L, there would be no way to guarantee the benefits of recursive simulation. In practice, all basic components must not behave much worse than independently failing p-faulty components.

The most basic theory that captures the threshold concept sets $p_1(p) \leq Cp^{t+1}$ for some nonzero real C and integer t. The parameter C bounds the number of ways that $t+1$ failures cause a total failure of a 1-block. Assume that the underlying error-correcting code can correct t errors, so only $t+1$ or more errors causes a total failure. Furthermore, C must include errors that can occur due to propagation into the 1-block, otherwise the recursive replacement step will be invalid. If one chooses p such that $p_1(p) \leq Cp^{t+1} \leq p$, then by self-similarity $p_2(p) \leq C\left[p_1(p)\right]^{t+1}$. By induction,

$$p_L(p) \leq \frac{1}{C}(Cp)^{(t+1)^L}. \tag{3.4}$$

Defining $p_{th} \equiv 1/C$,

$$\frac{p_L(p)}{p_{th}} \leq \left(\frac{p}{p_{th}}\right)^{(t+1)^L}. \tag{3.5}$$

Hence the failure probability of the L-simulated composite component approaches zero in the limit of large L as a double exponential. The threshold failure probability given by this construction is a combinatorial object; namely, the reciprocal of the

number of fault paths leading to total failure of a 1-block, counting input errors.

Let us review how many gates are required to construct an ϵ-faulty fault-tolerant wire when $p < p_{th}$ using recursive simulation. We want $\frac{p_L(p)}{p_{th}} \leq \epsilon$, which holds when $(p/p_{th})^{(t+1)^L} \leq \epsilon$. This implies that $(t+1)^L \leq \ln(1/\epsilon)/\ln(p_{th}/p)$. Now suppose that no more than $N = \alpha^L$ gates are needed for an L-simulated wire, for some positive real number α. Then

$$N \leq \exp\left(-\frac{\ln(t+1)}{\ln(\alpha)}\right)\frac{\ln(1/\epsilon)}{\ln(p_{th}/p)}, \tag{3.6}$$

so $N = \mathcal{O}(\log(1/\epsilon))$. Fortunately, one needs relatively few gates to achieve excellent reliability.

3.3 Quantum Fault-Tolerance

As we saw in Chapter 2, there are quantum error-correcting codes that can correct arbitrary errors on a fixed number of qubits. Fault-tolerant quantum circuits based on quantum error-correcting codes limit the spread of errors and periodically correct those errors. The basic concept behind quantum fault-tolerance and classical fault-tolerance is the same: a composite component that suffers t or fewer errors should not totally fail.

This section serves as a review of quantum fault-tolerance. Subsection 3.3.1 reviews important definitions for quantum fault-tolerance. Subsection 3.3.2 reviews the quantum equivalent of TMR as a preliminary to fault-tolerance using a general quantum error-correcting code.

3.3.1 Definitions for Quantum Fault-Tolerance

This section reviews the elements of quantum fault-tolerant circuits. The review includes quantum computation codes, L-simulations, fault-tolerant gates, noise, failure probabilities, and fault-tolerance thresholds.

Quantum computation codes

Like fault-tolerant classical circuits, scalably fault-tolerant quantum circuits are constructed using recursive simulation as well. However, there are several key differences. Quantum information involved in a fault-tolerant gate must be encoded using an $[[n,k,d]]$ quantum computation code [ABO99]. A *quantum computation code* is a quantum error-correcting code $\mathcal{C}$ accompanied by a discrete universal set of fault-tolerant gates $\mathcal{B}$ and by fault-tolerant encoding E, decoding D, and correction procedures R. Let $\mathcal{C}_*$ denote the entire construction $\langle \mathcal{C}, \mathcal{B}, E, D, R \rangle$. An $[[n,k,d]]$ quantum code encodes k qubits in n qubits with distance d, where d is the minimum weight Pauli element in $N(S) - S$ for a stabilizer code with stabilizer S. All fault-tolerant procedures must only use basic gates from $\mathcal{B}$, and the correction procedure must project any input onto a codeword. The quantum fault-tolerant gates $G \in \mathcal{B}$ limit the spread of single errors to $s < t$ qubits by design, where $t = \lfloor \frac{d-1}{2} \rfloor$ is the number of errors the quantum code corrects. Recursive simulation using quantum computation codes yields a threshold result as well, although the proof requires more care because quantum computers are neither entirely digital nor entirely analog [Sho96, Pre01, KLZ98, Got98b, ABO99].

Let an *L-simulation* for $L \geq 1$ be a recursive simulation using the $L-1$-concatenated code, where the 0-concatenated code is $\mathcal{C}$. Let E_L, D_L, and R_L be the L-simulations of the fault-tolerant procedures associated with $\mathcal{C}$.

The Calderbank-Shor-Steane (CSS) codes [Sho95, CS96, Ste96] are the family of quantum computation codes for which *logical* (i.e. composite) CNOT gates can legitimately act by *transversally* applying basic CNOT gates [Got97]. Figure 3-3 illustrates a transversal single qubit and two qubit gate. Transversal gates are fault-tolerant because they are implemented in a bitwise fashion, which automatically limits the spread of errors by restricting interaction. The CSS codes have other convenient properties due to their manner of construction that we will not review.

At least one nontrivial gate is necessary to construct a discrete universal set of fault-tolerant quantum gates. Nontrivial fault-tolerant gates such as the Toffoli gate

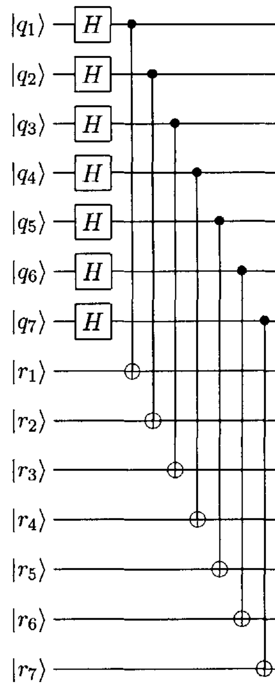


Figure 3-3: Transversal quantum gates are gates that can be applied bit-wise to transform the encoded input qubit(s) to the proper encoded output state. This figure shows transversal CNOT and Hadamard gates. Transversal gates are fundamental fault-tolerant gates because they limit the spread of errors by restricting interaction in a very simple way. All of the Clifford group gates (i.e. $\mathcal{C}_2$) are transversal for the CSS codes.

can be constructed using a general method based on quantum teleportation [ZLC00]. Fault-tolerant gates constructed in this manner consist entirely of $\mathcal{C}_2$ gates and projective measurements, both of which can be applied transversally. These gates accept a tensor product of input qubit states and a specially prepared ancilla state that is not a stabilizer state. See [Got97, Pre01] for a complete example of a Toffoli gate with ancilla state preparation.

Noise

We use a relatively simple model of noise in a quantum computer. An *ideal quantum circuit* G is a sequence of unitary gates and measurements, G_t, indexed by the integer time-step t in which each is applied. A *noise model* $\mathcal{N}$ is a sequence of quantum operations $\mathcal{E}_t(\rho)$ on the Hilbert space of the quantum computer. These maps are also indexed by the time-step at which each is applied. A *noisy quantum computation* $G_{\mathcal{N}}$ applies corresponding gates, measurements, and quantum operations at each time-step. If an ideal gate or measurement G_t and a quantum operation $\mathcal{E}_t(\rho)$ occur in the same time-step, $\mathcal{E}_t(\rho)$ happens first and is called a *gate failure*. Otherwise, $\mathcal{E}_t(\rho)$ is called a *memory failure*.

Fault-tolerant constructions are known to be effective for some types of correlated noise [TB04, ABO99]. However, for this thesis we restrict the noise models to the family of *independent, m-local noise models*. Independent m-local noise involves only those quantum operations that act on blocks of m or fewer contiguous qubits at the time the noise is applied. Each of these quantum operations $\mathcal{E}_t(\rho) \in \mathcal{N}$ has the form

$$\mathcal{E}_t = \mathcal{E}_t^{(1)} \otimes \cdots \otimes \mathcal{E}_t^{(M)}, \tag{3.7}$$

for M blocks of m or fewer qubits. Each noise operator $\mathcal{E}_t^{(i)}$ has the form

$$\mathcal{E}_t^{(i)}(\rho) = (1-p)\rho + p\mathcal{E}(\rho), \tag{3.8}$$

where $\mathcal{E}(\rho)$ is any quantum operation on ρ and p is a probability. For a particular $\mathcal{N}$,

let p be drawn from a set of *operation failure probabilities* P.

A quantum computer has a time dependent *state* that includes all of its quantum and classical degrees of freedom. An *operation* is a state changing action that can be assigned a well defined *operation failure probability* p_{op} and *operation duration* t_{op}. For example, quantum gates, projective measurements, and qubit movements by distance d are all operations.

Failure probability

Consider an $[[n, 1, 3]]$ quantum computation code $\mathcal{C}_*$ and an ideal quantum circuit G using only gates drawn from $\mathcal{B}$. Let G_L denote the ideal fault-tolerant quantum circuit obtained from G by L-simulation of each gate. For a noise model $\mathcal{N}$, $G_{L,\mathcal{N}}$ *fails* for an input density matrix ρ_0 if the ideal output density matrix $G(\rho_0)$ differs from $D_L \circ G_{L,\mathcal{N}} \circ E_L(\rho_0)$. Formally, define the failure probability p_L of $G_{L,\mathcal{N}}$ to be

$$p_L = 1 - \mathcal{F}^2 \leq D(G(\rho_0), D_L \circ G_{L,\mathcal{N}} \circ E_L(\rho_0)) \tag{3.9}$$

where $D(\rho, \sigma)$ is the trace distance between density matrices ρ and σ,

$$D(\rho, \sigma) = \mathrm{Tr}|\rho - \sigma|, \tag{3.10}$$

and $F(\rho, \sigma)$ is the fidelity,

$$F(\rho, \sigma) = \mathrm{Tr}\sqrt{\sqrt{\rho}\sigma\sqrt{\rho}}. \tag{3.11}$$

This definition isn't completely satisfying because it depends on the input state. The best measure of failure probability uses the *gate fidelity*

$$F(U, \mathcal{E}) = \min_{|\psi\rangle} F(|\psi\rangle\langle\psi|, \mathcal{E}(|\psi\rangle\langle\psi|)), \tag{3.12}$$

instead of the standard fidelity in equation (3.9).

Consider the following simple example to clarify the definition of failure probability. Let G consist of a single Hadamard gate applied to an unencoded qubit in

the state $\rho_0 = (|0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| + |1\rangle\langle 1|)/2$. Select a noise model that simply depolarizes the input qubit of each single qubit gate, $\mathcal{E}(\rho) = (1-p)\rho + pI/2$. The probability of failure p_f is

$$p_f = 1 - \langle 0|H\mathcal{E}(\rho_0)H^\dagger|0\rangle = p/2. \tag{3.13}$$

This is reasonable since with probability p the depolarizing channel replaces the input qubit by a fair coin that will fail half of the time for the given input.

Fault-tolerance threshold

A *fault-tolerance threshold for* $\mathcal{C}_*$ under an independent m-local noise model $\mathcal{N}$ is a constant failure probability $p_{th}(\mathcal{C}_*)$ such that when $p < p_{th}(\mathcal{C}_*)$ for all operation failure probabilities $p \in P$,

$$\lim_{L\to\infty} p_L = 0 \tag{3.14}$$

In fact, equation 3.5 also describes the functional behavior of p_L with L for quantum circuits when p_L is small. Typically, $p_{th}(\mathcal{C}_*)$ depends on the most complicated fault-tolerant gate in $\mathcal{B}$, i.e. simpler gate sets $\mathcal{B}$ lead to higher thresholds. The *maximum fault-tolerance threshold* p_{th},

$$p_{th} = \sup_{\mathcal{C}_*} p_{th}(\mathcal{C}_*), \tag{3.15}$$

is the highest achievable threshold over all fault-tolerant constructions. This threshold is difficult to calculate and is possibly higher than stated thresholds, which lie between 10^{-6} [ABO99] and 10^{-2} [Kni04, Rei04]. General constructions that increase the maximum fault-tolerance threshold and numerical models that probe new regions of the fault-tolerance parameter space are important because they reduce the experimental effort required to achieve quantum fault-tolerance. For the remainder of this thesis, we explicitly choose $\mathcal{C}_*$, so the extra notation to designate $\mathcal{C}_*$ will be dropped.

3.3.2 Quantum Triple Modular Redundancy

This section reviews one of the simplest codes to protect quantum information, the 3-qubit bit-flip code (or $[[3,1,1]]$ quantum code). Though qubits cannot be copied with unit fidelity, this code is a kind of repetition code analogous to TMR. There is no quantum code using fewer than 5 qubits that corrects an arbitrary quantum error [Got97]. Accordingly, the quantum TMR code can only correct arbitrary X rotations of a single qubit (i.e. bit-flips). In the conjugate basis (i.e. after applying Hadamard rotations to each qubit), quantum TMR can correct arbitrary Z rotations of a single qubit, but the code cannot correct both types of errors simultaneously as is required to correct an arbitrary quantum error. For this reason, we assume that only a single type of error can occur when discussing this code.

The quantum TMR encoder is given in Figure 2-1. If the state of the input qubit is one of the computational basis states, then the encoder simply copies that bit. Otherwise, the encoder creates an entangled quantum codeword $\alpha|000\rangle + \beta|111\rangle$ that represents the input state.

The recovery network for a quantum TMR encoded qubit is given in Figure 2-2. In order to ensure fault-tolerance, the first 4 CNOT gates and 2 measurements must be repeated at least three times before correcting the data. To see why this is the case, consider a single X error that occurs between the 2nd and 3rd CNOT gate on the control qubit. The resulting syndrome indicates that an error has occurred on the 3rd data qubit rather than the 2nd. If the recovery network acts on this syndrome alone, two errors can occur with probability p.

Several methods can estimate the fault-tolerance threshold of quantum TMR. We use two of them in this section: combinatoric analysis and positive operator analysis. A combinatoric analysis counts the number of ways failures within the circuit lead to total failure of the circuit. Combinatoric analysis leads to reasonable lower bounds on the fault-tolerance threshold, but can be used to give ball-park estimates as well. A positive operator analysis constructs the positive operator $\mathcal{A}(\rho)$ for the entire quantum circuit and directly computes the failure probability of that circuit. The positive

operator analysis gives exact results but requires an unreasonable amount of space for circuits much larger than the quantum TMR circuit.

The quantum TMR recovery network can be implemented at least two different ways. Let R_1 be the network implemented by collecting three syndromes and selecting the appropriate syndrome based on all three. The syndrome selection procedure is a majority vote except for a special case mentioned earlier that occurs on the second qubit between syndrome measurements. Let R_2 be the network implemented by collecting s syndromes expecting $s' < s$ to agree and aborting recovery if the first syndrome is zero or if there is no agreement.

By counting the number of places where two or more errors lead to total failure of R_1, we determine that the number of failure locations C should about 59 without considering errors that could have propagated from prior recovery operations, errors from failed initialization, and errors from waiting. This means $C \approx 59$ for R_1, so $p_{th} \approx 0.017$.

A positive operator analysis gives the exact failure probability of R_1. We do not include errors from waiting. Let $p_0(p)$ be the failure probability of a quantum circuit acting on physical qubits. The failure probability $p_0(p)$ of R_1 as a function of the single qubit and two qubit gate failure probability p is

$$p_0(p) = 59p^2 - 531p^3 + 1852p^4 + 3260p^5 + o(p^6) \tag{3.16}$$

and has nonzero coefficients for terms up to and including p^{59}. This equation confirms the combinatoric analysis, which was much easier to do in this case. For $p \ll \frac{59}{531}$, the second order term dominates and $C \approx 59$, like we found before.

The combinatoric analysis for R_2 assumes that a fixed number of syndromes s were gathered. Extracting $s = 3$ or more syndromes gives $p_0(p) \approx 31p^2$. The corresponding threshold estimate is $p_{th} \approx 0.032$. Again, this estimate does not include errors that could have propagated from prior recovery operations nor errors from waiting.

So we see that the recovery network for the three qubit code indeed has a threshold under bit-flip noise (or phase-flip noise). The threshold varies between 1.7% and

3.2% depending on the circuit implementation details. The latter threshold exceeds some experimentally realized quantum gates, so it may be possible to experimentally observe the improved reliability. Ultimately, the 3-qubit code may be a reasonable code for an active quantum memory in systems where either bit-flips or phase-flips dominate.

Chapter 4

Fault-Tolerance Thresholds are Experimentally Observable

The threshold for fault-tolerance is the maximum possible component failure probability at which a reliable circuit can be constructed despite errors, using systematic application of error-correction. The value of this threshold, p_{th}, is crucial to the design and realization of a quantum computer, because it dictates not just feasibility of an implementation proposal, but also how space, time, and energy resources must be marshaled to build a fault-tolerant system. Typically, p_{th} has been estimated using general theoretical techniques, but here, we present a method by which p_{th} could be determined using a careful set of laboratory experiments. We illustrate this method in detail using a set of numerical experiments, and identify the crucial problem – distinguishing between unrealistically optimistic *pseudo*-thresholds and the real thresholds which determine scalability.

Section 4.1 motivates our experimental study of fault-tolerance. Section 4.2 enumerates the criteria that quantum systems must satisfy in order to be scalably fault-tolerant. Some of these are strict criteria while others are assumptions that simplify threshold analysis. Section 4.3 reviews how to lift some of the simplifying assumptions, leading to an additional set of design considerations for engineering fault-tolerant quantum systems. These considerations highlight the complexity of estimating fault-tolerance thresholds for experimental systems. Section 4.4 presents numeri-

cal methods for evaluating fault-tolerance thresholds and verifies them against theoretical analysis. The numerical analysis suggests the existence of pseudo-thresholds, which we investigate in section 4.5. Section 4.6 concludes the chapter.

4.1 Introduction

Reliability will likely be a scarce resource for quantum computers, so experimental methods and engineering approaches are ultimately necessary to maximize the reliability of fault-tolerant quantum computers. Furthermore, as discussed in Chapter 3, the quantum computer is a practical concept only if some kind of quantum fault-tolerant gate can be realized experimentally. An experiment that determines parameters for which a fault-tolerant gate or set of gates is more reliable than a basic gate or corresponding set of basic gates is one meaningful prerequisite for realizing fault-tolerant quantum gates. Such an experiment also furnishes us with an estimate for the fault-tolerance threshold.

Analytical methods lead to definitive bounds on the threshold and offer existence proofs guaranteeing that fault-tolerant methods efficiently improve reliability. However, how can the threshold be experimentally estimated, bounded, or precisely determined? Of especial experimental interest are two sets of parameters: *reliable parameters*, which are the set of parameter values where an additional level of recursive simulation improves reliability of a gate, and *subthreshold parameters*, which are the set of parameter values below threshold. Reliable parameters might estimate the actual threshold, whereas subthreshold parameters are strictly below threshold.

In practice, the behavior of reliable parameters may be more important than subthreshold parameters because engineered systems may only use a few levels of recursive simulation. Asymptotic analysis ignore the regime of low concatenation. From an engineering perspective, experimentally determined reliable parameters are also useful because experiments let us release assumptions in analytical methods, such as exact self-similar replacement and simplified error propagation.

However, for many levels of recursive simulation, reliable parameters and sub-

threshold parameters are expected to correspond closely. In this regime beyond the reach of current simulation methods, where large-scale quantum computation is envisioned for quantum factorization, it is important to know subthreshold parameters.

This chapter shows how reliable parameters can be determined experimentally for a few levels of recursive simulation. Reliable parameters are believed to be estimates for subthreshold parameters. In the concluding sections of this chapter, we discuss the problem of pseudo-thresholds in greater detail and suggest how one might experimentally bound the set of subthreshold parameters.

4.2 Criteria for Scalable Fault-Tolerance

A *scalable quantum computer* is a physical system that can support coherent manipulation of an arbitrarily large number of qubits. However, a scalable quantum computer is not necessarily a useful quantum computer because noise may ruin the computation. The solution to this problem is to design a fault-tolerant quantum computer.

A *fault-tolerant quantum computer* is a computer designed in such a way that failures do not spread and can be periodically corrected. A *scalably fault-tolerant quantum computer* is a scalable, fault-tolerant quantum computer with a fault-tolerance threshold greater than experimental gate and operation fidelities. A scalably fault-tolerant quantum computer is necessary for applications such as factoring or large-scale quantum simulation, where the desired gate reliability may change for different inputs to the algorithm.

This section reviews two sets of criteria for scalably fault-tolerant quantum computing. The first set of criteria that must be met are scalability criteria, known as the DiVincenzo criteria. The second set of criteria are fault-tolerance criteria. The existence of a threshold implies a subset of the criteria in this second set.

4.2.1 DiVincenzo Criteria

The DiVincenzo criteria [Div00] state requirements for scalable quantum computing. These high-level criteria indicate when an experimental apparatus may be considered a quantum computer.

The DiVincenzo criteria state that a quantum computer must ...

1. ... *be physically scalable to an arbitrary number of well defined qubits.* A quantum computer must be scalable to a regime with interesting computational consequences, in a way that is only limited by engineering or economic constraints. Furthermore, the qubits in the quantum computer need to be addressable and coherently manipulable.

2. ... *be initializable to a well defined starting quantum state.* Without a high quality, standard initial state, input quantum states cannot be constructed.

3. ... *have long coherence times relative to gate times of a universal set of gates.* Very long relative coherence times allow an arbitrary, known, initial quantum state to be coherently transformed to an arbitrary final state using a universal set of gates.

4. ... *permit high quantum efficiency measurements on arbitrary single qubits.* High quality projective measurements of any qubit in the system are necessary to readout the outcome of a quantum computation.

Two additional DiVincenzo criteria define a quantum communication interface for a quantum computer. The DiVincenzo criteria for quantum communication state that a quantum computer with a quantum communications interface must ...

1. ... *interconvert stationary computing qubits and flying communication qubits.*

2. ... *faithfully transmit flying communication qubits.*

Quantum computers that satisfy these criteria can externally exchange quantum information with one another using "flying" qubits such as photons. Furthermore,

these criteria may allow fault-tolerant quantum computing with higher thresholds because quantum communication can independently distribute entanglement throughout a quantum computer. Distributed entanglement can facilitate gates between distant qubits.

4.2.2 Fault-Tolerance Criteria

The DiVincenzo criteria are necessary for a system to be a scalable quantum computer. However, a scalable quantum computer may be restricted to very brief or highly unreliable computations. Yet, quantum factoring of a significantly large input must run for days or months, so a system that is only scalable is inadequate. The system must also be fault-tolerant.

Fault-tolerance imposes several additional criteria on a scalable quantum computer [KL96, Got00, ABO99, Got02]. This subsection consolidates strict requirements for fault-tolerance and common assumptions that increase the fault-tolerance threshold. Strict requirements are necessary for a threshold to exist. Common assumptions, on the other hand, simplify analysis and lead to a higher fault-tolerance threshold, though the assumptions may be challenging or unphysical to meet.

A scalable fault-tolerant quantum computer does not have a threshold or does not compute reliably unless ...

1. ... *error rates are below threshold.* High error rates cause frequent gate and memory errors. If the rates are too high, even a fault-tolerant quantum computer cannot compute for very long before its quantum state is too corrupt to continue.

2. ... *long-range correlations are extremely weak in space and time.* The standard proofs assert that correlations must decay exponentially in space and time [ABO99]. Under some conditions, stronger correlations may be acceptable [TB04]. Of course, the fault-tolerant quantum computer should not be expected to cope with rare but inevitable failures affecting every qubit in the computer, i.e., failures caused by our sun engulfing the planet, for example.

3. … *error rates per qubit remain constant as a function of the number of qubits.* If this is not the case, then p depends on the number of qubits $N \propto n^L$ and the threshold may not exist.

4. … *control is massively parallel.* Serial recovery procedures R_L introduce a delay proportional to the total number of qubits N. Because N scales exponentially with L, the number of potentially memory failure locations scales this way as well, eliminating the threshold. However, parallel recovery procedures restore the threshold.

5. … *qubit erasure and reinitialization are almost in place.* If qubits must be initialized a distance d from their point of use, then they travel a distance d without the protection of quantum error-correction. If d is too great, high quality ancilla states cannot be prepared effectively for the recovery procedures R_L.

Assumptions simplify calculations of threshold estimates. Some of these assumptions are physical and others are not. In cases where several similar assumptions are possible, the assumptions stated here are the extreme forms of typical assumptions.

A scalable fault-tolerant quantum computer will have an increased threshold if …

1. … *qubits only become correlated when they directly interact.*

2. … *qubits are never lost.*

3. … *qubits never "leak" from the computational Hilbert space.*

4. … *measurement occurs rapidly and in place.*

5. … *classical information processing is fast and reliable.*

6. … *reliable ancilla are available in arbitrarily large numbers.*

7. … *nonlocal gates between distant qubits are as reliable as local gates.*

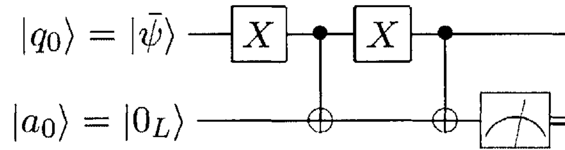


Figure 4-1: This circuit detects if $|\bar{\psi}\rangle$ has leaked from the computational Hilbert space. If this is the case, then the CNOT gates will act like the identity on qubit a_0 and the measurement outcome will be 0. Otherwise, the measurement outcome will be 1. If leakage occurs, the damaged qubit is reinitialized to $|0\rangle$.

4.3 Releasing Common Assumptions

Locality constraints and ancilla preparation constraints are among the most important, though until recently they have been neglected in fault-tolerance threshold estimates. Trade-offs between the two can dramatically influence the threshold and the reliability of fault-tolerant gates, as suggested in recent work [STD04, Kni04]. Hence, releasing these assumptions is critically important. This section reviews methods to release some common assumptions, including locality and ancilla preparation assumptions, to design realistically constrained fault-tolerant quantum computers.

First, we review the problems of leakage and qubit loss in subsection 4.3.1. Subsection 4.3.2 reviews ancilla preparation. Locality is reviewed briefly in subsection 4.3.3 but is largely postponed to Chapter 5, where a concrete local layout for the trapped-ion quantum computer is presented based on Gottesman's observations [Got00] and Steane's ancilla factory [Ste98].

4.3.1 Leakage and qubit loss

The Hilbert space of a qubit $\mathcal{H}$ may be smaller than the Hilbert space of the physical system $\mathcal{H}_{phys}$ representing the qubit. In this case, uncontrolled interactions may rotate the qubit into $\mathcal{H}_{phys} - \mathcal{H}$. This situation is called *leakage.*

The network shown in Figure 4-1 can detect leakage errors [Pre01]. Qubits that leak from the computational Hilbert space must be reinitialized to a state within that space, such as $|0\rangle$. Leakage may also be the result of qubit loss, which can be detected by this circuit as well. In these cases, error detection codes can be more efficient than

general codes [GBP97].

4.3.2 Ancilla Preparation

Fault-tolerant recovery operations, which occur very frequently in fault-tolerant circuits, essentially amount to preparation of highly entangled ancillary quantum states [Ste97]. Faulty preparation circuitry leads to damaged ancilla that are not useful for recovery operations. Hence, prepared states must be filtered by verification circuits, and those states that fail verification must be re-prepared. Preparing and verifying these states carefully can improve the fault-tolerant threshold by an order of magnitude or more [Rei04, Kni04].

The ancilla factory concept, introduced by Steane [Ste99, Ste98, Ste02], allows many such ancilla to be reliably prepared, verified, and maintained for recovery operations and for more complicated fault-tolerant gates. The ancilla factory pushes most of the complexity of fault-tolerant recovery into ancilla preparation in order to increase the fault-tolerance threshold and simplify the recovery operation itself.

The details of an ancilla factory can be complicated in practice because of the high degree of asynchronous parallelism required to prepare ancilla at a maximum rate while other parts of the quantum computer are executing an algorithm. The nondeterministic verification process further complicates detailed implementations of the ancilla factory because qubit movement patterns must change to accommodate failed verifications at all levels of the concatenated code hierarchy.

4.3.3 Locality and geometric constraints

Locality is one of the most important constraints imposed on realistic systems. Distant qubits involved in the same quantum gate must be moved near one another, and the distance between qubits depends on geometric constraints. Like quantum gates, movement processes are also expected to be error prone. Hence, the geometric arrangement of qubits in space, called the *layout*, must be taken into account when designing and evaluating fault-tolerant procedures.

A fault-tolerance threshold exists even with geometric constraints [ABO99]. Extra qubits prevent errors from spreading during swap operations, and a hierarchy of supporting ancilla allow intermediate error-correction steps during movement [Got00]. Careful threshold estimates place the local threshold an order of magnitude lower than the nonlocal threshold [STD04].

Chapter 5 considers layouts for trapped-ion quantum computers, so we defer further discussion to that chapter.

4.4 Experimentally Estimating Thresholds

We describe an experiment, demonstrated through numerical simulation, that estimates the fault-tolerance threshold. The experiment framed here gives a set of *reliable parameters* for which the component reliability improves with an additional level of recursive simulation. Reliable parameters estimated from experimentally accessible quantities lead to estimates of the fault-tolerance threshold.

One advantage of experiments is that they account for sources of error that are not typically counted in analytical threshold estimates, such as propagation, cancellation, and some asymmetry. One can gather detailed information about system reliability for complicated multidimensional system models as well, such as those that result from lifting common assumptions, like making gates local, or those that result from non-recursively constructed systems.

The numerical component of this method amounts to computing failure probabilities of fault-tolerant networks constructed from Clifford group gates and classical combinational logic gates. In particular, we study on the L-simulated fault-tolerant recovery operation R_L, though L-simulated Clifford group gates can also be numerically simulated using the methods from Chapter 2.

Subsection 4.4.1 broadly describes the threshold estimation method. Subsection 4.4.2 demonstrates the method for classical fault-tolerant circuits, where efficient simulation of the circuit itself is not an issue. Subsection 4.4.3 then describes how to apply the estimation method to quantum fault-tolerant circuits. Finally, subsec-

tion 4.4.4 demonstrates the method for the [[3, 1, 1]] and [[7, 1, 3]] quantum codes.

4.4.1 Experimental Method

Fault-tolerant classical and quantum circuits behave qualitatively like equation 3.5 when all of their components fail with some small probability p;

$$\frac{p_L(p)}{p_{th}} \leq \left(\frac{p}{p_{th}}\right)^{(t+1)^L}. \tag{4.1}$$

We would like to observe this behavior experimentally and estimate the fault-tolerance threshold $p_{th} \equiv 1/C$ from the coefficient of the leading order term of the failure probability.

Consider an experiment in which a quantum or classical circuit G is constructed from r different types of faulty basic components. Label each type of basic component with an integer i and write γ_i for the failure probability of that component. In the L-simulated circuit G_L, assume that each of the r types of L-simulated components is built up from basic components with *controllable failure probabilities.* The failure probability p_L of the L-simulated circuit G_L is a function of all r of the basic component failure probabilities $\vec{\gamma} = (\gamma_1, \ldots, \gamma_r) \in [0, 1]^r$. By establishing a map $\tilde{p} : [0, 1] \rightarrow [0, 1]^r$ from a single parameter p to $\vec{\gamma}$, the circuit failure probability $p_L(p) \equiv p_L \circ \tilde{p}(p)$ for each L traces out a curve with a form qualitatively similar to equation 3.5. For pairs of values l and l', $p_l(p)$ and $p_{l'}(p)$ may intersect at a value $p_{\tilde{p},(l,l')}$. This is the $\{\tilde{p}, (l, l')\}$ *crossing point.* Crossing points indicate regions of $[0, 1]^r$, given by $\tilde{p}(p)$, where recursion improves reliability – in other words, reliable parameters.

It is also possible to let $\tilde{p} : [0, 1]^{r'} \rightarrow [0, 1]^r$ be a function of $0 < r' < r$ parameters and consider surfaces with *crossing manifolds.* The choice of $\tilde{p}$ must be made carefully. Typically $\tilde{p}$ is simply an identity map onto a subset of coordinates γ_i. Whenever $\tilde{p}$ is understood, we omit it from the subscripts of crossing points.

For a fixed $\tilde{p}$ with $r' = 1$, circuits have a family of crossing points $\{\tilde{p}, (l, l')\} = \{\tilde{p}, (l'', l''')\}$ for all pairs $(l, l'), (l'', l''')$. From the basic theory that culminated in

equation 3.5, we expect these crossing points to all correspond closely to one another and to the fault-tolerance threshold.

How do we find $p_L(p)$? Suppose each noise operation in the circuit G_L is determined by the outcome of a random variable. Sampling each random variable selects a particular set of fault-paths within the circuit that may or may not be ultimately corrected. The circuit G_L fails or passes as a result of a particular set of fault-paths, so we model the circuit's behavior as a Bernoulli random variable with parameter p_L. As is well-known, a maximum likelihood estimate of the value of the parameter p_L can be constructed from a large number of samples of the corresponding Bernoulli random variable. The standard error σ_s of the estimate $\hat{p_L} = N_p/N_s$ is

$$\sigma_s = \frac{\sigma}{\sqrt{N_s}} \leq \frac{1}{\sqrt{4N_s}}, \tag{4.2}$$

where N_p is the number of times the circuit passes, N_s is the number of sampled events and $\sigma^2 = p_L(1 - p_L) \leq 1/4$ is the standard deviation of a Bernoulli random variable.

4.4.2 Classical Example

Figures 4-2 and 4-3 show results of a Monte-Carlo simulation of the TMR circuit in Figure 3-1 for a single wire ($T = 1$) and two composed wires ($T = 2$). Let γ_g be the identity gate failure probability, and let γ_m be the majority voter failure probability. For the numerical simulation, $\tilde{p}$ was chosen to map $p_0 \rightarrow (\gamma_g = p_0, \gamma_m = p_0)$. The number of samples $N_s = 10000$ corresponds to $\sigma_s \leq 0.005$. The level of recursive simulation varies from $L = 0$ to $L = 3$ in each plot. For each consecutive pair of values for L, we find a single crossing point. For $T = 1$, these are $p_{0,1} \approx 0.226$, $p_{1,2} \approx 0.164$, and $p_{2,3} \approx 0.153$. For $T = 2$ the $p_{1,2}$ and $p_{2,3}$ crossings decrease by approximately 0.02 due to propagating errors. The combinatoric threshold estimate for no input errors is $1/6 \approx 0.167$, which corresponds approximately with the crossing points observed for $T = 1$.

Numerical evidence suggests that $p_{1,2} \approx 0.065$ when $T = 1000$, which is even

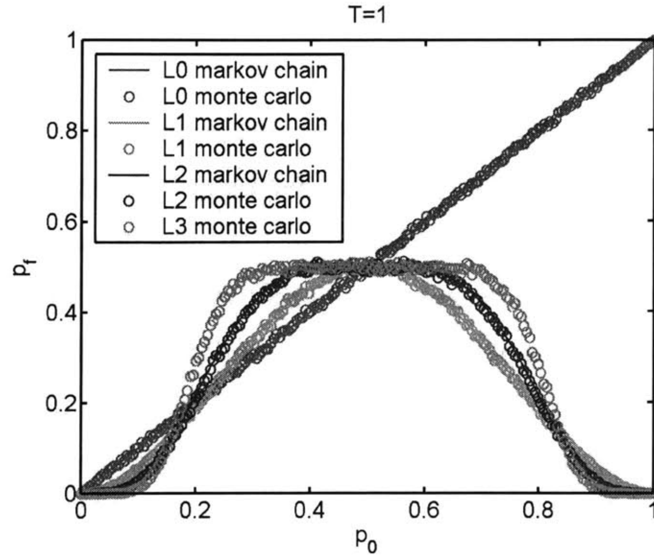


Figure 4-2: The failure probability of a fault-tolerant classical wire computed using Monte-Carlo simulation and evaluation of products of Markov chain transition matrices. The gate locations and voter locations both fail with probability p_0 initially (i.e., $\gamma_g = \gamma_m = p_0$. This plot shows the reliability of a single fault-tolerant wire ($T = 1$) and gives crossing points $p_{0,1} \approx 0.226$, $p_{1,2} \approx 0.164$, and $p_{2,3} \approx 0.153$.

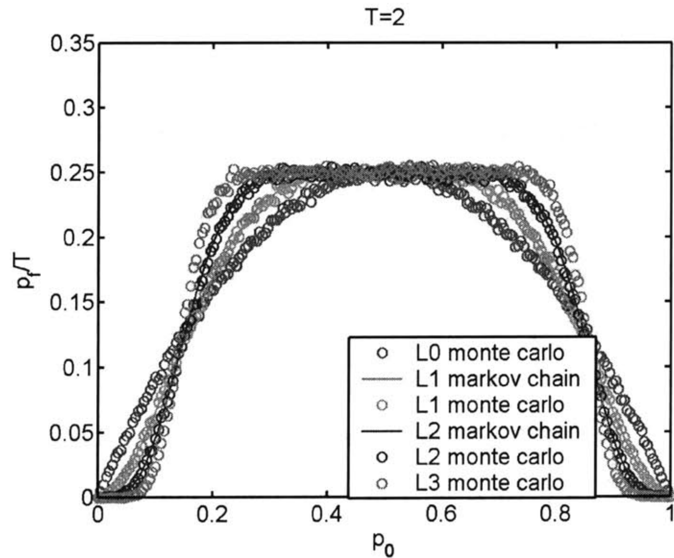


Figure 4-3: The failure probability of a fault-tolerant classical wire computed using Monte-Carlo simulation and evaluation of products of Markov chain transition matrices, like Figure 4-2. This plot shows the reliability of a single fault-tolerant wire computed by composing $T = 2$ fault-tolerant wires, then dividing the resulting failure probability by $T = 2$ to give an average failure probability. The crossing points $p_{1,2}$ and $p_{2,3}$ have shifted down by about 0.02 due to propagating errors.

less that $1/12 \approx 0.083$ expected from the combinatoric estimate (see equation 4.28). One explanation may be that some nondecodable outputs get corrected by the next fault-tolerant wire to result in a decodable but deviated output.

4.4.3 Evaluating quantum fault-tolerance

Fault-tolerant quantum computers spend a majority of time performing recovery operations and the background ancilla preparation for those recoveries. Recovery operations and ancilla preparation are two of the most complicated fault-tolerant procedures, only superseded in complexity by fault-tolerant gates in $\mathcal{C}_3 - \mathcal{C}_2$ and their corresponding ancilla preparation steps. However, recovery operations and corresponding ancilla preparation can be simulated efficiently within the stabilizer formalism (see Chapter 2). Hence, a majority of the basic computational steps in a fault-tolerant quantum computer can be simulated efficiently.

For this section, we compute crossing points for recursively simulated fault-tolerant recovery networks R_L. Steane has also numerically studied recovery networks using a stateless simulation method [Ste03b]. Storing a representation of the quantum state, as we do, carries advantages and disadvantages. On one hand, we can simulate arbitrary stabilizer circuits together with classical control and observe the state at any point, much like if we were using a digital computer design tool (see Chapter 6). Evolving a quantum state also lends confidence to our numerical results. On the other hand, measurement is a costly operation for stabilizer simulation algorithms and increases the simulation time significantly over stateless methods, even though stabilizer simulation algorithms run in polynomial time (when they give probabilistic measurement outcomes).

The noise model must also be efficiently simulated for selected operation elements of superoperators. Define a noise model $\mathcal{N}_{CG}$ based on probabilistic Clifford group gates and measurement of Pauli operators. The error operators $\mathcal{E}(\rho)$ have the form

$$\mathcal{E}(\rho) = \sum_k A_k \rho A_k^\dagger, \quad A_k = \sqrt{p_k} g_k, \quad g_k \in \mathcal{C}_2, \tag{4.3}$$

where the values p_k are understood to be probabilities. If measurement of Pauli elements M is allowed, forms such as

$$\mathcal{E}(\rho) = \frac{1}{4}\left[(I+M)\rho(I+M) + (I-M)\rho(I-M)\right], \tag{4.4}$$

or

$$\mathcal{E}(\rho) = \frac{1}{4}(I+M)\rho(I+M), \tag{4.5}$$

are possible, depending on if the measurement result is discarded or used to make a classically controlled correction, respectively. Typical error operators are tensor products of single qubit bit-flips, phase-flips, or depolarizing operators,

$$\mathcal{E}_X(\rho) = (1-p)\rho + pX\rho X, \tag{4.6}$$

$$\mathcal{E}_Z(\rho) = (1-p)\rho + pZ\rho Z, \tag{4.7}$$

$$\mathcal{E}_{XYZ}(\rho) = (1-p)\rho + \frac{p}{3}\left(X\rho X + Y\rho Y + Z\rho Z\right). \tag{4.8}$$

From this point forward, assume $\mathcal{E} = \mathcal{E}_{XYZ}^{\otimes m}$ for an m-qubit operation unless otherwise stated.

The $\mathcal{N}_{CG}$ noise model does not include systematic overrotation, for example, but is sufficient to verify quantum error-correcting properties and propagation of faults within a fault-tolerant stabilizer circuit. With current digital computer systems, stabilizer circuits of more than 1000 qubits, such as the $[[7,1,3]]$ quantum code at $L=3$ (1331 qubits) and the $[[3,1,1]]$ code at $L=4$ (625 qubits), can potentially be simulated under the $\mathcal{N}_{CG}$ noise model using a type of Monte-Carlo method in several months. The Monte-Carlo method samples a random variable, as in subsection 4.4.1, to select a particular operation element of each error operator.

4.4.4 Quantum Examples

This subsection applies the experimental method described in the previous subsection to the $[[3,1,1]]$ recovery network with bit-flip gate noise and the $[[7,1,3]]$ recovery

network with depolarizing gate noise. Both of these networks have a single parameter p_0 representing both single qubit and two qubit gate failure probabilities. This means that $\tilde{p}$ has been chosen to map p_0 to single qubit gate failure probability γ_1 and two qubit gate failure probability γ_2 while setting all other component failure probabilities γ_i to zero. We compare the numerical results to combinatoric threshold estimates.

3-qubit bit-flip code

The 3-qubit bit-flip code and its corresponding recovery network R_2 are described in Chapter 2 and Chapter 3. This subsection composes $T = 32$ L-simulated recovery networks and computes the failure probability p_f of the resulting circuit. We estimate the failure probability using about 4000 samples per data point to reduce the standard error to about 10^{-3} for the given range of probabilities p_0. Finally, for each L the failure probability p_f is divided by $T = 32$ to estimate the average failure probability of a single recovery operation.

Figure 4-4 shows numerical data for the 3-qubit bit-flip recovery network R_2. The data for three levels of recursion indicates crossing points $p_{1,2} \approx 0.035$ and $p_{2,3} \approx 0.037$. With coefficients rounded to three places, the least-squares polynomial fits shown in the figure are

$$\frac{p_1(p_0)}{T} \approx 2.00p_0^2 + 0.181p_0 - 0.001 \tag{4.9}$$

$$\frac{p_2(p_0)}{T} \approx 10^4(-1.61p_0^4 + 0.149p_0^3 - 0.00343p_0^2 + \tag{4.10}$$

$$(3.15 \times 10^{-5})p_0 - 9.41 \times 10^{-8}) \tag{4.11}$$

$$\frac{p_3(p_0)}{T} \approx 10^{11}(4.05p_0^8 - 0.784p_0^7 + (6.22 \times 10^{-2})p_0^6 \tag{4.12}$$

$$-(2.64 \times 10^{-3})p_0^5 + (6.54 \times 10^{-5})p_0^4 - (9.63 \times 10^{-7})p_0^3 \tag{4.13}$$

$$+(8.20 \times 10^{-9})p_0^2 - (3.66 \times 10^{-11})p_0 + 7 \times 10^{-14}). \tag{4.14}$$

These equations make clear that the basic theory described in Chapter 3 does not completely describe our data, for if it did, then $C \approx 2$ implies $p_{th} \approx 0.5$. The leading coefficient of p_2 gives a more reasonable result; $C^3 \approx 1.61 \times 10^4$ implies $p_{th} \approx$

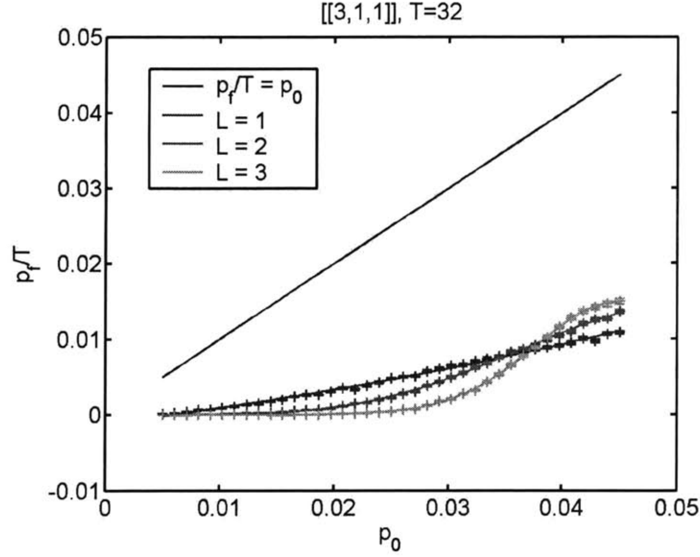


Figure 4-4: This figure shows the $p_1(p)$, $p_2(p)$, and $p_3(p)$ curves for the fault-tolerant recovery network of the 3-qubit bit-flip code. The points are numerically computed data, and the solid lines are polynomial fits to that data. The solid black line is the line $p_f/T = p_0$ included for reference. Independent bit-flip noise occurs prior to every gate in the network with probability p_0. The vertical axis is the average failure probability the L-simulated recoveries, computed by composing $T = 32$ recoveries and dividing the failure probability by $T = 32$. The numerically estimated threshold is $p_{th} \approx 0.035$.

0.0396. Finally, the leading coefficient of p_3 suggests a lower threshold of $p_{th} \approx 0.022$. Compare these results to the combinatoric threshold estimate $p_{th} \approx 0.032$, derived by counting failure locations for R_2 in Chapter 3.

7-qubit code

The $[[7, 1, 3]]$ recovery network we simulate is based closely on Preskill's network [Pre01]. Refer back to Figure 2-4 for an example of this kind of recovery network. Specifically, the network uses three cat state ancilla, collectively called the Shor ancilla,

$$|A_{\text{shor}}\rangle = \frac{1}{\sqrt{8}} \sum_{\text{even } v} |v\rangle. \tag{4.15}$$

to extract a syndrome bit for each of the 6 stabilizer generators. The ancilla are encoded into logical zero states using a noiseless network, then encoded into a logical

cat state using a noisy network with parity verification. If the first syndrome is trivial, then no recovery occurs. Otherwise, syndromes are extracted until two adjacent syndromes agree. The agreeing syndrome is used for error-correction. We assume that the probability for a memory error is negligible and that errors only occur during quantum gates.

To model the effects of error propagation into and out of the recovery network, we simulate the composition of T recovery networks. In this case, we choose $T = 32$. At the end of T recoveries, we check to see if the state can be perfectly decoded to a valid quantum codeword. If not, then that is counted as a failure of the T recoveries. The choice of T increases the failure probability to a numerically observable value and allows the input error distribution to converge to a typical distribution for this network.

Errors occur prior to single qubit and two qubit gates, both with probability p_0, in other words $\tilde{p} = (p_0, p_0)$. By randomly choosing operation elements from each error operator with the appropriate probabilities, we sample a random variable p_f representing the failure or success of the recovery network. For $L = 1$ (i.e., the bare recovery network on the 1-concatenated codewords), we collect roughly $120,000$ samples for each p_0 to estimate $p_f = p_1$. For $L = 2$ (i.e., the 1-simulated recovery network), we collect roughly $40,000$ samples for each p_0. Both keep the standard error on the order of 10^{-4}.

Finally, we plot p_f/T versus p_0 for $L = 1$ and $L = 2$ to show the mean recovery network failure probability. The value of p_0 where these curves cross is a type of pseudothreshold. We call this the memory pseudothreshold and claim that it approximates the fault-tolerance threshold. Intuitively, the fault-tolerance threshold ought to be lower than the memory pseudothreshold because there are additional failure locations in a fault-tolerant C_3 gate.

This embarrassingly parallel numerical experiment took slightly less than one month of real time on a cluster computer with 32 computing nodes. For $L = 3$, the time may be several months due to the exponential scaling of the quantum circuit with L.

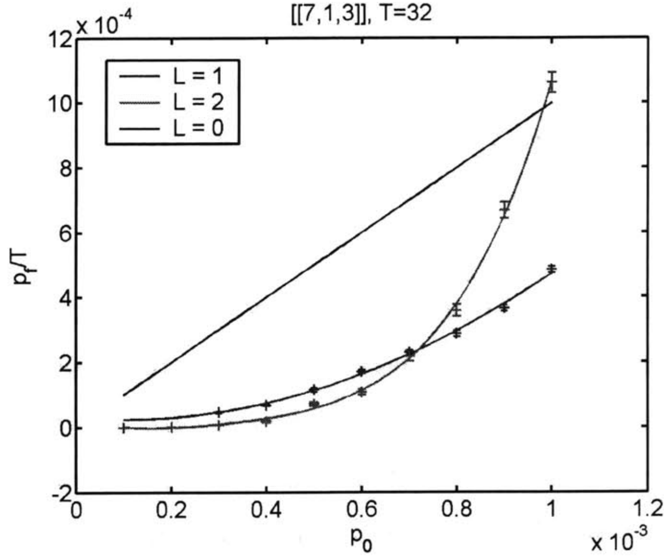


Figure 4-5: The $p_1(p)$ and $p_2(p)$ curves for the [[7, 1, 3]] fault-tolerant recovery network [Pre01]. Each point is numerically computed data, and the solid lines are polynomial fits to that data. The solid black line $p_f/T = p_0$ is included for reference. The numerically estimated threshold is $p_{th} \approx 7.2 \times 10^{-4}$.

Figure 4-5 shows the numerical results. The $p_1(p)$ and $p_2(p)$ curves cross in the neighborhood of $p_{th} \approx 7.2 \times 10^{-4}$. A least-squares polynomial fit to the $L = 1$ data with coefficients rounded to three places is

$$\frac{p_1(p_0)}{T} \approx 552p_0^2 - 0.108p_0 + 2.96 \times 10^{-5}. \tag{4.16}$$

Similarly, the fit to the $L = 2$ data is

$$\frac{p_2(p_0)}{T} \approx (3.09 \times 10^9)p_0^4 - (0.00364 \times 10^9)p_0^3 + 2000p_0^2 - 0.407p_0 + 0.00002. \tag{4.17}$$

On the other hand, counting arguments place the threshold without memory errors near $p_{th} \approx 6 \times 10^{-4}$ for a similar recovery network [Pre01].

Note that if the basic theory from Chapter 3 accurately describes this numerical experiment, then we observe that $C = 552$ based on the $L = 1$ polynomial fit. This implies a threshold $p_{th} \approx 1.8 \times 10^{-3}$. However, we find $C^3 = 1.68 \times 10^8$, but this is an order of magnitude lower than 3.09×10^9 observed from the $L = 2$ polynomial fit.

This latter value implies $C \approx 1457$ and $p_{th} \approx 6.9 \times 10^{-4}$.

4.5 The Pseudo-threshold problem

Numerical experiments can indeed give us approximations to the fault-tolerance threshold. However, the accuracy of these estimates is influenced by effects that the simple one-parameter model in section 3.2 does not capture. These effects cause the crossing points and adjusted coefficients (i.e., the $(2^L - 1)$-th roots) of leading order terms to differ for all pairs of values of L, making it difficult to absolutely identify or bound a threshold.

Two such effects can be observed in both classical and quantum circuits. First, the recursive simulation of many different types of basic components leads to a model with multiple parameters. Second, differing input error distributions across recursively simulated components perturbs the failure probabilities of those components.

This section presents a different model for fault-tolerance thresholds, again by example, that elaborates on results in [STD04]. We calculate how differing input error distributions and multiple component types influence our experiments, causing us to potentially identify *pseudo-thresholds* rather than true thresholds. Subsection 4.5.1 presents a new model for the fault-tolerant classical wire that clearly shows how pseudothresholds arise, and reveals a rich behavior that was not visible in the original model of fault-tolerance thresholds. Subsection 4.5.2 analyzes the 3-qubit bit-flip code using the tools developed in [STD04] to gauge how such effects influence our quantum threshold estimates.

4.5.1 Classical Pseudo-thresholds

The simple counting we have been doing in section 3.2 neglects input and output errors. How can a single error exiting the wire still be counted as a success, and how can we be sure the wire works correctly with a single input error? The counting analysis is not difficult to continue for these *deviated inputs*, but instead let us compute the product of transition matrices. The transition matrix method, which is analogous

to a positive operator analysis of quantum circuits, also allows us to quickly compute failure probabilities without counting and easily allows us to assign wires and voters different failure probabilities, a step that is absolutely necessary to arrive at the correct conclusion.

First, we need to describe a single noisy wire. The Markov chain transition matrix

$$B_p = \begin{pmatrix} 1-p & p \\ p & 1-p \end{pmatrix} \tag{4.18}$$

describes a single binary symmetric channel with parameter p. This matrix left multiplies a column vector $\vec{p} = (p_0, p_1)^t$ representing a discrete probability distribution over the possible input bit values 0 and 1, which maps it to a distribution of possible output bit values.

Next, we need transition matrices for decoders and majority voters. The transition matrix for a perfect TMR decoder has a single 1 in each column. Starting from column 0 at the left, if the binary representation of the column number decodes to 0 (resp. 1), a 1 is placed in the top (resp. bottom) row, like

$$D = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}. \tag{4.19}$$

Similarly, we can write down a matrix M for the fan-out and majority gates together in Figure 3-1. The matrix M is a zero matrix except for the first and last rows which equal the first and last rows of D.

These three transition matrices allow us to compute the failure probability $\gamma_g(\gamma_g, \gamma_m)$ of the fault-tolerant classical identity gate, where γ_g is the basic identity gate failure probability and γ_m is the basic majority voter failure probability. Given an arbitrary input error distribution

$$\vec{p} = (p_{000}, p_{001}, \ldots, p_{111})^t, \quad \sum_i p_i = 1, \tag{4.20}$$

we compute the probability of success or failure of the wire by composing transition

matrices,

$$(p_0, p_1)^t = DB_{\gamma_m}^{\otimes 3} M B_{\gamma_g}^{\otimes 3} \vec{p}. \tag{4.21}$$

The probability of failure of a fault-tolerant identity gate given no input error is

$$\gamma_g(\gamma_g, \gamma_m|0) = 3\gamma_g^2 - 2\gamma_g^3 + 3\gamma_m^2 - 18\gamma_g^2\gamma_m^2 + 12\gamma_g^3\gamma_m^2 - 2\gamma_m^3 + 12\gamma_g^2\gamma_m^3 - 8\gamma_g^3\gamma_m^3, \tag{4.22}$$

which checks against the counting analysis when $\gamma_g = \gamma_m$. For a uniformly distributed single bit input error, the probability of a fault-tolerant identity gate failure given a single input error is

$$\gamma_g(\gamma_g, \gamma_m|1) = 2\gamma_g - 3\gamma_g^2 + 2\gamma_g^2 - 12\gamma_g\gamma_m^2 + 18\gamma_g^2\gamma_m^2 - 12\gamma_g^3\gamma_m^2 - 2\gamma_m^3 + 8\gamma_g\gamma_m^3 - 12\gamma_g^2\gamma_m^3 + 8\gamma_g^3\gamma_m^3. \tag{4.23}$$

To compute the probability of failure when this wire is contained in a larger fault-tolerant computation, assume that all input errors are due to failing majority voters in a previous fault-tolerant wire,

$$\gamma_g(\gamma_g, \gamma_m|0,1) = \frac{\gamma_g(\gamma_g, \gamma_m|0)(1-\gamma_m)^3 + \gamma_g(\gamma_g, \gamma_m|1)(3\gamma_m(1-\gamma_m)^2)}{(1-\gamma_m)^3 + 3\gamma_m(1-\gamma_m)^2} \tag{4.24}$$

$$= (3\gamma_g^2 - 2\gamma_g^3 + 6\gamma_g\gamma_m - 12\gamma_g^2\gamma_m + 8\gamma_g^3\gamma_m + 3\gamma_m^2 - 18\gamma_g^2\gamma_m^2 \tag{4.25}$$

$$+ 12\gamma_g^3\gamma_m^2 + 4\gamma_m^3 - 36\gamma_g\gamma_m^3 + 84\gamma_g^2\gamma_m^3 - 56\gamma_g^3\gamma_m^3 - 4\gamma_m^4 \tag{4.26}$$

$$+ 24\gamma_g\gamma_m^4 - 48\gamma_g^2\gamma_m^4 + 32\gamma_g^3\gamma_m^4)/(1+2\gamma_m) \tag{4.27}$$

$$\leq 12\max(\gamma_g, \gamma_m)^2. \tag{4.28}$$

This equation gives the failure probability of a fault-tolerant gate assuming that the input is *correctable.* Let γ_{g0} and γ_{m0} be the basic component failure probabilities of the identity gate and majority voter, respectively. Then the final inequality tells us that $\gamma_g(\gamma_{g0}) \leq \gamma_{g0}$ if $\gamma_{g0} < 1/12$ and $\gamma_{g0} \geq \gamma_{m0}$, meaning that recursive simulation strictly improves the reliability of the fault-tolerant gate.

In order to complete the analysis, we must find a similar map $\gamma_m(\gamma_m)$ for the majority voter. At first glance, this map appears to equal $\gamma_g(\gamma_m, \gamma_m)$, but this is not

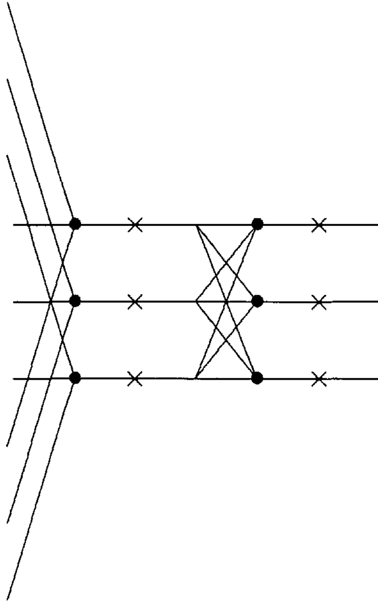


Figure 4-6: The construction for a fault-tolerant majority voter is identical to a fault-tolerant wire in Figure 3-1 except the first wire is replaced by a voter. If input errors are ignored, the wire and voter have the same failure probability. However, input errors cause the wire and voter to behave differently in recursive simulations.

true. Figure 4-6 shows a fault-tolerant majority voter. Let M_9 denote the $2^3 \times 2^9$ transition matrix for the first triplicated majority voting gate. The transition matrix of the entire circuit is

$$DB_{\gamma_m}^{\otimes 3} M B_{\gamma_m}^{\otimes 3} M_9. \tag{4.29}$$

As expected from the counting analysis, the failure probability of the voter without input errors $\gamma_m(\gamma_m|0)$ does equal $\gamma_g(\gamma_m, \gamma_m|0)$,

$$\gamma_m(\gamma_m|0) = 6\gamma_m^2 - 4\gamma_m^3 - 18\gamma_m^4 + 24\gamma_m^5 - 8\gamma_m^6. \tag{4.30}$$

However, the maps differ when input errors are modeled. There are 27 ways to distribute one input error uniformly on each of the three input blocks of the first triplicated majority voting gate, giving

$$\gamma_m(\gamma_m|1) = \frac{14}{9}\gamma_m + \frac{4}{3}\gamma_m^2 - \frac{92}{9}\gamma_m^3 + \frac{146}{9}\gamma_m^4 - \frac{40}{3}\gamma_m^5 + \frac{40}{9}\gamma_m^6. \tag{4.31}$$

All together,

$$\gamma_m(\gamma_m|0,1) = \frac{\gamma_m(\gamma_m|0)(1-\gamma_m)^9 + \gamma_m(\gamma_m|1)(27\gamma_m^3(1-\gamma_m)^6)}{(1-\gamma_m)^9 + 27\gamma_m^3(1-\gamma_m)^6} \quad (4.32)$$

$$= (6\gamma_m^2 - 22\gamma_m^3 + 54\gamma_m^4 + 96\gamma_m^5 - 406\gamma_m^6 + 552\gamma_m^7 \quad (4.33)$$

$$- 408\gamma_m^8 + 128\gamma_m^9)/(1 - 3\gamma_m + 3\gamma_m^2 + 26\gamma_m^3) \quad (4.34)$$

$$\leq 6\gamma_m^2 \quad (4.35)$$

Compare this result with equation 4.28. The majority gate is somewhat less sensitive to input errors because the errors are quickly corrected unless they conspire in some sense.

Because the fault-tolerant gate failure probability $\gamma_g(\gamma_g, \gamma_m)$ depends on the majority voter failure probability, the error rejecting property of the recursively simulated voter influences the recursively simulated identity gate as well. Specifically, the two maps $\gamma_g(\gamma_g, \gamma_m)$ and $\gamma_m(\gamma_m)$ are the coordinate functions of a two-parameter *probability map*

$$\vec{\gamma}(\vec{\gamma_0}) = \begin{pmatrix} \gamma_g(\gamma_{g0}, \gamma_{m0}|0,1) \\ \gamma_m(\gamma_{m0}|0,1) \end{pmatrix}, \quad (4.36)$$

that completely describes how recursive simulation changes the reliability of the TMR fault-tolerant wire, assuming that input errors are correctable. The initial probability vector $\vec{\gamma_0} = (\gamma_{g0}, \gamma_{m0})$ gives the failure probability of the basic identity gate γ_{g0} and the basic majority voter γ_{m0}.

The iterated probability map gives a flow in the space of identity gate and majority voter failure probabilities. Figure 4-7 shows the action of the probability map on the rectangle $[0, \frac{1}{2}] \times [0, \frac{1}{2}]$ for the first few iterations. This rectangle is mapped into itself by the probability map, and points in the rectangle appear to flow vertically away from the center of the rectangle.

Suppose that one of the two basic components is flawless. What are the fault-tolerance thresholds in this case, and how much do they differ? Choose an initial

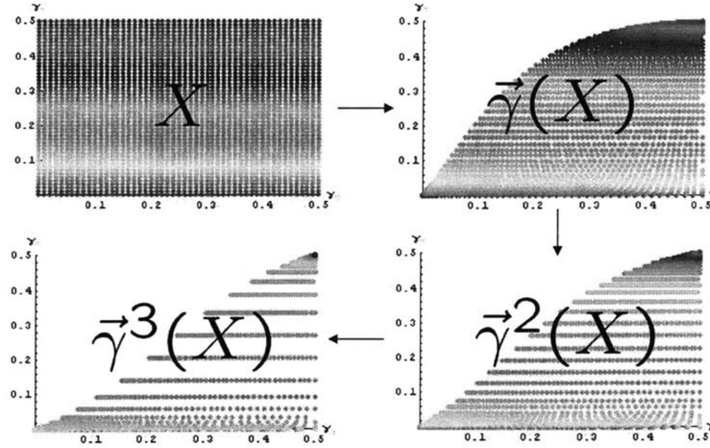


Figure 4-7: This sequence of graphs shows the image of the rectangle $R = [0, \frac{1}{2}] \times [0, \frac{1}{2}]$ under several iterations of the probability map given by equation 4.36. In particular, this rectangle is an invariant set, meaning $\vec{\gamma}(R) \subseteq R$, and the origin and the upper right hand corner are attracting fixed points.

probability vector $\vec{\alpha_{\gamma_g}} = (\gamma_g, 0)$ or $\vec{\beta_{\gamma_m}} = (0, \gamma_m)$, then the *gate threshold* is

$$\gamma_{g,th} = \sup \{\gamma_g \mid \vec{\gamma}^{\infty}(\vec{\alpha_{\gamma_g}}) = 0\}, \tag{4.37}$$

and the *voter threshold* is

$$\gamma_{m,th} = \sup \{\gamma_m \mid \vec{\gamma}^{\infty}(\gamma_{\vec{\beta_m}}) = 0\}. \tag{4.38}$$

The gate threshold, $\gamma_{g,th} = 0.5$, is the threshold of Figure 3-1 when the majority voters do not fail. The value of the gate threshold is obvious because the circuit reliability improves only if $3\gamma_g^2(1-\gamma_g) + (1-\gamma_g)^3 < \gamma_g$ in this case. The voter threshold, $\gamma_{m,th} \approx 0.175$, is the threshold of Figure 3-1 when the gates (wires) do not fail. This makes clear that noisy voters have the greatest influence on the threshold.

Now suppose that we were aware of the fact that voters and gates behave differently under recursive simulation, but the voters and gates failed with the same probability $p = \gamma_g = \gamma_m$. Let us call the threshold we find by setting all parameters

equal, the *diagonal threshold.* In this case, we find a diagonal threshold

$$p_{th} = \sup \{p \mid \vec{\gamma}^{\infty}(p) = 0\} \approx 0.175 \tag{4.39}$$

equal to the voter threshold. Again, this emphasizes the importance of the voters, which are analogous to quantum recovery networks.

A common way to calculate thresholds is to consider the first iterate alone and find its fixed point [Ste03b]. Using the probability map, however, the image is actually a two-dimensional vector. We need to define the *pseudo-threshold* p_{pt},

$$p_{pt} = \sup \{p \mid \ \| \vec{\gamma}(p, \ldots, p) \|_{\infty} < p\} \tag{4.40}$$

in order to make comparisons between the first iterate of a multiparameter map and the first iterate of a single parameter map. In this definition, $\| \cdot \|_{\infty}$ is the largest element of a finite dimensional vector.

There are many different ways to define a pseudo-threshold, depending on the chosen path through parameter space (i.e., $\tilde{p}$, see section 4.4.1) and the metric used to evaluate the first iterate ($\| \cdot \|_{\infty}$ in this case). Unlike one-dimensional maps where the first iterate determines the fixed point, which is the threshold as well, the first iterate can give considerably less information for maps with multiple parameters. For TMR, this pseudo-threshold $p_{pt} \approx 0.12$ differs from the actual threshold by about 0.055. As a fraction of the actual threshold, the difference is about 31%.

To conclude this section, we apply the basic theory of discrete dynamical systems to characterize all of the fixed points of equation 4.36 and numerically determine all points attracted to the origin (i.e., the basin of attraction of the origin) [Wig03]. Figure 4-8 summarizes these results. At each fixed point $(\gamma_{gf}, \gamma_{mf})$, the eigenvalues and eigenvectors of the differential

$$D\vec{\gamma}|_{(\gamma_{gf},\gamma_{mf})} = \left.\begin{pmatrix} \frac{\partial \gamma_g}{\partial \gamma_{g0}} & \frac{\partial \gamma_g}{\partial \gamma_{m0}} \\ \frac{\partial \gamma_m}{\partial \gamma_{g0}} & \frac{\partial \gamma_m}{\partial \gamma_{m0}} \end{pmatrix}\right|_{(\gamma_{gf},\gamma_{mf})} \tag{4.41}$$

determine the stability of the map at that point. Specifically, let $\{\lambda_{\vec{p},i}\}$ be the set of eigenvalues of the differential evaluated at the fixed point $\vec{p}$. If $\lambda_{\vec{p}} = \max \lambda_{\vec{p},i} < 1$ then $\vec{p}$ is an attracting fixed point. If $\lambda_{\vec{p}} > 1$ then $\vec{p}$ is a repelling fixed point (along some eigenvector). The TMR circuit has fixed 5 points $(0,0)$, $(\frac{1}{2},0)$, $(\frac{1}{2},\frac{1}{2})$, $(\frac{1}{2},0.17594\dots)$, and $(1,0)$. Three of these points, $(0,0)$, $(\frac{1}{2},\frac{1}{2})$, and $(1,0)$, are attracting fixed points.

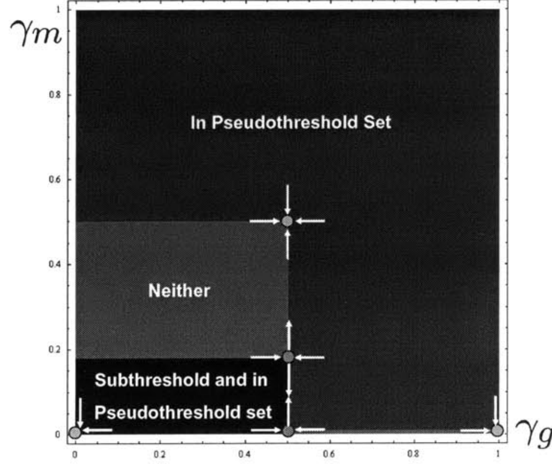


Figure 4-8: This plot characterizes the fixed points of the probability map given by equation 4.36. Green circles denote attractive fixed points, and red circles denote saddle points. The white arrows are eigenvectors of the linearized system at each fixed point. Ingoing arrows correspond to associated eigenvalues with magnitude less than 1 and outgoing arrows correspond to associated eigenvalues with magnitude greater than 1.

The interior of the basin of attraction of the origin S_0 is the interior of the rectangle $[0,\frac{1}{2}] \times [0, 0.17594\dots]$, as shown in Figure 4-8. Any point in this rectangle will converge to the origin. The interior of the plane $[0,\frac{1}{2}] \times [0,\frac{1}{2}]$ minus the rectangle $[0,\frac{1}{2}] \times [0.17594\dots,\frac{1}{2}]$ is the interior of the *pseudo-threshold set*

$$S_{pt} \equiv \{\vec{p} \mid \vec{\gamma}(\vec{p})_i < p_i \text{ for some } i\}. \tag{4.42}$$

The pseudo-threshold set contains those points that might appear to be below threshold based on projections of the first iterate onto the coordinate axes. This simple example suggests that S_{pt} can contain points outside of S_0, though common sense excludes those points in this example.

4.5.2 Quantum Pseudo-thresholds

An experiment that determines parameters for which a fault-tolerant gate is more reliable than a basic gate is one prerequisite for realizing fault-tolerant quantum gates. Yet is such an experiment sufficient to identify parameters that guarantee *scalable* fault-tolerance? The example in subsection 4.5.1 confirmed a negative answer to this question for classical circuits and clarified the concept of a pseudo-threshold [STD04], the component failure probability at which recursive simulation could improve reliability of one or more components but does not do so asymptotically.

This subsection follows [STD04] to estimate the probability map for quantum TMR. The goal is to give a simple example of pseudothreshold behavior in quantum circuits, provide a threshold estimate for error-correction using the TMR recovery network R_2, and quantify the gap between possible pseudo-thresholds and the actual threshold.

The probability map analysis requires some assumptions. There are five types of locations that fail with probability γ_l: single-qubit gates ($l = 1$), controlled-NOT gates ($l = 2$), controlled-NOT gates followed by measurement of the target qubit ($l = 2m$), preparation of a $|0\rangle$ state ($l = p$), and a waiting qubit ($l = w$). Measurements, CNOT's, and single qubit gates take the same amount of time, and classical computing takes negligible time. Error-correction is parallelized as much as possible to reduce the number of times qubits must wait. Ancilla qubits are prepared just in time during the previous syndrome extraction. Finally, we assume a very low error rate, so faults do not cancel and error-correction using the wrong syndrome does not occur. CNOT gate errors correspond to the operator

$$\mathcal{E}(\rho) = (1-p)\rho + \frac{p}{3}\left[IX\rho IX + XI\rho XI + XX\rho XX\right], \tag{4.43}$$

so there is an error on a given output qubit with probability $2p/3$.

Let β be the probability that the first syndrome is zero. The first syndrome is zero if no errors occur during ancilla preparation, no controlled-NOT gates fail in a way that introduces errors on the ancilla, no memory failures occur on the data that

can propagate to the ancilla, and there are no preexisting errors on the data. This statement gives a lower bound,

$$\beta \geq (1-\gamma_p)^2(1-2\gamma_2/3)(1-\gamma_2)(1-2\gamma_{2m}/3)^2(1-\gamma_w)\mathbb{P}(\text{no incoming errors}). \quad (4.44)$$

Incoming errors can be due to the prior transversal gate or the last error-correction. Consider the last error-correction alone to estimate the probability that there are no incoming errors, $\mathbb{P}(\text{no incoming errors})$. Suppose that the first syndrome is zero, then there are no incoming errors if there are no errors from the first syndrome extraction and if there are no errors while waiting for other parallel syndrome extractions to finish. On the other hand, if the first syndrome is nonzero, then there are no incoming errors if none of the syndrome extractions leave an error and if recovery leaves no error given $s' < s$ syndromes agree. These probabilities are

$$\mathbb{P}(\text{no } S_1 \text{ errors}) \geq (1-2\gamma_2/3)^2(1-2\gamma_{2m}/3)^2(1-\gamma_w)^5, \quad (4.45)$$

$$\mathbb{P}(\text{no waiting errors}) \geq (1-\gamma_w)^{9(s-1)}, \quad (4.46)$$

$$\mathbb{P}(\text{no } S_i \text{ errors for all } i) \approx \mathbb{P}(\text{no } S_1 \text{ errors})^s, \quad (4.47)$$

$$\mathbb{P}(\text{no recovery error}) = (1-\gamma_w)^2(1-\gamma_1). \quad (4.48)$$

The probability that the prior transversal gate does not fail is $(1 - \max(\{\gamma_i\}))^3$. Finally, take all of the expressions to be valid approximations and solve the linear equation

$$\beta = AB\left[\beta C + (1-\beta)D\right] \quad (4.49)$$

for β, where

$$A = (1-\gamma_p)^2(1-2/3\gamma_2)(1-\gamma_2)(1-2/3\gamma_{2m})^2(1-\gamma_w) \quad (4.50)$$

$$B = 1 - \max(\{\gamma_i\}))^3 \quad (4.51)$$

$$C = \mathbb{P}(\text{no } S_1 \text{ errors})\mathbb{P}(\text{no waiting errors}) \quad (4.52)$$

$$D = \mathbb{P}(\text{no } S_i \text{ errors for all } i)\mathbb{P}(\text{no recovery error } | s' \text{ agree}). \quad (4.53)$$

The vector $\vec{\gamma} = (\gamma_1, \gamma_2, \gamma_{2m}, \gamma_p, \gamma_w)$ characterizes the reliability of the quantum TMR circuit. We would now like to see how this vector changes when each location is recursively simulated. Let $\vec{\gamma}(\vec{\gamma_0})$ denote the map that gives the component failure probabilities after recursive simulation, and let the set of functions $\{\gamma_l(\vec{\gamma_0})\}$ be its coordinate functions, one for each type of location.

Before continuing, we need to define some expressions for estimating the failure probability of a set of locations. Let

$$\mathbb{P}(1^+ \in I, s_x) = 1 - (1 - \delta(I))^{N(I)} \tag{4.54}$$

be the probability of one or more errors originating from error source I when s_x syndromes are collected during error-correction. Table 4.1 lists error sources I together with their location failure probability $\delta(I)$ and location count $N(I)$. Similarly, let

$$\mathbb{P}(2^+ \in I, s_x) = 1 - (1 - \delta(I))^{N(I)} - N(I)\delta(I)(1 - \delta(I))^{N(I)-1} \tag{4.55}$$

be the probability of two or more errors originating from error source I when s_x syndromes are collected during error-correction.

SOURCE	δ	N
Fault in CNOT in S	$2\gamma_2/3$	$2s_x$
Fault in CNOT or measurement in S	$2\gamma_{2m}/3$	$2s_x$
Memory faults in S	γ_w	$4s_x$
Memory faults in R	γ_w	$2\delta_{s_x,s}$
Fault in gate or R	γ_1	$1\delta_{s_x,s}$
Memory fault when $s = 1$	γ_w	$9(s-1)\delta_{s_x,1}$
Encoded gate error in rect. type l	γ_l	3

Table 4.1: This table lists sources of error in the quantum TMR circuit. The first column is the error source which is a subset of locations in the circuit. The second column is the failure probability δ associated with an individual error event. The third column is the number of locations N in the error source.

The probability of failure of a location $l = 1$, w, or p is

$$\gamma_l(\vec{\gamma}) = \beta F_l[1] + (1 - \beta) F_l[s] \tag{4.56}$$

where

$$F_l[s_x] \approx \sum_{I>J} \mathbb{P}(1^+ \in I, s_x)\mathbb{P}(1^+ \in J, s_x) + \sum_{I} \mathbb{P}(2^+ \in I, s_x) \tag{4.57}$$

is the probability of two or more errors in a recursively simulated location of type l given that s_x syndromes were extracted.

Similarly, the probability of failure of a location $l = 2$ or $2m$ is

$$\gamma_l(\vec{\gamma}) = \beta^2 F[1,1] + 2\beta(1-\beta)F[1,s] + (1-\beta)^2 F[s,s] \tag{4.58}$$

assuming that $F[1,s] = F[s,1]$. In this expression,

$$F[a,b] \approx \mathbb{P}(2^+ \in G) + 3\gamma_l(1-\gamma_l)^2 \sum_{I \neq G} \mathbb{P}(1^+ \in I, a, b) + (1-\gamma_2)^3 (F_l[a] + F_l[b]) \tag{4.59}$$

is the probability of two or more errors in a recursively simulated location $l = 2$ or $l = 2m$ given that a syndromes were extracted for error-correction of the control qubit and b syndromes were extracted for error-correction of the target qubit. G is the transversal CNOT gate as an error source.

The coordinate maps $\gamma_l(\vec{\gamma}_0)$ completely determine an approximation to the probability map $\vec{\gamma}(\vec{\gamma}_0)$ for quantum TMR. This 5-dimensional probability map is more difficult to visualize than the 2-dimensional map for classical TMR. Instead, we numerically compute threshold results using definitions analogous to the classical probability map definitions.

One measure of the threshold for quantum TMR is the *diagonal threshold*

$$\gamma_{th} = \sup\{p \mid \vec{\gamma}^{\infty}(p, \ldots, p) = 0\}, \tag{4.60}$$

which is defined like the classical case. The diagonal threshold considers starting probability vectors along the ray leaving the origin equidistant to all of the coordinate axes. For quantum TMR, the diagonal threshold is $\gamma_{th} \approx 0.0022$ including wait errors, and $\gamma_{th} \approx 0.009$ without wait failures but including error-correction at wait locations.

Table 4.2 lists all of the location thresholds and pseudothresholds for the quantum

TMR circuit. A *location threshold* for location type l is the failure probability

$$\gamma_{l,th} = \sup\{\gamma_l \mid \vec{\gamma}^{\infty}(0,0,\ldots,\gamma_l,0,\ldots,0) = 0\} \tag{4.61}$$

that corresponds to choosing an initial point γ_l and setting $\gamma_{else} = 0$. A location threshold is the threshold when only a particular type of location is faulty. A *location pseudothreshold* for location type l is the failure probability

$$\gamma_{l,pt} = \sup\{\gamma_l \mid \vec{\gamma}(0,0,\ldots,\gamma_l,0,\ldots,0)_l < \gamma_l\}. \tag{4.62}$$

A location threshold estimate is based only on the l-th coordinate function of the first iterate. One might mistake a location pseudothreshold for a location threshold if the threshold estimate is based solely on a 1-simulated component.

γ_l	$\gamma_{l,th}$	$\gamma_{l,pt}$
γ_1	0.26	0.26
γ_2	0.0159	0.026
γ_{2m}	0.032	0.059
γ_w	0.0028	0.0036
γ_p	0.5	0.5

Table 4.2: This table lists the location thresholds and pseudothresholds for the quantum TMR circuit. Wait errors have the lowest location threshold, and CNOT gates have the largest difference between thresholds and pseudothresholds.

4.6 Conclusion

We have developed a new experimental method to estimate fault-tolerance thresholds for quantum fault-tolerant circuits based on stabilizer simulation. This method allows us to compute reliable parameters that estimate the threshold for potentially complicated system models within the stabilizer formalism. This new method for evaluating fault-tolerant systems opens up the possibility for a type of quantum computer architecture design tool, which we revisit in Chapter 6.

These experiments identify several problems for future study. How can we distinguish thresholds from pseudothresholds? What is the largest difference between a pseudothreshold and a threshold? Is there an experimental method by which actual fault-tolerance thresholds can be exactly determined or tightly bounded? While we cannot answer these questions completely, we outline in conclusion a possible approach to the question of experimentally bounding the threshold.

We have already shown that stabilizer circuit reliability can be evaluated numerically. This means that we can compute the actual probability map $\vec{\gamma}$, as defined in Chapter 3, provided that each coordinate map γ_i is associated with a stabilizer quantum circuit C_i.

For each C_i, the input error distribution and the input quantum state is unknown when C_i is taken out of its context within a larger quantum algorithm. Suppose we assume that these inputs must be valid in the sense that they ought to decode correctly. For a given i, there must be a set of decodable inputs that minimize γ_i and a set of decodable inputs that maximize γ_i, so each γ_i can be bounded numerically given these inputs. This is the case for all γ_i, so we can bound all of the coordinates of $\vec{\gamma}(\vec{\gamma})$.

To compute the threshold, we must determine the boundary of the set of points attracted to the origin by successive iterations of $\vec{\gamma}(\vec{\gamma})$. Naively, simply iterate $\vec{\gamma}(\vec{\gamma})$ numerically for the "best" and "worst" decodable inputs. Take p_{th} to be the maximum coordinate of all points within the basin of attraction. If the input errors and input states were chosen correctly for each location and for each replacement, then the resulting experimentally determined threshold would be exact. However, perhaps the "best" and "worst" maps yield a numerical bound on p_{th}, or at least a good estimate.

Chapter 5

Threshold for Trapped-Ion Quantum Computers

Trapped-ion quantum information processing experiments are the most capable candidates for quantum computing at present. In fact, recent experimental efforts suggest that this system may also be the most scalable [CSB+04, RHR+04]. However, the precise fault-tolerance threshold for computing with trapped-ions is unknown because the threshold depends on many system level assumptions and on the choice of fault-tolerance construction. Many architectural trade-offs can increase or decrease the threshold. Furthermore, the threshold can be interpreted as a fidelity goal for experimental quantum gates and other operations, so knowledge of the threshold under more realistic models can guide experimental efforts. In this chapter, we develop a system model of the trapped-ion quantum computer, suggest several layouts for large-scale ion traps, and predict a fault-tolerance threshold for trapped-ion quantum computing with the $[[7,1,3]]$ code under particular models and assumptions. The insights gained in Chapters 3 and 4 give us a natural platform from which to build the ion-trap system model and estimate the fault-tolerance threshold based on reliable parameters.

Section 5.1 begins by reviewing trapped-ion quantum computing at a high level and surveying experiments that demonstrate elements of a trapped-ion quantum computer. Section 5.2 defines a stabilizer simulation model of large-scale trapped-ion

quantum computers. Within this section, we give two concrete layouts with desirable fault-tolerance properties. Section 5.4 presents simulation results estimating the threshold for fault-tolerant trapped-ion quantum computing within our model. Finally, section 5.5 suggests directions in which this work can be extended.

5.1 Trapped-Ion Quantum Information Processing

Trapped atomic ions behave like elementary quantum systems that are well isolated from their environment, yet they can be precisely controlled. Controlled state transitions in trapped ions have been used in precision timekeeping applications for decades. Laser cooling techniques can bring trapped atoms nearly to rest, creating some of the coldest matter known to humankind. More recently, experiments have demonstrated fundamental quantum logic gates and the essential elements of scalable quantum information processing.

We begin by reviewing the current state of trapped-ion quantum information processing, focusing on experimental work carried out by NIST (Boulder) and the University of Innsbruck. Without going into great physical detail, the brief review describes quantum gates, measurements, and techniques for moving ion-qubits.

5.1.1 Ion-qubits and RF Traps

Trapped-ion quantum computation, as initially proposed by Cirac and Zoller [CZ95], uses a number of atomic ions trapped in a linear RF trap that can interact with laser beams to quantum compute. Each qubit is identified with two internal electronic or nuclear states of an ion. For example, the $^{40}\mathrm{Ca}^{+}$ ions used in experiments at Innsbruck identify $|0\rangle$ with the $4^2S_{1/2}$ ground state $|g\rangle$ and $|1\rangle$ with the $3^2D_{5/2}$ metastable excited state $|e\rangle$. Two or more ions can be contained in a single trap, where they couple to each other through Coulomb repulsion, forming a linear chain of ions called a Coulomb crystal. The vibrational modes of this chain provide a qubit-qubit interaction. Single qubit rotations and the qubit-qubit interaction yield a universal set of quantum gates for quantum computation, as will be discussed in subsection 5.1.2.

The Cirac-Zoller proposal does not scale to large numbers of qubits. As the length of the ion chain increases, the vibrational modes become progressively harder to identify [RBKD+02], decreasing the gate fidelities. These modes also couple more strongly to ambient fields, increasing the heating rate and, hence, the dephasing rate.

Kielpinski, Wineland, and Monroe propose a scalable extension of the Cirac-Zoller proposal, using a network of interconnected ion traps [KMW02]. This approach to scaling the ion-trap was originally proposed by NIST [DMW+98]. Multiple traps allow for smaller linear ion chains, and thus greater control over logic operations. Furthermore, externally adjusted static electrode voltages move ions between traps within the network, potentially allowing coherent manipulation of a large number of ions. This kind of trap network is called a quantum charge-coupled device (QCCD).

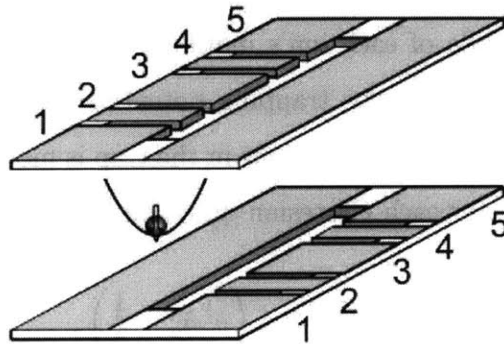


Figure 5-1: Schematic of the ion trap used in ion shuttling experiments at NIST Boulder [RBKD+02], courtesy of David Wineland. Individual ions are trapped near electrodes 2 and 4. Ions can be moved by adjusting the static potentials on electrodes 1, 3, and 5.

Current alumina micro-machining techniques have realized QCCD traps. Figure 5-1 is a schematic of a dual trap at NIST [RBKD+02]. Individual ions are trapped in regions near electrodes 2 and 4. Slowly varying DC potentials on the other electrodes move ions between the two trapping regions. Subsection 5.1.3 discusses this experiment in more detail.

Madsen et al have proposed ion-trap arrays that are constructed using semiconductor materials [MWD+04]. Quantum information processing with semiconductor traps

may be more experimentally challenging but has the benefit of combining the scalability of semiconductor fabrication processes with the quantum control techniques of atomic physics. More recently, Hensinger et al propose three-layer T-junctions to allow trapped ions to turn corners within a large network of traps [WDM+04].

5.1.2 Gates and Measurement

This brief discussion of gates and measurements in trapped-ion quantum computers is based on recent reviews [SB02, DMW+98].

A potential established by the trapping electrodes confines the ions. The trap potential is typically tight in the radial direction, so the N ions are trapped in a linear configuration, and can indexed from left to right by integers. This is referred to as a linear ion chain, so if N ions couple to one another vibrationally, we say they are chained. The amplitude of each ion's motion is also sufficiently small, because we assume the ions are cold, so the trapping potential is approximately harmonic. The quantized vibrational motion of the ions in the trap is modeled by $3N$ uncoupled harmonic oscillators, one for each Cartesian direction of each ion,

$$\hat{H}_{\text{bus}} = \sum_{\alpha=1}^{3N} \hbar\nu_\alpha \left(\hat{a}_\alpha^\dagger \hat{a}_\alpha + \frac{1}{2} \right), \tag{5.1}$$

where $\hbar$ is Planck's constant, ν_α is the normal frequency of the mode labeled α, and $\hat{a}_\alpha$, $\hat{a}_\alpha^\dagger$ are the annihilation and creation operators, respectively. The $\{|n\rangle, |n+1\rangle\}$ manifold of one of the low-order longitudinal modes is selected as a "bus" qubit to mediate interactions between ions, where $n = 0$ in the Cirac-Zoller scheme.

Each ion's internal qubit $\{|e\rangle, |g\rangle\}$ is represented by the Hamiltonian

$$\hat{H}_{\text{int}} = \frac{\hbar\omega_0}{2}\sigma_z + \frac{E_e + E_g}{2}\mathbb{I}, \tag{5.2}$$

where σ_z is the Pauli z-operator, $\mathbb{I}$ is the identity operator, and $\omega_0 = \frac{E_e - E_g}{\hbar}$ is the angular frequency of the qubit transition, given by the difference in energy between the ground and excited state. Shifting the energy origin, the total Hamiltonian for

an ion qubit (indexed by integer j) and a single motional "bus" mode with frequency ν is

$$\hat{H}_0 = \frac{\hbar\omega_0}{2}\sigma_{zj} + \hbar\nu\hat{a}^\dagger\hat{a}. \tag{5.3}$$

Laser beams of specific duration, power, and phase are single qubit quantum gates when applied on resonance to a particular ion. Assume beams can be steered to individually address ions at any location in the trap [NDR+99]. The laser's electric field is a plane wave,

$$\mathbf{E}(t, \mathbf{q}) = E_0 \boldsymbol{\epsilon} \cos(\omega_L t - \boldsymbol{\kappa} \cdot \mathbf{q} + \phi), \tag{5.4}$$

that interacts with the ion through dipolar coupling, in the simplest case. In this expression, E_0 is the amplitude of wave, $\boldsymbol{\epsilon}$ is the polarization vector of the electric field, ω_L is the angular frequency of wave, t is time, $\boldsymbol{\kappa}$ is the wave vector which has magnitude $|\boldsymbol{\kappa}| = \frac{2\pi}{\lambda_L}$, λ_L is the free-space wavelength, $\mathbf{q}$ is the position vector, and ϕ is a phase shift. The interaction Hamiltonian for dipolar coupling is

$$V = -q_e \mathbf{r}_j \cdot \mathbf{E}(t, \mathbf{q}_j) \tag{5.5}$$

where q_e is the magnitude of an elementary charge, $\mathbf{r}_j$ is the (internal) position operator of the valence electron of the jth ion, and $\mathbf{q}_j$ is the (external) position operator of the jth ion.

The ion vibrational motion is typically cooled to the *Lamb-Dicke regime* as a prerequisite for applying quantum gates. The Lamb-Dicke regime corresponds to the physical situation where the spatial extent of the ion motion is much smaller than the laser wavelength. The ion spontaneously emits mostly on carrier because the recoil energy is much smaller than the energy of a vibrational quanta. Sideband cooling techniques applied in the Lamb-Dicke regime can cool the ion motion to the ground

state. The formal conditions, collectively called the *Lamb-Dicke limit*, are

$$\eta(\langle n\rangle + \frac{1}{2})^{1/2} \ll 1, \quad (5.6)$$

$$\eta^2/2 \ll 1, \quad (5.7)$$

where $\eta = \kappa k z_0$ is the *Lamb-Dicke parameter*, $\langle n\rangle$ is the average number of phonons in the selected bus mode, κ is a parameter that depends the selected bus mode, and k is the magnitude of the laser wave vector. The distance $z_0 = \left(\frac{\hbar}{2m\omega_z}\right)^{1/2}$ is the extent of the ion's ground state wave function found from the expectation of the position operator, where m is the ion mass and ω_z is the longitudinal trap frequency (which corresponds to ν in equation 5.3). One calculation for $^{40}Ca^+$ ions gives $\eta \approx 0.06$ [SB02], for which the Lamb-Dicke limit implies that $\langle n\rangle \ll 200$. For $\langle n\rangle > 200$, the ion is too warm for quantum computation.

The ion-laser interaction Hamiltonian V is not obviously useful for implementing quantum gates until it is written in a different form. This well-known formal manipulation involves a sequence of approximations such as the rotating wave approximation, the weak-coupling approximation, and application of the Lamb-Dicke limit [SB02]. When the laser is on-resonance, i.e. $\omega_L = \omega_0$, the final interaction Hamiltonian is

$$\mathcal{A} = \sum_{n=0}^{\infty} \cos(\frac{|A_n|t}{2})(|e\rangle\langle e| \otimes |n\rangle\langle n| + |g\rangle\langle g| \otimes |n\rangle\langle n|) \quad (5.8)$$

$$- i\sum_{n=0}^{\infty} \sin(\frac{|A_n|t}{2})(|e\rangle\langle g| \otimes |n\rangle\langle n|e^{-i\phi} + |g\rangle\langle e| \otimes |n\rangle\langle n|e^{i\phi}). \quad (5.9)$$

The laser intensity and dipole matrix elements in equation 5.5 determine the Rabi frequencies A_n, and ϕ is the phase of the laser. The effect of *carrier* excitation on the level populations is illustrated in Figure 5-2.

Similarly, when the laser is tuned to $\omega_L = \omega_0 - \nu$, the interaction Hamiltonian

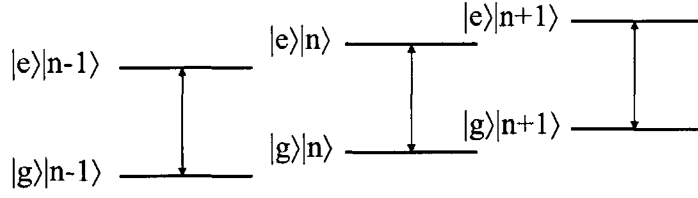


Figure 5-2: Energy levels of the atom-oscillator system together with carrier-driven transitions ($\omega_L = \omega_0$). Carrier-driven transitions rotate the qubit to desired superposition states.

becomes

$$\mathcal{B} = \sum_{n=0}^{\infty} \cos(\frac{|B_n|t}{2})(|e\rangle\langle e| \otimes |n\rangle\langle n| + |g\rangle\langle g| \otimes |n+1\rangle\langle n+1|) \tag{5.10}$$

$$- i \sum_{n=0}^{\infty} \sin(\frac{|B_n|t}{2})(|e\rangle\langle g| \otimes |n\rangle\langle n+1|e^{-i\bar{\phi}} + |g\rangle\langle e| \otimes |n+1\rangle\langle n|e^{i\bar{\phi}}). \tag{5.11}$$

This causes transitions on the first *red sideband*, as illustrated in Figure 5-4. Finally, tuning to $\omega_L = \omega_0 + \nu$ causes transitions on the first *blue sideband*, shown in Figure 5-3. Again, B_n is determined by laser intensity, the dipole matrix elements, and the Lamb-Dicke parameter.

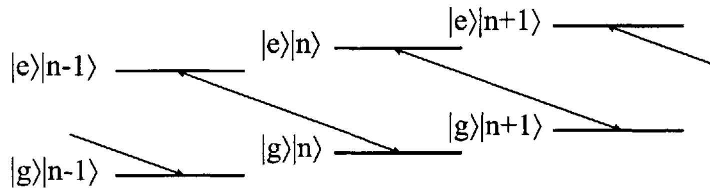


Figure 5-3: Energy levels of the atom-oscillator system together with red-driven transitions. Red-driven transitions change the populations of the qubit energy levels and the oscillator energy levels simultaneously. The oscillator loses one quanta of vibrational energy in a transition on the red sideband.

Single qubit rotations of the ion's internal state in the $\{|g\rangle, |e\rangle\}$ basis are repre-

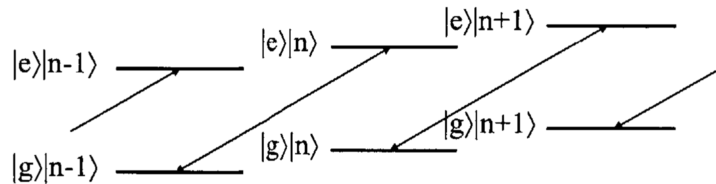


Figure 5-4: Energy levels of the atom-oscillator system together with blue-driven transitions. Blue-driven transitions change the populations of the qubit energy levels and the oscillator energy levels simultaneously. The oscillator gains one quanta of vibrational energy in a transition on the blue sideband.

sented by operators

$$\mathcal{R}(\theta,\phi) = \begin{pmatrix} \cos(\frac{\theta}{2}) & e^{i\phi}\sin(\frac{\theta}{2}) \\ -e^{-i\phi}\sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{pmatrix}, \tag{5.12}$$

where θ is the angle of rotation and ϕ is the relative phase shift of the ground and excited states. Equation 5.8 directly implements a single qubit rotation. Laser intensity, phase, and duration determine the angles θ and ϕ.

Given that we know how to perform arbitrary single qubit rotations, universal quantum computation only requires a two qubit entangling gate, such as the controlled-NOT (CNOT). There are many ways to perform CNOT gates with trapped-ions [CZ95, SM99, SM00, Ste03a], but all of them can be implemented using carrier, red, and blue sideband pulses of varying intensity, duration, and relative phase. The Cirac-Zoller and Molmer-Sorensen gates have been implemented successfully in laboratories [CDB+95, DDV+03, SKHR+03, SKHG+03].

The electron shelving method accomplishes reliable single qubit projective measurement in the computational basis $\{|g\rangle, |e\rangle\}$. An intense laser excites a transition that transfers population between $|g\rangle$ and a fast-decaying auxiliary level $|aux\rangle$. If the qubit collapses to the ground state, that population is transferred to the $|aux\rangle$ level. The auxiliary level spontaneously emits a photon, decaying back to $|g\rangle$. As long as the cycling transition is driven, the presence or absence of fluorescence at $\omega_c = (E_{aux} - E_g)/\hbar$ determines if the post-measurement state is $|g\rangle$ or $|e\rangle$, respec-

tively. However, if the qubit collapses to $|e\rangle$, the qubit remains in that state during the readout process and no photons are emitted.

5.1.3 Moving Ions between Traps

Quantum information needs to be communicated across a quantum computer, either directly or indirectly, because quantum algorithms interact and entangle a majority of the qubits involved in the algorithm. Ion-trap quantum computing differs from other physical systems because qubit swapping can be implemented using either laser pulses (swap gates) or *ballistic transport*. The latter method modulates static electrode voltages to push mobile ions from trap to trap. This method is a critical feature of the QCCD proposal because ions no longer have to be in the same trap for the duration of the computation.

Ballistic transport is not only advantageous for scaling, but also for designing quantum fault-tolerant experiments. Ballistic transport may be more reliable than swap gates. Ions can be separated from one another during measurement to reduce scattering errors induced by the fluorescing ion [LMBK$^+$04]. Extra swapping qubits will not be needed in the design of a fault-tolerant system with full ancilla support structures, reducing the number of qubits by more than a factor of two (see section 5.3).

Ballistic ion transport experiments at NIST have used the trap illustrated in Figure 5-1 [RBKD$^+$02]. Modulated static voltages on trapping electrodes 1, 3, and 5 shuttled a single $^9Be^+$ ion back and forth between traps 2 and 4. The data corresponds to 10^6 transfers over the $1.2mm$ distance. Each transfer took about $50\mu s$ and, in one particular experiment, occurred between each pulse of a spin-echo experiment. The resulting interference fringe contrast for two transfers, $96.8 \pm 0.5\%$, was due to imperfections in state preparation, detection, and the spin-echo pulses, rather than from environmental influences. No ion loss was ever observed as a result of the transfer.

Ions shuttled by ballistic transport in the NIST experiment with average velocity $0.024\text{mm}/\mu\text{s}$ heated at a rate of about $8 \times 10^{-6}\text{quanta}/\mu\text{m}$. This heating presents a

challenge because qubit-qubit gates such as the controlled-NOT degrade when acting on hot ions. Hence, ions must be recooled periodically by a sequence of recooling laser pulses on blue sideband transitions.

The cooling laser can change the state of the qubit, so ions must be cooled *sympathetically* instead. Sympathetic cooling couples an extra ion of a different species to the target ion. Cooling pulses applied to the extra ion no longer badly decohere the target ion because the optical transitions are at different frequencies [BDS+03, KKM+00].

The NIST experiment also studied separating and joining linear chains of ions between two traps, A and B, 1.2mm apart. Two ions confined in trap A were separated into traps A and B, then brought back together into trap A. This required several steps: laser cooling in trap A, trap parameter adjustment, and static voltage ramping. The entire process required 10ms. Discrete voltage steps during the ramping process, as well as other imprecisions in control, caused motional mode heating. The splitting process left an ion in each trap 95% of the time, and a mean on-axis motional mode population of 140 ± 70 quanta assuming a thermal distribution. More recent experiments have a success rate greater than 99% and heating of about 1 quanta in the center of mass mode and 0 in the stretch mode. The separation time is also reduced to around 1ms [CSB+04].

5.1.4 Noise and other imperfections

Noise limits the fidelity of operations within an ion trap quantum computer, and can do so by influencing the ion motion, influencing the ion qubit, or by systematically altering the parameters of quantum gates. The experimental issues surrounding noise in ion traps are discussed in [DMW+98] and the references therein, to which we refer the reader.

5.1.5 Revisiting the DiVincenzo criteria

Does the trapped-ion quantum computer realize the five DiVincenzo criteria enumerated in Chapter 4? While a definitive answer must come from further experiments, there is some cause for optimism.

Again, a quantum computer must ...

1. ... *be physically scalable to an arbitrary number of well defined qubits.* In principle, arrays of segmented ion traps allow scaling to an arbitrary number of qubits. Experiments have realized traps with several segments and manipulated ions in these traps. However, further experiments must investigate scaling to large numbers of ions to determine if scaling is strictly a matter of economics and engineering.

2. ... *be initializable to a well defined starting quantum state.* Optical pumping and laser cooling techniques initialize ion-qubits into a known pure quantum state, particularly the ground state of the atom-oscillator system. Experiments have achieved ground state cooling for a small number of ions.

3. ... *have long coherence times relative to gate times of a universal set of gates.* The lifetime of entanglement used in quantum information processing experiments can be longer than 1 second in practice [RLR+04]. More recent unpublished results use first order field independent transitions to protect states from fluctuating potentials on the trapping electrodes, leading to even longer coherence times [Bla04]. Gate times are on the order of tens of microseconds.

4. ... *permit high quantum efficiency measurements on arbitrary single qubits.* Electron shelving techniques allow measurement in the computational basis with efficiency approaching 100%.

5.2 Ion-Trap System Architecture Model

This section defines a high-level stabilizer model of the QCCD ion trap quantum computer that can be efficiently simulated to determine fault-tolerance threshold estimates using the technique described in Chapter 4. The section abstracts physics reviewed in section 5.1 to give a simplified stabilizer simulation model. When appropriate, we take parameters for this model from experimental data, so these experimental values are listed in this section as well.

Subsection 5.2.1 describes the stabilizer model in detail, then subsection 5.2.2 summarizes the model as implemented in software.

5.2.1 Stabilizer Simulation Model

This subsection describes a system architecture model for stabilizer simulation. An appropriate model captures relevant operation times and failure probabilities so we can study system-level trade-offs, like those studied in modern computer architecture. Even without detailed simulations of ion trap physics, the model captures aspects of ion trap quantum computing that most greatly influence the fault-tolerance threshold.

Ion-qubits and RF Traps

This subsection describes the system architecture model for ion-qubits and RF traps. The QCCD described in subsection 5.1.1 contains trap features such as movement channels, trapping regions, and T, X, or L junctions. A simple *trap matrix* of tri-state *trap elements* describes the layout of a QCCD. A trap element is either an electrode on substrate, an empty space, or an ion. The ions also have an associated function such as data storage, ancillary use, or sympathetic cooling. We neglect ion species in this model. The trap matrix is an arbitrary matrix over trap elements, though practical constraints may dictate the placement of these elements. Several electrodes may constitute a single trap, containing at most a few ions. Figure 5-5 shows a sample trap matrix with electrodes inserted along the boundaries of empty trap elements.

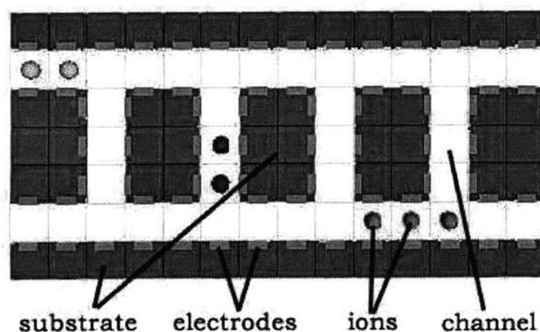


Figure 5-5: A sample layout of a QCCD ion trap architecture is specified by a trap matrix. Each non-empty location in the trap matrix can contain an electrode or a single ion.

Gates and Measurement

Each ion in the trap matrix is identified by a unique label that associates that ion with a qubit in a stabilizer circuit (see Chapter 2 for a discussion of stabilizer circuits). Any ion in the trap matrix may be the object of a single qubit gate at any location in the trap matrix. Ions adjacent to one another in the trap matrix may participate in two qubit gates regardless of their position in the trap matrix. Depending on the specifics of the model's implementation in our simulation tools, ions may need to be explicitly placed into chains. We model chains as disjoint sets of ions and assume that ions may participate in two qubit gates if and only if they belong to the same chain.

The ion trap system permits all single qubit rotations, so the stabilizer model admits all of the single qubit stabilizer gates X, Y, Z, H, and S. These gates take time t_1 on the order of 1 μs. Two qubit gates in trapped-ion systems are typically controlled-Z gates implemented using blue sideband pulses. The controlled-Z gate is composed with Hadamard gates to implement controlled-NOT. All two qubit gates take time t_2 on the order of 10 μs. Trapped-ion systems also allow controlled-Z gates controlled by several qubits in the same ion chain. Because these gates are not stabilizer gates though, they are excluded from the model.

Electron shelving implements a highly reliable projective measurement. The stabilizer model simply makes this a single qubit projective measurement taking time

t_m on the order of 100 μs. Ions can be measured anywhere in the trap matrix.

Moving Ions between Traps

Individual ions can be moved from their current position in the trap matrix to any adjacent, unoccupied position. Moving to an adjacent position takes time t_t, and moving d steps takes time dt_t. Assuming that each cell of the trap matrix is 10 μm square and that the average ion velocity during ballistic transport is about 10 ns/μm, t_t is on the order of 0.1 μs.

Depending on the implementation in software, the system model may include operations to split ion chains or join two ion chains into a single chain. Each of these operations takes a fixed amount of time t_{split} and t_{join}. Published results place these times on the order of 10ms [RBKD+02].

Moving, splitting, and joining are expected to cause a small amount of heating. We do not model heating, but suggest a very simple system model. A single parameter h represents the mean population of the bus qubit. Each operation, particularly movement, increments h. The computation fails if the ions leave the Lamb-Dicke regime, or if $h \gg 100$. Cooling pulses of duration t_c on the order of 10 μs can reduce this heating [KWM+98]. Longer pulses may be required to cool other motional modes, and the parameters may be different for sympathetic cooling rather than direct sideband cooling.

Noise

The stabilizer system model uses the depolarizing noise model for errors in ion trap systems. The model depolarizes qubits individually with a given probability (after or) prior to each gate or operation (see Chapter 4). Noise may be biased toward phase flips in trapped-ion systems, but we model uniform depolarization. This conservative assumption overestimates noise contributions due to operation failures.

Single qubit gates fail with probability p_1. In current practice, p_1 is approximately 10^{-4} [DMW+98]. Two qubit gate fidelity is realistically a function of the motional mode population. However, we neglect this dependence in our model and choose

instead a fixed probability of failure p_2 for a two qubit gate. Published results have demonstrated two qubit gate failure probabilities of about 3×10^{-2} [DDV+03].

Projective measurements may fail with probability p_m. In practice, p_m is approximately 10^{-2} in the worst case. Scattering from projective measurements can also decohere adjacent qubits. This might be modeled using a depolarizing error with probability proportional to the inverse square separation between the measured ion and the adjacent ion, $p = p_s/d^2$ [LMBK+04]. We do not include such effects in the model for this thesis, but leave this to future work.

We neglect heating caused by movement and assume frequent cooling pulses. However, movement leads to dephasing because the ion will sample ambient electromagnetic fields and the movement process may introduce some systematic error. Movement operations fail with probability $p = 1 - \exp(-dp_t)$, where p_t is a parameter of the model and d is the number of rows or columns traversed in the trap matrix.

Perhaps the strongest assumption is the neglect of memory errors. We assume that memory errors can be incorporated into gate failure probabilities. This assumption is based partly on the fact that dephasing times nearly correspond to spontaneous emission times [RLR+04].

5.2.2 Model implementation summaries

This section serves as a reference to the system architecture model as implemented in two software simulators. The *ion-trap simulator* (ITSIM) has a detailed ion-trap system architecture model that counts heat, time, and automatically parallelizes operations. The *quantum architecture simulator* (ARQ) is a faster but less detailed variant of the ion-trap simulator that includes several different "machine" models. Because section 6.3 describes both simulator implementations in detail, this section avoids software details to give the reader a clear view of the models themselves and clarify how these models differ between the two software implementations.

Table 5.1 summarizes the system architecture model implemented in the ITSIM simulation tool. The ITSIM model uses a rectangular trap matrix to describe trap layouts. Each ion in the layout has a position in the trap matrix, a label associating

it with a qubit, and a type. The ion type describes its function in the simulated computer, such as "data", "ancilla", or "sympathetic". The collection of ions is partitioned into disjoint sets of chains that update during split or join operations. Ion heat is recorded and changes during movement and cooling operations. All operations have an associated time and failure probability. When operations fail, their qubit arguments are depolarized with the appropriate probability. Memory errors on resting qubits are not modeled.

Table 5.2 summarizes the ARQ model. The significant difference from the ITSIM model is that the mean cost of splitting and joining ion chains is absorbed into the two qubit gates. This simplicity allows the ARQ implementation to simulate larger circuits and more general architectures. Other minor differences are that ions are no longer typed and heat is not recorded. However, future work can easily extend the ARQ implementation. See Chapter 6 for details about implementing new models and for a discussion of simulation time.

Finally, Table 5.3 is a guide to the parameters of the system architecture model as determined by present and anticipated experimental results. Each operation fails with a fixed probability that is related to the experimentally determined fidelity and takes some fixed, average amount of time (see Chapter 4). Two qubit gates introduce the most error in current experiments, and splitting and cooling take the greatest amount of time.

5.3 Quantum Computer Layouts

As mentioned in Chapter 4, geometric constraints influence the reliability of fault-tolerant quantum computers. A layout is an important practical constraint because it specifies the arrangement of qubits in space. The trap matrix of the ion trap system model described in subsection 5.2.1 permits many different layout choices for the ion trap quantum computer. However, not all layout choices lead to a reliable quantum computer. What traits are desired in an ion-trap layout? How can we ensure that a fault-tolerance threshold exists?

MODEL FEATURE	DESCRIPTION
Trap layout	rectangular trap matrix
	ions have types based on their function
Ion chains	ions are grouped into sets of chains
Ion heat	modeled by an integer h
	incremented by movement, decremented by cooling
Memory errors	Not modeled
Single qubit gates	H, S, X, Y, Z
	probability p_1, time t_1
Two qubit gates	CX, CZ on ions in the same chain
	probability p_2, time t_2
Measurement	projective measurement of single qubits
	probability p_m, time t_m
Movement	linear vertical and horizontal movement in integer increments
	ions in the same chain participate in the same movement operation
	probability $p(d) = 1 - e^{-p_t * d}$, time $t = d * t_t$
Split	separates a chain into two chains
	probability p_{split}, time t_{split}
Join	joins two chains into a single chain
	probability p_{join}, time t_{join}
Cool	probability p_{cool}, time t_{cool}

Table 5.1: Summary of the trapped-ion system model as implemented in the ITSIM simulation tool. The upper half of the table lists important points regarding the trap layout, ion chaining, heating, and memory error model. The lower half of the table lists the available operations and their adjustable parameters.

Model Feature	Description
Trap layout	rectangular trap matrix
Ion chains	Not modeled
Ion heat	Not modeled
Memory errors	Not modeled
Single qubit gates	H, S, X, Y, Z
	probability p_1, time t_1
Two qubit gates	CX, CZ on nearest-neighbor ions
	probability p_2, time t_2
Measurement	projective measurement of single qubits
	probability p_m, time t_m
Movement	linear vertical and horizontal movement in integer increments
	probability $p(d) = 1 - e^{-p_t * d}$, time $t = d * t_t$

Table 5.2: Summary of the trapped-ion system model as implemented in the ARQ simulation tool. The ARQ tool does not model the ion-trap in as much detail as the ITSIM tool, but the simplified model allows us to explore larger systems at higher levels of recursive simulation.

Operation	Time	Error Probability	Time	Error Probability
Single Gate	$1\mu s$	0.0001		10^{-8}
Double Gate	$10\mu s$	0.03		10^{-7}
Measure	$100\mu s$	0.01		10^{-8}
Movement	$10ns/\mu m$	$0.005/\mu m$		10^{-6}/cell
Split	$1000\mu s$	0.001		
Cooling	$300\mu s$			

Table 5.3: Summary of physical parameters for ion trap system models. The current experimental values of movement and splitting parameters are taken from [RBKD+02, CSB+04]. The current experimental values of gate parameters are taken from [DMW+98] and [DDV+03]. Sympathetic cooling times can be found in [BDS+03]. Projected parameter values are motivated by [WH00].

This section presents three different ion-trap layouts, two of which have desirable fault-tolerance properties. The first layout in subsection 5.3.1 permits universal computation in the local setting. The second layout in subsection 5.3.2 uses a tree to keep qubits that frequently interact close to one another. This reduces failures due to movement errors. The final layout in subsection 5.3.3 also uses a tree, but incorporates the concept of supporting ancilla to make the layout very reliable at high levels of recursive simulation. This layout is less efficient at low levels of recursive simulation, but exhibits a fault-tolerance threshold.

5.3.1 Turing machine layouts

What is the simplest possible way to organize an ion-trap quantum computer to permit universal quantum computation? Suppose we give an ion-trap quantum computer a circular "tape" of ions and a single trapping region for quantum logic operations, as shown in Figure 5-6. Any ion or pair of ions can be loaded into the logic region, participate in a quantum gate or measurement, and be unloaded back onto the tape simply by rotating the tape to the appropriate address.

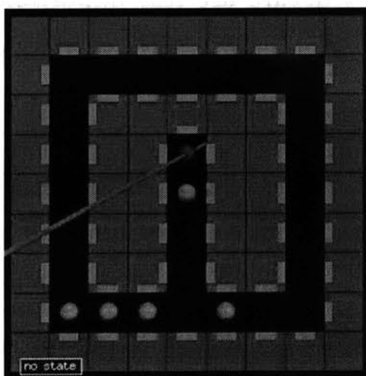


Figure 5-6: This is one possible layout for ion-trap quantum computing based on a Turing machine with a finite circular tape. The blue squares represent non-conductive substrate that supports electrodes represented by gray rectangles. Black squares are empty space in which ions can be trapped. Each ion is drawn as a colored sphere. Green spheres are ions that hold data while the blue sphere is an ion used for sympathetic cooling. The green line represents a beam of laser light striking an ion in the accumulation region, the region where quantum logic operations take place.

The control for the Turing machine layout is exceedingly simple from a computer

architect's perspective, since each qubit can be given a specific linear address on the tape and operations are essentially serial. The radius of the circular tape can be as large as necessary to accommodate the application. The farthest movement distance is no worse than linear in the number of qubits, so the time overhead is not excessive.

However, each movement operation is unreliable, so qubits load from the tape with a failure probability that grows linearly in the number of qubits. Therefore, even if we execute a fault-tolerant quantum circuit, the layout does not scale reliably to a large number of qubits. To compound the problem, this simple layout does not permit parallel quantum gates, as fault-tolerant quantum computation requires.

5.3.2 H-Tree layouts

The H-tree layout solves some of the problems identified though the Turing machine layout. An H-tree is a type of balanced tree [Hua85] that we apply in this section to create layouts for fault-tolerant trapped-ion quantum computing. For layouts, the details of the tree as a data structure are less important since the layout is static, i.e. the positions that qubits can occupy are static, though the positions that qubits occupy at any give time are dynamic.

The H-tree can be viewed instead as an L-system fractal in the plane [Lin68]. There are several ways to think about an L-system fractal, which arises from a language described by an L-system grammar. Rules of the grammar are iteratively applied to generate the fractal. Our H-tree is generated from an initial region of the plane and a sequence of transformations. We select a very specific H-tree generated from an H-shaped region of the plane and a set of four transformations that translate and scale the H. More specifically, let $\{A, B, C\}$ be the alphabet of our language, where A means "move forward 1 unit", B means "rotate counter-clockwise 90 degrees", and C means "rotate counter-clockwise 180 degrees". The initial string constant,

$$S_0 = ABACAACABAABACAACABA, \tag{5.13}$$

draws the first H. The rules of the grammar,

$$p_1 : A \longrightarrow AA \tag{5.14}$$

$$p_2 : C \longrightarrow BS_0B \tag{5.15}$$

correspond to recursive simulation. The p_1 rule doubles the length of each segment in the first H, and the p_2 rule inserts four smaller H's at the tips of each arm. Figure 5-7 shows the H-tree fractal generated by a similar grammar. This figure corresponds to 6 levels of recursive simulation.

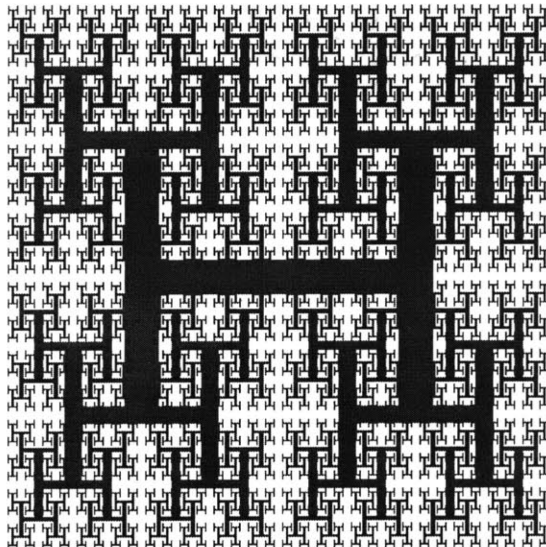


Figure 5-7: This H-tree fractal is generated from the central "H" by scaling and translation. As an ion-trap layout, the gray region should be interpreted as a large channel with many smaller rows and columns of traps. The white region should be interpreted as substrate. A 6-simulated quantum error-correction circuit on the H-tree looks like this figure.

For our purposes, a logical qubit is a single qubit encoded using an $[[n, 1, d]]$ quantum computation code. Each physical qubit is assigned a home position at a leaf of the H-tree. Each parent node of the tree is a single logical qubit at the next level of recursive simulation, i.e., an $(L + 1)$-block. Figure 5-8 shows a 1-block, the simplest node of the H-tree. Each logical qubit is localized to a single node, and all logical qubits at level L reside at the same scale of the fractal. This localization makes

movement distances within a logical qubit relatively small compared to movement distances between logical qubits. Because the H-tree is space filling, we also expect the layout to be space efficient in some sense.

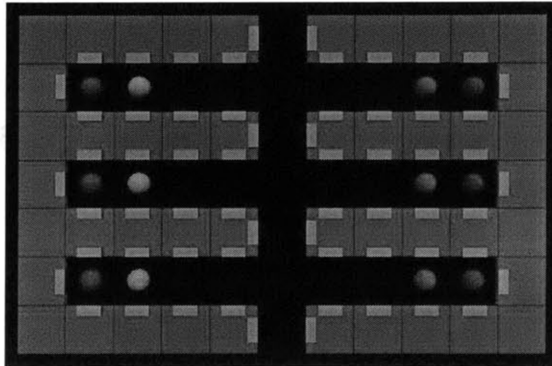


Figure 5-8: The basic unit of the H-tree layout for the 3-qubit bit-flip code is constructed from a trap with 6 channels that branch from a central channel. Three of these channels store data qubits, shown in green, and three of these channels store ancillary qubits, shown in red. Ancilla and data qubits each share their channels with a sympathetic cooling ion shown in blue. The ancilla are prepared first using the central channel, then move transversally to extract an error syndrome from the data. The data is stationary for the entire error-correction process and only moves to interact with other data qubits.

Figures 5-9 and 5-10 show the H-tree layout populated with qubits in their home location. This layout corresponds to a 3-simulated 3-qubit quantum memory. Each parent has 6 leaves. Three of these leaves store the 3 qubits that encode data, and three of these leaves store ancilla qubits for syndrome extraction (see the circuit in Figure 2-2). Notice that the width of the channel increases exponentially, as in Figure 5-7.

Ancilla preparation occurs in the leaves of the H-tree. At each node of the tree, a single logical ancilla qubit is paired with each logical data qubit. There is obviously enough space for local interactions that prepare this qubit. Many more ancilla can be prepared and verified by adding leaves to the tree.

Figure 5-9: An ion-trap layout for a 3-simulated 3-qubit quantum memory. The solid circles represent ions and the cyan squares represent substrate. Black regions of the layout represent free space in which ions can be trapped and moved. Green circles are data ions, red circles are ancilla ions, and blue circles are sympathetic cooling ions.

Figure 5-10: An alternate view of Figure 5-9.

5.3.3 Ancilla-supported layouts

The H-tree layout has two desirable properties: logical qubit localization and space efficiency. However, movement distances grow exponentially with the level of recursive simulation, so the probability of an uncorrectable error will quickly approach unity. This is not a great problem for a few levels of recursion, and perhaps for practical purposes. Nevertheless, we would like to have a concrete ion-trap layout with a fault-tolerance threshold.

As mentioned in Chapter 4, the interplay of layout and ancilla preparation could have significant impact on current threshold estimates. In fact, ancilla preparation and layout are interdependent because movement operations must consume ancilla to ensure that errors do not accumulate during movement and that a threshold continues to exist.

This subsection is divided into several parts. The first part describes an argument that a threshold exists with local gates [Got00, STD04]. The second part describes a potential path to realizing a concrete ancilla factory in systems that allow dense arrays of movable qubits, such as ion-trap arrays. The third part explicitly constructs a layout for an ion-trap quantum computer that exhibits a fault-tolerance threshold.

Local fault-tolerant quantum computation

Moving or swapping qubits within a quantum computer causes errors. Some movement distances within a recursively simulated quantum circuit grow exponentially with the level of recursive simulation. This growth causes errors to accumulate such that naive fault-tolerant computation eventually becomes impossible.

Intuitively, error accumulation can be halted and reversed through very frequent *intermediate error-correction* (IEC). This idea is made precise in [ABO99, Got00, STD04]. Intermediate error-correction must be performed exponentially often, so a local quantum computer suffers an exponential slowdown in the number of levels of recursive simulation. Nevertheless, scalable fault-tolerant local quantum computation is possible, so IEC-capable layouts are in some sense canonical layouts for scalable,

fault-tolerant quantum computers.

Following [Got00], suppose that the failure probability of a fault-tolerant quantum gate satisfies

$$p_{k+1} = Cr^{k+1}p_k^2, \tag{5.16}$$

where k is the level of recursive simulation, C is a positive real constant bounding the number of fault-paths that lead to failure, r is a scale factor determined by layout, and the quantum computation code corrects $t = 1$ errors. The probability of failure for an L-simulated gate is

$$p_L = \frac{1}{Cr^2}(Cr^2 p)^{2^L}/r^L. \tag{5.17}$$

Hence, the *threshold for local quantum computation* follows a rule like

$$p_{th,local} \equiv \frac{1}{Cr^2} = p_{th}\frac{1}{r^2}. \tag{5.18}$$

This suggests that the threshold is scaled down by some power of the layout scale factor r. The latter equality is true if C does not change between nonlocal and local error models. More recent work suggests that the local threshold scales by $1/r$ [STD04].

Ancilla support structure

To achieve a local threshold, logical qubits and their ancilla must be near one another, as in the H-tree layout. Furthermore, the ancilla must move with their associated logical qubits, and a large number of ancilla must be prepared so that fresh ancilla are always available for error-correction during movement. The *ancilla support structure* of a logical qubit is the collection of ancilla necessary to refresh the logical qubit as the logical qubit moves. Some ancilla within the support structure must be used to correct the support structure itself because the support structure moves with the logical qubit.

Following [Got00], we design an ancilla support structure for a fault-tolerant trapped-ion quantum computer. The construction starts in three dimensions and

later reduces to two. Begin with a 3-dimensional lattice of physical ion qubits, organized into a stack of planes. Each ion qubit in the lattice belongs to a particular quantum codeword encoded by a q-concatenated $[[n, 1, d]]$ code for some q. Each plane is a single logical data qubit encoded at level L and many ancilla qubits encoded at levels varying from 1 to L.

One such plane is shown in Figure 5-11. A *q-codeblock* in this figure is a collection of physical qubits belonging to the same quantum codeword encoded at level q. A collection of n^L physical qubits that all belong to q-codeblocks for a fixed q are called a *q-superblock*. There are n^{L-q} q-codeblocks in a q-superblock. Each number q across the top of the figure labels a vertical q-superblock of ancilla qubits. The letter "d" labels the data L-superblock. Adjustable nonzero integer parameters M_k determine the total number of supporting ancilla of each type.

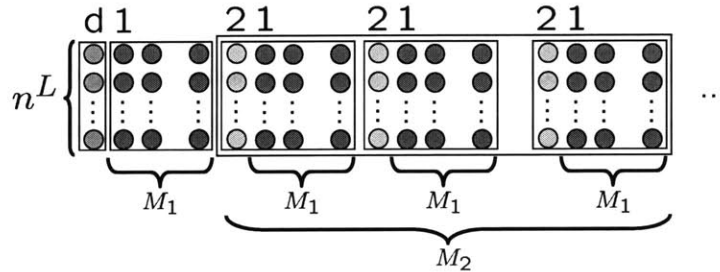


Figure 5-11: A single logical qubit and its ancilla support structure reside in a plane for a quantum computer built from a 3-dimensional lattice of ions. The left column of green circles represents the data qubits in an L-codeblock. The remaining columns of red and blue circles represent ancilla qubits. Each column of red ancilla qubits is a 1-superblock and each column of blue ancilla qubits is a 2-superblock. The pattern continues – for each q-superblock with $q > r > 0$ there are M_r r-superblocks supporting that q-superblock.

When $M_k = 1$ for all k, the logical qubit plane looks like Figure 5-12. Except for the data superblock, the solid boxes in this figure are called *q-chunks*. Each contains 2^q superblocks: a leading (leftmost) q-superblock to correct the data and $2^q - 1$ r-superblocks, $r < q$, to correct the leading q-superblock. This shows that the ancilla support for $M_k = 1$ is a binary tree, and suggests that the data and each q-codeblock can be error-corrected using nearby ancilla. When $M_k \neq 1$ for some k, a q-chunk

contains $N_q = M_q(1 + N_{q-1})$ qubits and $N_1 = M_1$.

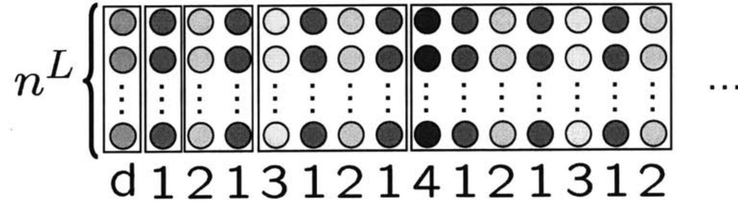


Figure 5-12: This is the same structure shown in Figure 5-11 when $M_k = 1$ for all k. The data codeblock is on the left. Each box of qubits to the right of the data is called a *q-chunk* – one q-superblock to correct data and the rest to correct ancilla in the box. There are L chunks to correct data encoded at level-L.

There are good reasons for choosing the arrangement in Figure 5-11 and Figure 5-12. Assume that there is space between qubits in these figures so that a given superblock can complete operations local to itself and can move through other superblocks if necessary. Notice that the relative distance between a q-chunk and the data "d" grows exponentially in q, hence equation 5.16 expresses how the failure probability (of a transversal single qubit gate at least) changes from level to level. In addition, syndrome extractions can proceed in parallel and each q-superblock for $q > 1$ has associated ancilla available for error-correction that can move with that q-superblock.

So far we have not mentioned the details of ancilla preparation. However, ancilla preparation is crucial for fault-tolerant quantum computing. How does this layout account for ancilla preparation? The layout can include enough ancilla by adjusting the M_k. However, the layout must include a sufficient number of verification qubits as well. If $M_k \gg 1$ for all k then we can simply recruit verifiers. Yet, the classical control may become very complicated. We may be able to simplify this control, perhaps, by designating verifiers in the layout.

The optimal ancilla preparation scheme is both massively parallel and highly data-dependent, in the sense that recovery operations must block until enough ancilla are available. All M_k ancilla must be encoded in parallel for a threshold to exist. After encoding, or *generation*, the ancilla must be verified [Ste03b]. Some ancilla will not pass the verification and will be left behind while the successfully prepared ancilla

participate in recovery operations. During recovery, ancilla that failed verification are re-encoded until they pass verification, then continuously error-corrected until a sufficient number have passed verification. A single "thread" might prepare and monitor each ancilla in the support structure. When the ancilla is ready, the thread may send a signal to waiting error-correction operations.

Planar ion-trap layout for the [[7, 1, 3]] code

A two dimensional layout may be more practical than a three dimensional layout, so we now describe a concrete planar ion-trap layout and include designated verification qubits. First, stretch out the supported logical qubit in Figure 5-11 so that it becomes a single line of qubits. Next, stack many of these lines, each of which is a logical qubit, to form a plane.

There are at least two highly symmetric ways to stretch out planes to lines such that a threshold exists; we can *interleave* data and ancilla qubits, or we can place data *adjacent* to ancilla. Both ways lead to a linear grouping of qubits called the *a-supported l-block* for integers $a \leq l$.

Figure 5-13 shows a 1-*supported L-block* with data and ancilla interleaved. This block contains data encoded at level-L interleaved with M_1 ancilla encoded at level 1 to error correct the data. In addition, the block has $n_v(M_1+1)$ verification qubits. For the [[7, 1, 3]] code, for example, $n_v = 4$ to measure half of the 6 stabilizer generators and the logical Z operator.

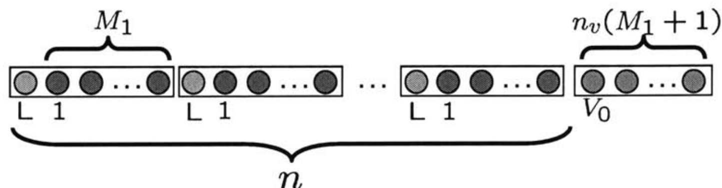


Figure 5-13: This is a 1-supported L-block for which data and ancilla are interleaved. M_1 red level 1 ancilla follow each of the n green data qubits. These nM_1 level 1 ancilla are used in level 1 error-correction of the n data qubits, though the data qubits are part of a larger L-codeblock. There are $n_v(M_1 + 1)$ gray verification qubits used to prepare up to $(M_1 + 1)$ 1-codeblocks, the total number contained in the 1-supported L-block if the data qubits need to be prepared as well.

Figure 5-14 also shows a 1-supported L-block, except the data is placed adjacent to the ancilla and verifiers. There are the same number of qubits in this block, and they have the same roles as those in Figure 5-13.

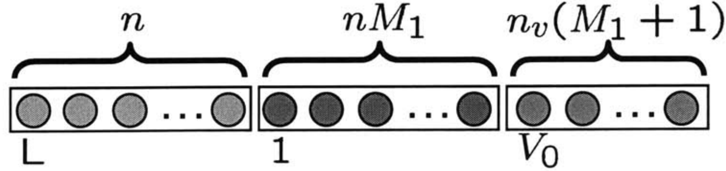


Figure 5-14: This is also a 1-supported L-block, as in Figure 5-13, except data is placed adjacent to the ancilla and verifiers. Compared to the interleaved block, this block has a lower ancilla preparation failure rate due to movement but a higher data-ancilla interaction failure rates due to movement.

Continuing the recursive construction, an interleaved 2-supported L-block is shown in Figure 5-15, and an adjacent 2-supported L-block is shown in Figure 5-16. These figures clarify the construction process, which continues in an obvious way to an a-supported L-block, and ultimately to an L-supported L-block. An L-supported L-block is a logical data qubit encoded at level L together with enough ancilla to carry out fault-tolerant error-correction of the data with intermediate error-correction.

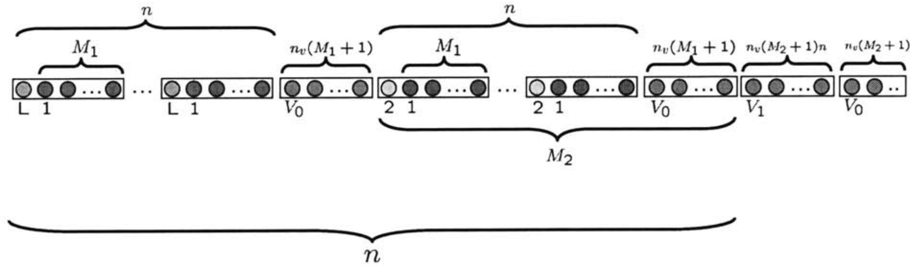


Figure 5-15: This is an interleaved 2-supported L-block. Notice that Figure 5-13 is contained within this figure n times for the green data qubits and n times for the blue level 2 ancilla qubits. A total of $n_v(M_2+1)n$ gray verifiers V_1 appear at the end of the block to verify the blue level 2 ancilla qubits. These V_1 verifiers have their own verification qubits V_0 because the V_1 verifiers themselves must be prepared into a logical zero state and verified.

The specific recursion rules (A) and (B) for constructing an a-supported l-block are

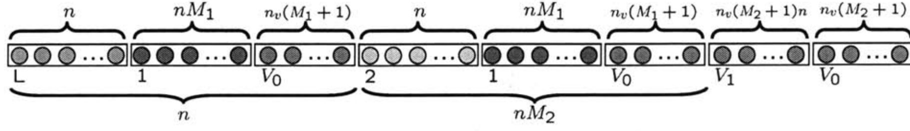


Figure 5-16: This is an adjacent 2-supported L-block. There are the same number of qubits as in Figure 5-15. The qubits have different locations but the same roles, and they are labeled as before.

(A) A 1-supported a-block terminates the recursion.

(B) An a-supported l-block consists of n $(a-1)$-supported l-blocks (housing supported data) that are either

(i) *interleaved* with nM_a $(a-1)$-supported a-blocks (housing supported ancilla) in an alternating pattern, or

(ii) *adjacent* to nM_a $(a-1)$-supported a-blocks (housing supported ancilla),

and followed by

$$N_v \equiv n_v(M_a+1)(n^{a-1}+\cdots+n^2+n+1) = n_v(M_a+1)\frac{n^a-1}{n-1} \tag{5.19}$$

verification qubits.

There may be reason to prefer either the interleaved or adjacent layout. The interleaved layout has a higher preparation failure rate but suffers fewer errors during interaction. The adjacent layout has the opposite property, namely fewer preparation failures but more interaction errors. Hence, the adjacent layout requires fewer ancilla than the interleaved layout because more will pass verification. However, the interleaved layout may have a modestly higher threshold because the error-correction circuits will be more reliable, at the cost of an increased number of ancilla. Post-selection may remove many of the errors in interleaved ancilla preparation.

Using the recursion rule, the number of physical qubits for an $[[n,1,d]]$ code in an L-supported L-block can be written down. The first part of the rule gives

$$N_{a,l} = nN_{a-1,l} + nM_aN_{a-1,a} + n_v(M_a+1)\frac{n^a-1}{n-1}. \tag{5.20}$$

The second part of the rule terminates the recursion, telling us that a 1-supported a-block contains $N_{1,a} = (n + n_v)(M_1 + 1)$ physical qubits. Note that if $M_k = 1$ for all k and $n_v = 0$ then $N_{L,L} = (2n)^L$. Placing these physical qubits on the layout from left to right is equivalent to a depth-first traversal of the tree generated by the recursion rule, and recovering the logical qubits is equivalent to postfix traversal of the same tree.

Each line of qubits can undergo simultaneous ancilla preparations and recovery operations at all levels of encoding if the lines of qubits are parallel to one another and spaced apart by no less than $(2n)^L$. The exponential separation is sufficient to prevent collisions, but may not be strictly necessary. In addition, the parallel orientation permits direct movement between logical qubits for transversal swaps and gates.

5.4 Thresholds for Trapped-Ions

Trapped-ion experiments show great promise as building blocks for progressively larger quantum computing experiments. Experiments with tens of ions now seem imminently possible, and these may lead to the first experimentally demonstrated fault-tolerant quantum gate [Ste04]. What gate and movement failure probabilities are needed to realize a reliable local fault-tolerant gate? More simply, for what parameters does a recursively simulated local fault-tolerant recovery network become more reliable? Using the techniques, models, and layouts developed in this chapter and previous chapters, we can decisively answer this question for specific recovery networks, ion-trap models, and layouts.

The answer is a precise set of reliable parameters – parameters for which the local fault-tolerant recovery network becomes more reliable. Reliable parameters are useful in their own right, but they also give us an estimate of the fault-tolerance threshold, as discussed in Chapter 4. In this case, we expect that the fault-tolerance threshold will be no greater than the greatest reliable parameters we can observe for a fault-tolerant recovery network, though this is not strictly true.

This section experimentally estimates the threshold for localized fault-tolerant ion-trap quantum computing for the $[[3,1,1]]$ bit-flip code and the $[[7,1,3]]$ quantum code. We use the ARQ system model from this chapter and recovery networks described in Chapter 4. Subsection 5.4.1 computes reliable parameters for the 3-qubit bit-flip code, and subsection 5.4.2 does the same for the 7-qubit quantum code.

In both cases, the recovery networks are localized to the H-tree layout described in subsection 5.3.2. The H-tree layout is a practical layout for a few levels of recursion, because the overhead of intermediate error-correction is undesirable. This means that our reliable parameters are realistic for ion-trap quantum computers with tens or hundreds of ions, but the threshold estimates are potentially optimistic.

5.4.1 $[[3,1,1]]$ bit-flip code

The $[[3,1,1]]$ bit-flip code is analyzed in the nonlocalized setting in subsection 4.4.4. There we found an estimate of the fault-tolerance threshold, $p_{th} \approx 0.035$ for a quantum memory based on this code. In this subsection, we determine how this estimate changes when the network is localized to the H-tree layout.

As in subsection 4.4.4, we compose $T = 32$ recovery networks, compute the failure probability p_f under bit-flip noise, then divide that probability by T to find an average recovery failure probability p_f/T. The average recovery failure probability now depends on both the gate failure probability $p_g = \gamma_1 = \gamma_2$ and the movement failure probability $p_m = \gamma_m$. Varying p_g and p_m produces a surface for each L-simulated recovery network, as shown in Figure 5-17. The surfaces in Figure 5-17 interpolate between 400 uniformly spaced data points. Each data point is taken to smaller standard error than Figure 4-4.

The simulation reveals a region in which recursive simulation improves the reliability of the fault-tolerant recovery network. The boundary of this region, shown in gray, fluctuates slightly due to statistical noise in the data, but Figure 5-18 suggests that these fluctuations are negligible. Both surfaces appear to vary almost monotonically, crossing along what is approximately a line when projected onto the horizontal plane. The intersection manifold of Figure 5-18 extends from $(p_g, p_t) \approx (0.005, 0.0027)$

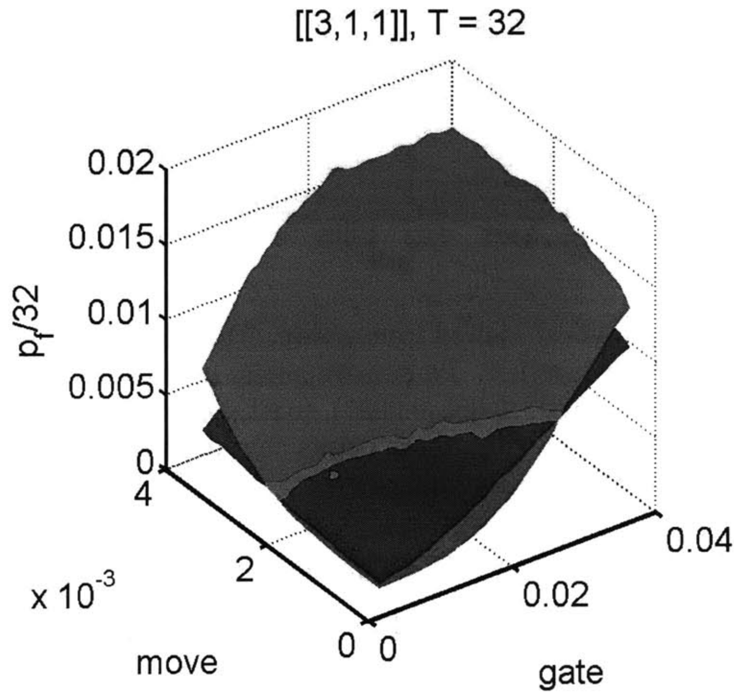


Figure 5-17: This figure shows how the failure probability of an L-simulated fault-tolerant recovery network for the [[3, 1, 1]] code depends on both gate and movement failure probabilities under bit-flip noise. The blue surface is the average failure probability of a recovery network for a 0-concatenated code, and the red surface is the average failure probability of a recovery network for a 1-concatenated code (i.e., a 1-simulated recovery network). The "gate" axis is the failure probability $p_g = p_1 = p_2$ and the "move" axis is the failure probability p_t (see Table 5.2). There is a region where recursive simulation improves reliability of the recovery network – the region where the red curve passes beneath the blue curve. The neighborhood of the crossing manifold is colored gray; these points can be identified with neither the red nor the blue surface within the standard error.

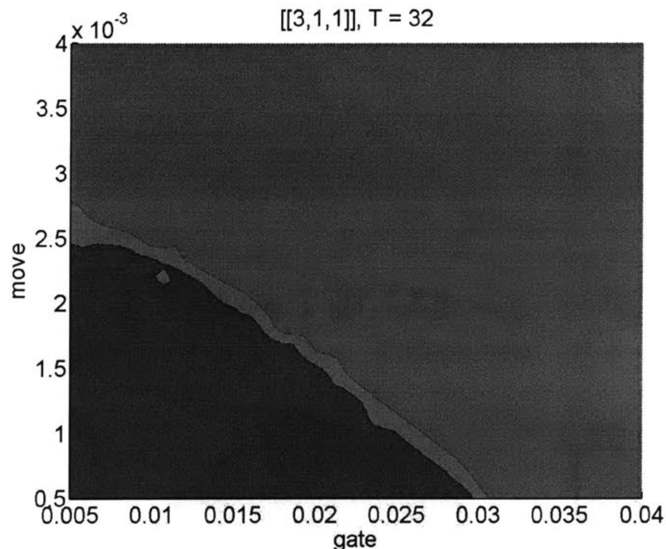


Figure 5-18: This is Figure 5-17 viewed from above. This plot emphasizes the set of reliable parameters in the lower left. These parameters lie approximately within the triangle given by $(0.005, 0.0005)$, $(0.005, 0.0027)$, $(0.03, 0.0005)$. The neighborhood of the crossing manifold is colored gray; these points can be identified with neither the red nor the blue surface within the standard error.

to $(p_g, p_t) \approx (0.03, 0.0005)$ based on the interpolated data. The maximum distance across a single node of the H-tree is 9 cells, which explains the order of magnitude difference between movement and gate failure probabilities.

The numerical results suggest that a reasonable estimate of the fault-tolerance threshold for the $[[3,1,1]]$ quantum code is $p_{th} \approx 3 \times 10^{-3}$. More specifically, recursive simulation improves the reliability of the recovery network for this code under bit-flip errors, even when localized to an H-tree layout, provided that gate and movement failure probabilities are below 0.3%.

5.4.2 $[[7,1,3]]$ quantum code

This subsection calculates the reliable parameters for the $[[7,1,3]]$ fault-tolerant recovery network, localized to the H-tree layout, under depolarizing noise. This network is experimentally analyzed in the nonlocalized setting in Chapter 4. There we find a threshold estimate of approximately 7.2×10^{-4}.

Our method for calculating reliable parameters is the same as the method used

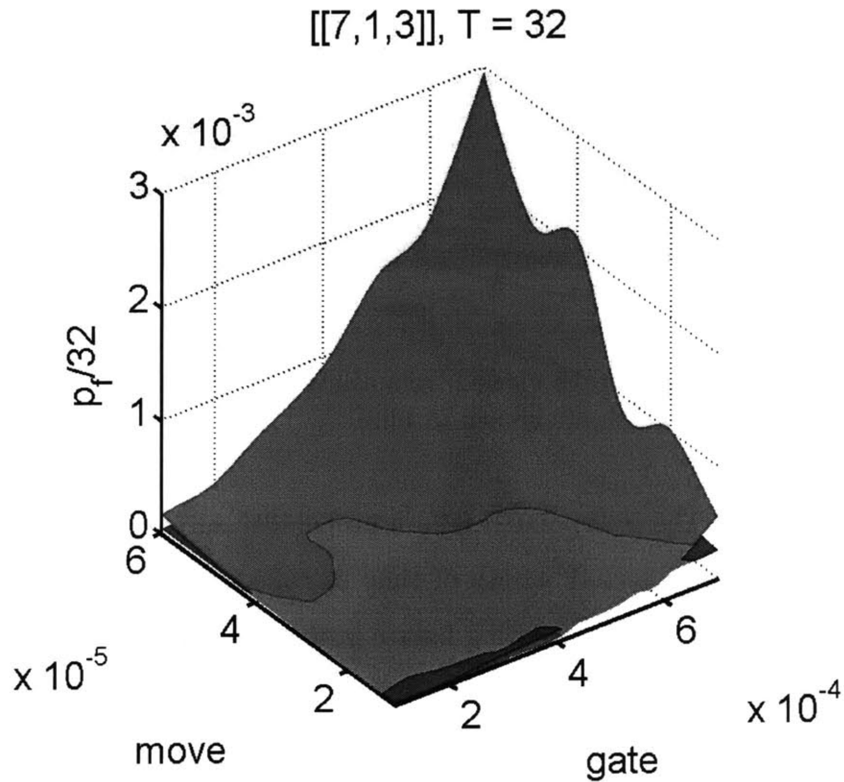


Figure 5-19: This figure shows how the failure probability of an L-simulated fault-tolerant recovery network for the [[7, 1, 3]] quantum code depends on both gate and movement failure probabilities under depolarizing noise. The blue surface is the average failure probability of a recovery network for a 0-concatenated code, and the red surface is the average failure probability of a recovery network for a 1-concatenated code (i.e., a 1-simulated recovery network). The "gate" axis is the failure probability $p_g = p_1 = p_2$ and the "move" axis is the failure probability p_t (see Table 5.2). There is a region where the red surface passes beneath the blue surface. Recursive simulation improves reliability of the recovery network in this region. Grey portions of both surfaces represent interpolated data that could be attributed to neither the blue nor the red surface within the standard error.

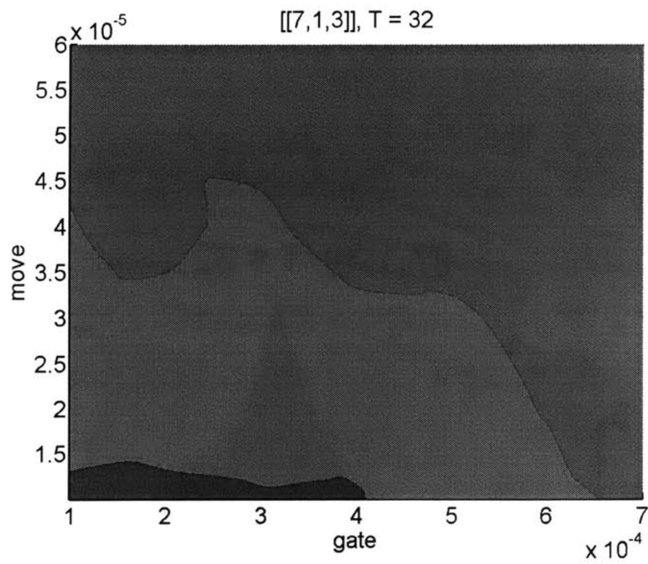


Figure 5-20: This is Figure 5-19 viewed from above. This plot emphasizes the region of reliable parameters which are shown in blue.

in subsection 5.4.1. The interpolated failure probability surfaces are generated from 100 data points, which took 1 month of time to collect on a 32 node cluster computer. Figure 5-19 shows the resulting failure probability surfaces. The blue surface is the failure probability surface of the recovery network acting on 0-concatenated codewords, and the red surface is the failure probability surface of the recovery network acting on 1-concatenated codewords (i.e., the 1-simulated network). There is region of gate and movement failure probabilities for which recursion improves reliability. This region gives us a set of reliable parameters from which we can estimate a fault-tolerance threshold.

Figure 5-20 shows the failure probability surfaces from above. This figure emphasizes the region where recursion improves reliability, and clarifies the uncertainty in the crossing manifold by coloring the manifold gray within the standard error. On this manifold, we find movement failure probabilities an order of magnitude lower than gate failure probabilities because typical movement distances are on the order of 10 cells for the H-tree.

These results suggest a threshold estimate for the [[7, 1, 3]] quantum code localized to the H-tree. The parameters $(p_g, p_t) \approx (3\times 10^{-4}, 1\times 10^{-5})$ are numerically computed

reliable parameters for this code that lie near the crossing manifold, suggesting a fault-tolerance threshold of $p_{th} \approx 1 \times 10^{-5}$.

5.5 Conclusion

This chapter has reviewed the state of experimental trapped-ion quantum computing and developed an architectural model of the trapped-ion quantum computer. Within this model, we explored two layouts with desirable fault-tolerance properties: the H-tree at error probabilities far below threshold and the ancilla-supported IEC layout at error probabilities near threshold. This exploration highlights the role and complexity of realistic layout in fault-tolerant design.

The methods of Chapter 4 allows us to give one of the first threshold estimates for trapped-ion quantum computing constrained to a realistic layout. We have observed a threshold reduction for the $[[7, 1, 3]]$ code from 7×10^{-4} without layout to 1×10^{-5} on an H-tree. This reduction is due to movement on the layout, and can be understood based on the mean movement distance per gate. However, the latter result is likely a pseudothreshold which suggests that the threshold reduction may be even greater.

This program of research can continue along several directions. First, the ion-trap architectural model can be improved along lines already discussed in subsection 5.2.1. Specifically, we need to account for memory errors and decoherence due to heating and laser scattering. Second, many possible layouts remain to be explored and critical trade-offs identified. For example, the trade-offs associated with ancilla-supported layouts have not been studied. Such trade-offs can be explored for the 3-qubit code as a modest extension of the work presented in this chapter. Finally, there remains the overarching question – rather than estimate thresholds, how can fault-tolerance thresholds be experimentally bounded or exactly determined for system architectures with realistic layouts?

Chapter 6

A Software Architecture for Quantum Computer Design Tools

Much like modern digital computers, future quantum computing systems, from experimental demonstrations of quantum fault-tolerance, to complete, large-scale quantum factoring machines, will require complex sequences of control instructions and, ultimately, a system architecture. Though the proper architectural components of a quantum computer are not completely known, fault-tolerant design is almost certainly necessary across all potential technologies. The overhead introduced by fault-tolerant design adds a new element of complexity to quantum computer design that is not present in modern digital computer design.

However, overhead and complexity is not the only problem. In Chapter 5, for example, we not only witnessed the enormous number of ion-qubits and quantum gates in an ion-trap quantum computer acting as a quantum memory, but also the need for layouts with certain properties to maintain fault-tolerance. How does one algorithmically translate from a quantum circuit to a fault-tolerant quantum circuit constrained to a layout? In doing so, is it possible to assert with some confidence that the final design is reliable enough?

To deal with these questions, this chapter applies the divide-and-conquer strategy seen in modern computer design. Software tools for quantum computer design, such as compilers, CAD tools, and simulators, will be indispensable for identifying

and ultimately engineering the architectural components of even modestly complicated quantum computing systems. This chapter proposes a design flow for high-level quantum computer architecture design that is implemented as a chain of interoperable software tools. Along the way, the chapter presents one view of the hardware-software interface based on the observations in Chapter 4; specifically, knowledge of physical resources and mechanisms appears to be necessary to guarantee fault-tolerance. Finally, the chapter describes the interface and organization of two different but related stochastic quantum error-correction simulation tools that implement the final phase of the tool-chain. These tools are used implicitly in the previous chapters to estimate fault-tolerance thresholds.

The first section of the chapter, section 6.1, introduces some results about quantum computer organization that have shaped the view of quantum computing presented here. Section 6.2 offers a concrete design flow consisting of phases of language translation. Each phase corresponds to a collection of language translators, schedulers, CAD tools, and/or simulators. The idea is to provide a scaffolding on which to erect interoperable quantum computer design software. Section 6.3 describes implementations of the simulation stage of the design flow. Finally, section 6.4 concludes the chapter.

6.1 Quantum Computer Organization

This section outlines the high-level organization of the quantum computer viewed as a complex engineered system. This means that we might imagine quantum computers constructed hierarchically from high-level components, which are themselves constructed from architectural components, which are constructed from elementary devices. This approach has been applied to the modern digital computer with great success, so it is reasonable to begin to think about engineering quantum computers in the same way.

The appropriate paradigm for quantum computer system architecture is not yet known, but one can envision a quantum analog of many digital computing concepts.

Of course, the digital computer did not evolve directly into its modern form. It continues to evolve, and some abandoned paths of development still have special application or renewed relevance. Modern digital computers are general-purpose machines computing any classical computable function within the limits of their fixed hardware resources. Contrast this with the field-programmable gate array (FPGA) or the application-specific integrated circuit (ASIC). FPGA hardware can be configured to support arbitrary architectures within speed and area limits, effectively allowing software to be mapped directly to representation in hardware. ASIC hardware, on the other hand, is optimized to compute a specific function very quickly or with minimal resources.

General-purpose digital computers greatly benefit from the *stored-program* concept originated by Von Neumann. Programs are stored in memory just like their respective input and output. Programs that are read from memory control architectural components, such as the arithmetic logic unit (ALU), and invoke state changes within a set of registers. It is natural to ask if a quantum stored program carries any advantage over a classical stored program. The answer appears to be negative. A deterministic quantum gate array must have orthogonal programs to implement distinct unitary gates [NC97]. Hence, to know with certainty what program has executed, the program register of a quantum gate array must be effectively classical. This implies that the machine language expressing a quantum program for a general-purpose quantum computing system will look like a machine language for a general-purpose digital computer and can be stored in a classical memory.

Registers store bit vectors within digital processors for direct input to architectural components. Load-store operations transfer register contents to and from memory. The quantum analog of these operations is called the *QRAM* model [BCS03]. Essentially, qubits in an arbitrary unknown state cannot be copied because this violates unitarity. We assume that individual qubits are accessible, so a classical pointer refers to each qubit. Vectors of pointers constitute a *quantum register.* Hence, the quantum register is an abstract classical interface to a collection of quantum subsystems. These subsystems may be individual physical qubits or large collections of qubits in

an encoded state that represents a single qubit, depending on the organization of the quantum computer.

Computer architecture explores trade-offs related to locality and parallelism within computation [PH98]. A significant fraction of a modern computer's architecture is devoted to extracting performance gains by predicting and exploiting parallelism. Practically speaking though, this process actually starts before the particular machine instructions exist. Multi-stage compilers perform many algorithmic optimizations to select appropriate space-time trade-offs and prepare an instruction sequence tailored to a particular hardware architecture.

However, quantum computer architecture is fundamentally different from modern computer architecture because quantum computers must be fault-tolerant. As discussed in Chapter 4, fault-tolerant design places stringent demands on a quantum computer architecture. Parallelism and locality are the most significant constraints, and these constraints are *physical* – they cannot be met by a software solution alone. Hence the problem of inserting fault-tolerance into a quantum algorithm requires an approach that crosses the hardware-software interface.

Furthermore, fault-tolerance requires gate overhead that is large from a practical point of view and classical control that is relatively complicated compared to an ideal quantum computer. This overhead and complexity is introduced via algorithmic replacement rules best handled by automated software tools.

6.2 Design Flow

A quantum computer design flow determines the order of phases of language translation that transform a general unitary operator into a sequence of physical operations to control a quantum computer. A design flow may also include phases for verifying the output of other phases of the flow or optimizing output of earlier phases. Each phase can potentially provide feedback for subsequent passes through the flow.

Figure 6-1 labels each phase of our proposed design flow. The flow behaves much like a modern compiler such as `gcc` [gcc04]. The *upper stage* of the design flow con-

sists of two phases that are described briefly here. The front end maps a quantum program, perhaps given by but not limited to a unitary matrix, into an intermediate quantum circuit over some universal basis of gates. The technology independent optimizer applies transformation rules to optimize the quantum circuit according to some metric. For example, some applications may demand a minimum number of qubits while others may require minimum circuit depth. The technology independent optimizer passes the optimized quantum circuit to the technology dependent optimizer in a *quantum assembly language* (QASM) that represents the quantum circuit exactly over some universal basis and includes classical control instructions. We will not consider the upper stage in greater detail here, but instead refer the reader to [SCA+04].

The *lower stage* of the design flow also consists of two phases. The technology dependent optimizer applies methods such as the Solovay-Kitaev approximation [HRC02] to decompose the input circuit into a circuit over a discrete universal set of gates that is appropriate for the target technology. Furthermore, swap gates or movement instructions need to be inserted at this stage because the quantum computation takes place in a geometrically constrained system. The output of the technology dependent optimizer is a quantum circuit expressed in a *quantum physical operations language* (QPOL) for execution on a particular system. The final phase of the design flow simulates all or part of the quantum circuit to provide feedback such as execution time and reliability.

One strength of the design flow is that the hardware-software coupling need not be specified. The target quantum computing system might be a general-purpose quantum computer or an application-specific factoring engine. Either way, the design flow accommodates each system's design.

The technology-dependent optimization phase of the design flow produces instructions that contain a great amount of detail about how to physically execute the quantum algorithm. In other words, given the output of the technology-dependent phase of the design flow, a complete description of the quantum computing system only requires a description of the physical system in terms of its Hamiltonian(s), its

geometry, and the details of its supporting digital systems.

Section 6.1 argued that the requirements for fault-tolerance are physical requirements, so they must be supported by hardware. This means that the design flow crosses the hardware-software interface, hence a design flow implementation solves a hardware-software co-design problem. The location of the hardware-software interface determines the complexity of the underlying quantum computer architecture.

The hardware-software interface lies somewhere within the technology-dependent optimization phase. The placement of the interface determines how the design flow treats fault-tolerance, and appears to be related to the conceptual separation of logical and physical qubits. For example, if the interface is at the top of the phase, the lower stage is *architecture-driven*, whereas if the interface is at the bottom of the phase, the lower stage is *compiler-driven*.

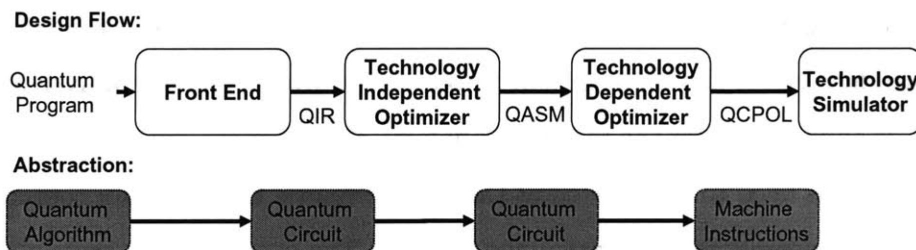


Figure 6-1: The four-phase design flow accommodates most approaches to quantum computer system design. Each phase maps between languages representing quantum circuits with varying degrees of technology dependence.

An *architecture-driven design process* inserts fault-tolerant gates from a pre-designed library during technology-dependent code generation. A design team creates the library of universal, fault-tolerant, technology-specific components using a combination of replacement rules, heuristic methods, and device models, and publishes the library together with design rules for connecting the composite components.

A *compiler-driven design process* inserts fault-tolerant gates during the technology-independent code generation using replacement rules based on quantum circuits. During technology-dependent code generation, sophisticated schedulers and layout tools insert QPOL instructions to preserve fault-tolerance. Algorithmic optimizations make fine-grained replacements, and feed-back from simulators may be used to focus the

optimizers on critical regions of the circuit.

6.2.1 Quantum assembly language

A quantum assembly language (QASM) expresses quantum circuits input to the technology dependent optimization phase of the design flow. A quantum assembly language is a language that can unambiguously represent quantum circuits intermixed with classical computing and control.

The term "quantum assembly language" is chosen to draw an analogy with modern digital assembly languages. This suggests that quantum assembly languages are not high level languages. Also, quantum assembly languages might formally resemble digital assembly languages. The analogy should not be taken too far though, because unlike their digital counterparts, quantum assembly languages are technology independent.

Quantum bits are static resources in a quantum assembly language. A given program declares a fixed number of named initial qubits. We assume these resources are initialized to the zero state on declaration. In contrast, classical resources can be dynamic in a QASM.

The quantum assembly language should subsume the descriptive power of a classical assembly language. One alternative is to build the QASM around the MIPS assembly language. This RISC assembly language controls a virtual processor that has load-store operations, arithmetic operations, and branching operations.

A quantum assembly language must exactly express *ideal quantum circuits*. Ideal quantum circuits consist only of unitary gates on one or more qubits and generalized measurements. Rather than choose a discrete basis for the unitary gates, we allow a QASM to declare arbitrary unitary gates on one, two, or three qubits, provided that the matrix elements of these gates have concise mathematical descriptions. A set of symbols must be included in the language to allow families of irrational matrix elements.

Finally, quantum measurements modify the classical virtual machine state. Each measurement operation must specify a classical register in which to store the result.

The result overwrites all of the bits in the register, even if only a few measurement outcomes are possible. The result now becomes part of the classical computation. As such, it can be used to conditionally execute future quantum gates and measurements.

6.2.2 Quantum physical operations language

A quantum physical operations language (QPOL) describes precisely how a given quantum circuit should be executed on a particular physical system. Each physical system has a potentially distinct physical operations language, so we first describe the general properties of a physical operations language, then move to a more detailed example for an ion-trap quantum computer.

General properties

A quantum physical operations language specifies what physical operations a quantum system should execute to realize quantum algorithms. Physical operations are classified into five categories based on the DiVincenzo criteria: initialization, computation, communication, classical control, and system-specific operations. This classification admits devices that have realized quantum information processing to date, and may admit future systems describable within the quantum circuit model.

Initialization operations specify how to prepare the initial state of the system. These include operations to load qubits into the system and deterministically put those qubits into a known starting state. In practice this requires moving or creating the physical qubit carriers, which could be ions, photons, nuclear spins, etc. Each carrier's additional physical degrees of freedom must be controlled as well, perhaps by lowering the temperature through cooling.

Computation operations include both gates and measurements. For most physical systems, gates correspond to controlled electromagnetic pulses. Gate types and speeds depend strongly on the interaction that couples qubits, so typical systems permit only a limited set of gates. Measurement methods rely on coupling to a measurement apparatus and will be limited to particular observables in practice.

Movement operations control the relative distance between qubits, bringing groups of qubits together to participate in local gates. Some systems have stationary qubit carriers and will spend a majority of their time performing swap gates. Other systems have mobile qubit carriers, or perhaps a mixture of mobile and immobile carriers. These systems will have machine-specific movement instructions.

Quantum information processors will contain at least a subset of classical logic operations. In the simplest case, quantum processors will be controlled by external computers and have access to a complete classical instruction set. Future quantum processors, however, may have integrated classical logic with specific low-level functions and interfaces.

Finally, a physical operations language includes system-specific operations that may not fall into general categories. These instructions are likely to control other degrees of freedom of the qubit carriers or nondestructively detect the presence or absence of carriers.

A complete program, expressed using a QPOL, is a sequence of instructions together with appropriate synchronization primitives. Each instruction is tagged for distribution to one of the (many) available instruction processing units in a quantum computer.

A QPOL for ion-trap quantum computers

Trapped-ion devices use charged, electromagnetically trapped atoms as qubit carriers. Each qubit is represented by internal electronic and/or nuclear states of a single ion. Laser pulses of specific frequencies addressing one or more ions apply single and multi-qubit quantum gates. Laser pulses, appropriately tuned, can also perform measurement, by causing ions to fluoresce when they are in the $|0\rangle$ state. Two or more ions can be contained in a single trap, where they couple to each other through Coulomb repulsion, thus providing a qubit-qubit interaction through their collective vibrational modes. These modes can serve as a "bus" qubit, as long as ion temperatures are kept low, and vibrational states are controlled. We call ion-qubits *chained* if they are close enough to interact using the bus qubit. This bus qubit is also manipulated optically

using *sideband* laser pulses.

Trapped-ion systems have shown considerable potential as a future technology for quantum information processing. Several groups have demonstrated a universal set of gates and highly efficient measurement [DDV+03, CDB+95, SKHR+03, SKHG+03, DAKJ+01, DMW+98]. Further, experiments have demonstrated that static voltages can move ion-qubits between traps [RBKD+02], and that sympathetic cooling can be used to reduce heat after movement [BDS+03, KKM+00]. These primitives have been joined to accomplish deterministic quantum teleportation [RHR+04, CSB+04]. Together, all of these experiments offer a route toward a scalable system, possibly configured in a large micro-array akin to charge-coupled-devices [KMW02].

We suggest this instance of a QPOL targeted to ion traps, consisting of initialization, computation, movement, classical control, and system-specific instructions:

Initialization of an ion trap processor has two stages: loading of multiple ions into a special loading region, and laser cooling to reduce ion temperatures. Measurement followed by conditional rotations puts all qubits into the $|0\rangle$ state.

Computation with quantum gates is naturally described in terms of single-qubit rotations in the $\hat{x} - \hat{y}$ plane, achieved using pulsed laser excitation, and a controlled-phase gate between ions in the same trap, implemented using three sideband pulses. Chained ions may also participate in a multiply controlled phase gate, useful for creating large entangled states [SB02, SM99, SM00]. Qubit readout with a laser pulse is described by a projective measurement.

Movement of ions is accomplished into and out of traps (and chains) using electrostatic fields. We assume a set of movement instructions sufficient for a planar rectangular trap configuration with "T" and "X" junctions. Additional splitting and joining instructions separate and rejoin chains.

Classical control of ions is assumed to be universal, and implemented by a fast, reliable, external classical processor. In practice, this can either be a remote control PC, or a local microprocessor chip integrated nearby the trap.

System-specific instructions for trapped ions are necessary to deal with the heating and decoherence of ion-qubits and bus qubits caused throughout a computation, in

the movement, splitting, and joining operations. In the worst case, high temperatures may eject ions from the trap. Thus, the instruction set includes a system-specific method to reinitialize the bus qubit, using recooling pulses. These are also sideband pulses like those used in multi-qubit gates, but they are applied differently and must be treated specially by the design tools.

6.3 Simulation Software Implementations

The design flow's practical embodiment is as a set of interoperable software tools. Some of these tools are compiler phases, and some are architecture design tools, such as CAD tools, schedulers, and simulators. In this thesis, we are primarily concerned with how the system design of a quantum computer is influenced by the need for fault-tolerance. For this reason, we focus on the lower stage of the design flow, particularly on simulation tools and some aspects of the technology dependent optimizer.

We have based all of the simulation tools presented in this section on Aaronson's implementation of CHP [AG04]. The general concepts of this simulation method are described in Chapter 2. The first subsection describes a graphical ion trap simulator, and the second subsection describes a more general quantum computer architecture simulator. Both simulation tools provide results like those in Chapter 5.

6.3.1 Ion-trap Simulator (ITSIM)

The ion-trap simulator, or ITSIM, uses the efficient simulation method reviewed in Chapter 2 and the ion-trap model described in Section 5.2 to determine threshold estimates for fault-tolerant quantum error-correction with trapped-ions. We give an overview of this simulator and describe its design and interfaces. Appendix B lists all of the source code for ITSIM.

ITSIM simulates arbitrary quantum error-correction circuits and fault-tolerant $\mathcal{C}_2$ gates with locality and ion movement constraints imposed by the trapped-ion quantum charge-coupled device architecture [KMW02]. The specific model ITSIM implements is summarized in Table 5.1. ITSIM computes the discrete time evolution

of a quantum stabilizer state. A pseudorandom number generator applies the probabilistic noise model discussed in Chapter 4. Comparing the final quantum state to the expected quantum state yields a pass or failure outcome, and these outcomes are used to compute the failure probability of the quantum circuit. Each virtual classical control processor is attached to its own software thread. Each thread sums operation times to give a total time for the fault-tolerant computation.

ITSIM is written almost entirely in an interpreted object-oriented language called Python [pyt04]. Python has many standard and contributed high-level modules that hasten development and testing of large projects. For example, ITSIM makes use of preexisting modules for threading, three-dimensional graphics, and compiler building. When existing modules are too slow, Python allows users to develop their own modules in a compiled language such as C. We used this feature of Python to incorporate parts of CHP [AG04].

One way to verify that a technology-dependent optimizer has produced correct code is simply to look at the sequence of operations. ITSIM graphically displays each QPOL instruction in three dimensions. Laser pulses and ion movements are all represented within the animation. The combined use of a visual extension called VPython and Python's built-in threading allow these operations to occur and be visualized in parallel as specified by the ITSIM user.

In addition to the ITSIM core, supporting Python scripts emulate parts of the technology-dependent optimizer. These scripts construct layouts and recursively simulate stabilizer circuits. Additional supporting scripts help collection, process, and plot simulation results for various types of numerical experiments. A final set of scripts creates standalone animations from ITSIM graphics.

ITSIM's entirely modular design mirrors the elements of the trapped-ion quantum computer described in Chapter 5. In addition, the software can interpret a QPOL program, provided as a Python binary object, and simulate the implied noisy dynamics while providing graphical feedback. The software allows graphical output to be toggled so that strictly quantitative simulations are possible.

A modular design decouples elements of the application, as shown in Figure 6.3.1.

This module dependency diagram shows precisely how the software design mirrors the envisioned trapped-ion quantum computer. The `iontrap` module is the focal point of the diagram. This module constructs ion-trap objects given a trap geometry (a matrix) and a set of physical parameters such as gate failure probabilities and times provided by the `physics` module. The `iontrap` object constructs a `grid` of `cells` to graphically represent the ion-trap. Simultaneously, the `iontrap` object constructs `ion` objects that can be joined into `chains`. Finally, the user attaches a `bundle` of physical operations to be executed. The `bundle` executes methods in the `control` object contained within the `iontrap`. These direct ion movements, execute gates through `aqc`, and update the graphical representation on the `grid`.

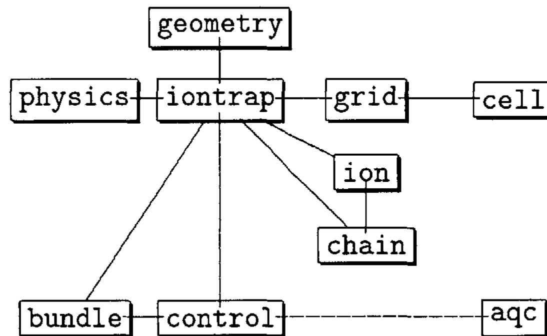


Figure 6-2: The ITSIM module dependence diagram (MDD) includes 10 different components. Lines connecting the modules signify inclusion, instantiation, or some other dependence. See appendix B for a more detailed MDD.

Users interact with ITSIM like it is a virtual quantum computer. This means that users provide a QPOL program and a layout describing the quantum computer to ITSIM. ITSIM's parser processes the QPOL program and the layout and writes a `bundle` object and a layout matrix to files. Think of the bundle object as machine code for the simulator. The user then sets the ion trap parameters through the `physics` object and passes the bundle to ITSIM. During the simulation, ITSIM shows an animation of the QPOL program running on the layout. Finally, the simulator prints a line to the screen and to a file that reports success or failure each time the QPOL program runs. This file is later processed to give the failure probability of the QPOL program running on the layout with the given physics parameters.

The QPOL input file has a layout section and a code section. The layout section begins with a sequence of commands specifying free and occupied regions of the layout. The end of the layout section has collection of statements giving each ion's name, location, type, and initial chain configuration. The code section begins by defining named subroutines and later organizes these subroutines into a program. The code has instructions for measurements, gates, movements, and limited classical control. These instructions are grouped into blocks using different kinds of braces. The braces distinguish between serial and parallel executing blocks of code and can be nested arbitrarily. Ions names must correspond to those declared in the layout section, whereas classical bits are created as they are used.

Consider the following example of a QPOL program for ITSIM:

```
# File: teleport.qcpol
# Author: Andrew Cross <awcross@mit.edu>
# Last Modified: 4 April 2004
#
# Teleport a single ion state

<layout>
#+-+-+-+
# | |A| |
# | | | |
# |     |
# | | | |
# |D|A| |
#+-+-+-+
grid (7,7)
fill (1,1) - (7,7)
empty (2,2) - (2,6)
empty (4,2) - (4,6)
empty (6,2) - (6,6)
```

```
empty (3,4),(5,4)
# braces indicate groups of ions that initially share an
# oscillator mode (they can all be involved in multi-qubit
# gates)
< ion d1, "data", (2,2) >
< ion a1, "ancilla", (4,2) >
< ion d2, "data", (4,6) >
</layout>

<qcpol>

# Subroutines to correct target qubit
def xd2 { message "x correct" gate "x", (d2) }
def zd2 { message "z correct" gate "z", (d2) }

# Create epr pair
# write a message to the simulator log
message "create epr pair"
# move data qubit d2 to position (4,3)
move d2, (4,3)
gate "h", (a1)           # apply a hadamard gate to a1
# join a1's linear chain with d2's chain
join a1, d2
# apply a controlled not gate; a1 is control
gate "cnot", (a1,d2)

# Distribute it
message "distribute pair"
move d2, (4,4)
split a1, d2             # separate the linear chain at a1-d2
```

```
[ < move d2, (6,4)        # execute these two move instructions
    move d2, (6,6) >      # sequentially, but in parallel with
  < move a1, (4,4)        # these three movement instructions
    move a1, (2,4)
    move a1, (2,3) > ]

# Teleport
message "teleport"
join d1, a1
gate "cnot", (d1,a1)
gate "h", (d1)
split d1, a1
# measure qubit d1, store the result in cz
readout d1, cz
readout a1, cx
condition (cx), (1), xd2          # if cx == 1 call xd2
condition (cz), (1), zd2

</qcpol>
```

This example demonstrates quantum teleportation within a planar ion-trap. First, the example creates a planar layout with three ions. Gates place the first two ions into an entangled state. The second ion is moved to meet the third ion. A sequence of gates and measurements teleports the third ion's state to the first ion.

During the simulation, users may choose to see an animation like that shown in Figure 6-3. The animation shows each instruction in the QPOL input file as it executes. The instruction is printed on the display as well. Ions are represented by spheres and gates by lines representing laser pulses. Ions move within the black regions of the figure. The lighter regions depict electrodes.

After the simulation, ITSIM writes a file to disk with a list of outcomes. Each

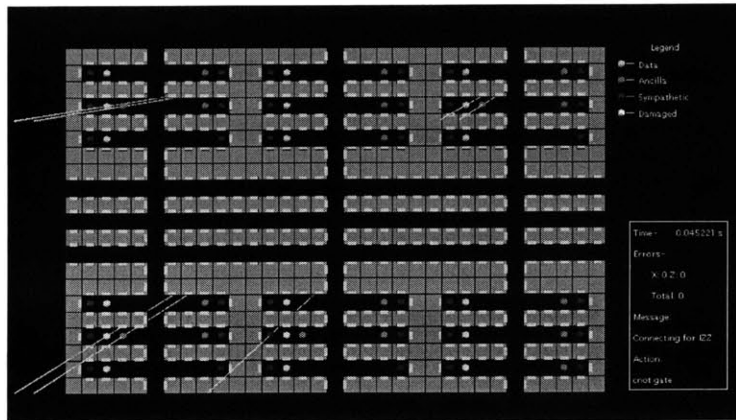


Figure 6-3: Snapshot of the ITSIM graphical display showing an H-tree layout. Qubits are ions represented by spheres, and gates are applied using laser pulses, represented by lines. The qubits can move within the black regions of the figure and are prohibited from moving into the substrate which is drawn using light squares. In the right window the simulator displays feedback regarding the current operations, noise induced failures, and estimated execution time.

outcome is either a pass or a failure depending on the faults that occurred during a particular simulation. A large number of outcomes can be used to estimate the failure probability of the QPOL program for the given set of physical parameters.

6.3.2 Quantum Architecture Simulator (ARQ)

ITSIM helps system architects visualize the operations that take place in a fault-tolerant trapped-ion quantum computer, and gives reliability results within hours or days for quantum codes that are not concatenated. However, even after optimizations, ITSIM cannot gather data quickly enough to study the $[[7, 1, 3]]$ quantum code, for example. Threading, ion grouping in the chain module, and Python's overhead contribute the most to the performance problem.

The quantum architecture simulator (ARQ) is a rewrite of ITSIM that addresses the performance problem. ARQ is more than a rewrite, however, because it generalizes the quantum computer architecture model to encompass most if not all of the physical systems under consideration for large-scale quantum computation. This subsection gives an overview of ARQ together with its design and interfaces. Appendix C lists the complete source.

The quantum architecture simulator (ARQ) is a general tool for studying local fault-tolerant quantum computation within the stabilizer formalism. Like ITSIM, ARQ simulates a QPOL quantum program that runs on a quantum computer with a particular layout. ARQ can also compute the circuit reliability and the quantum state evolution. Supporting scripts produce circuit diagrams and animations from the detailed logs that ARQ outputs.

Unlike ITSIM, ARQ implements several *virtual quantum machines* rather than a single ion-trap quantum computer. Each machine interprets a different, but related, QPOL input language. Because of the hierarchical design, new machines are relatively straightforward to add.

The core ARQ simulator is written entirely in C++. Separate scripts written in Python process ARQ's output log to report the quantum state, animate the quantum processor behavior, create postscript circuit diagrams, and construct QPOL source files. A separate C++ program executes ARQ to run numerical experiments and process the resulting data.

To be a useful engineering tool, ARQ must simulate noisy fault-tolerant circuits based on small doubly concatenated quantum codes in under a month. Simply rewriting ITSIM in C++ without using threads and qubit groups (i.e., chains, see Chapter 5) meets this goal. In fact, it seems that substantial future improvements over ARQ must come from algorithmic improvements in the stabilizer simulation method.

To be a useful quantum computer architecture simulator, ARQ must simulate many different system models. The solution we have chosen for this problem is a hierarchy of virtual quantum machines. This hierarchy provides classical control, Clifford group quantum operations, and planar movement as building blocks for more detailed system models. Such a design permits study of fault-tolerant quantum circuits in local or nonlocal settings using generic or system specific models.

Each machine in the virtual machine hierarchy, shown in Figure 6.3.2, builds on the machines that lie beneath it in the hierarchy. A classical control machine (`cc`) sits at the base of the hierarchy. This machine implements classical bits, boolean gates, and branching. Although tedious, any classical program can be written in the

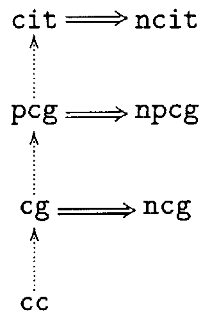


Figure 6-4: The ARQ virtual quantum machine hierarchy contains 7 machines. The *classical control* (`cc`) machine lies at the bottom of the hierarchy and implements conditions and branching. The *Clifford group* (`cg`) machine includes the classical control machine and adds measurement, CX, CZ, H, S, X, Y, and Z gates. The *planar Clifford group* (`pcg`) machine adds a layout and the accompanying movement and placement instructions. Finally, the *Clifford ion trap* (`cit`) machine adds ion-trap specific instructions such as cooling. Each machine has a noisy equivalent (`ncg, npcg, ncit`).

language of the `cc` machine. For example, a simple counter written in the `cc` language looks like:

```
# halfadder.arq - example of classical control
# with subroutines
# Andrew Cross <awcross@mit.edu>
#
# calls a two-bit half-adder subroutine to add 1 to a two-bit
# "register" until the register equals 3
        bit high0, low0, high1, low1
        set low1,1
        label top
        if high0, low0
        jump end
        call halfadder
        jump top
# half adder subroutine
# expects: high0, low0,
```

```
#             high1 , low1
# Adds (high0,low0) and (high1,low1)
# putting the result in (high0,low0).
        label halfadder
        xor high0,high0,high1
        if low0, low1
        xor high0,high0,1
        xor low0,low0,low1
        return
        label end
```

If the reader is familiar with assembly language then this example is self-explanatory. The first line declares classical bits, not registers. Bits can be toggled with the `set` instruction. Labels, jumps, calls, and if statements allow branching. Finally, boolean operations such as `xor`, `and`, and `or` are implemented by `cc`.

The `cg` machine includes the `cc` machine and adds quantum gates and measurements within the stabilizer formalism. Again, we will give an example of the `cg` machine's QPOL language. This example creates one of the ancilla states used in a fault-tolerant recovery operation for the $[[7,1,3]]$ code (see Figure 2-3):

```
# 7encode.arq
qubit q0,q1,q2,q3,q4,q5,q6
h q0; h q1; h q3
cnot q1,q2 ; cnot q3,q4
cnot q0,q2; cnot q4,q5
cnot q0,q4; cnot q5,q6
cnot q1,q5
cnot q1,q6
cnot q0,q6
logket
```

The example declares seven qubits on the first line, then a sequence of quantum

gates create the logical zero state of the $[[7,1,3]]$ quantum code. Semicolons separate operations that can occur in parallel, such as the group of Hadamard gates on the first three qubits. The `logket` command writes the quantum state in ket notation to ARQ's output log.

Finally, the `pcg` machine adds a layout and a movement operation `move qubit, direction, steps`. The layout is simply a list of integers that represent occupiable locations on the plane. The additional QPOL instructions are shown in the following example, which moves a qubit on a layout:

```
# movement example
qubit d1
init d1,2002
move d2,W,3
```

This example places a named qubit onto a layout at position $(2,2)$. The subsequent movement command moves this qubit left three units. Movement actions become part of models higher in the virtual machine hierarchy, where they are assigned times, failure probabilities, and perhaps some restrictions.

6.4 Conclusion

This chapter proposes an approach to quantum computer design based on an abstract design flow. We have started to implement that design flow from the bottom up by completing and demonstrating a new kind of quantum computer simulator. However, there are many directions for future work.

One direction relates to the speed of simulation tools. The speed of simulation tools like ARQ depend strongly on the algorithmic details of the quantum simulation. Finding improved algorithms would improve these tools greatly, perhaps making the difference between days and months.

Other phases of the design flow suggest future work. For example, general circuit rewriting tools to incorporate stabilizer-based fault-tolerance in an arbitrary quantum

circuit might be possible. These phases could be implemented in software as well so as to interoperate with tools like ARQ. In the process, the precise placement of fault-tolerance within the design flow may become more clear.

ARQ and ITSIM implement several system architecture models. Other models could be developed for physical systems not discussed in this thesis. In addition to creating new models, the existing models can be improved.

Chapter 7

Conclusion

This thesis can be viewed as an introduction to quantum fault-tolerance in the trenches. It has highlighted and taken steps toward solution of the complex problems that must be overcome for quantum fault-tolerant design to make the transition from theory to practice. To say there are directions for future research is a vast understatement, as each contribution reveals new challenges.

For quantum fault-tolerance to make the transition from a theoretical gem to an experimental reality, we need to create experimental techniques for designing and characterizing fault-tolerant quantum circuits. Because it is possible to experimentally observe the fidelity of quantum circuits while controlling, to some extent, the fidelity of basic components, we can conceivably witness fault-tolerant design improving the reliability of quantum circuits in the laboratory. In Chapter 4, we have developed a new numerical simulation method paralleling this idea. The method computes reliable parameters of a recursively simulated stabilizer circuit acting on fewer than 1000 qubits. Reliable parameters are basic component failure probabilities for which recursive simulation improves reliability of a fault-tolerant circuit. We have verified though numerical simulation that classical and quantum fault-tolerant circuits qualitatively obey equation 3.5,

$$\frac{p_L(p)}{p_{th}} \leq \left(\frac{p}{p_{th}}\right)^{(t+1)^L}.$$

Moreover, the numerical method extends easily to complicated models of proposed physical quantum computing systems with many failure probability parameters. The boundary between reliable and unreliable parameters for these systems and circuits estimates their fault-tolerance threshold, particularly at high levels of recursive simulation. Based on this concept that reliable parameters can estimate sub-threshold parameters, we computed threshold estimates for the $[[3,1,1]]$ bit-flip code, $p_{th} \approx 3.5 \times 10^{-2}$, and the $[[7,1,3]]$ code, $p_{th} \approx 7.2 \times 10^{-4}$.

However, the numerical results we observed differed quantitatively from equation 3.5. In particular, multiple crossing points are possible, and none of these correspond to the intersection of the 1-simulated circuit failure probability and the 0-simulated circuit failure probability. This led us to clarify how pseudo-thresholds, first identified in [STD04], arise in both classical and quantum fault-tolerant circuits. We computed pseudothresholds and thresholds for the classical TMR recovery network and the 3-qubit bit-flip recovery network. We applied basic dynamical systems theory to thoroughly characterize the classical TMR recovery network. For the 3-qubit network, we found that the threshold and pseudo-threshold could differ by as much as 84% of the actual threshold (see Table 4.2). Hence, Chapter 4 identified a core problem to experimental threshold studies. How do we verify that we have experimentally observed a threshold and not just a pseudo-threshold? Moreover, what is the largest difference between thresholds and pseudothresholds? These questions are a potentially interesting result in and of themselves, as current threshold estimates in the literature may differ significantly from the actual threshold.

Having built a conceptual and numerical framework in Chapter 4 for experimental evaluation of realistic models of fault-tolerant quantum computing systems, we observe that there are no realistic fault-tolerance threshold estimates for particular quantum computing systems. Specifically, there are no threshold estimates for systems realistically constrained to planar layouts except those estimates given by counting arguments. Trapped ion quantum computers are the most capable candidates for experimental quantum computing, so in Chapter 5 we create a system model for the trapped-ion quantum computer based on current experimental progress. Within this

system model, we investigate three potential layouts for ion-trap quantum computers, two of which have desirable fault-tolerance properties. Using concrete examples, we expose some of the complexities of the layout problem, and clarify how the demand for fault-tolerance influences layout. Finally, we compute reliable parameters for the $[[3,1,1]]$ and $[[7,1,3]]$ recovery networks when those networks are constrained to operate on a local layout [MCT+04]. The reliable parameters provide us with estimates of the corresponding fault-tolerance thresholds, in this case, $p_{th} \approx 3 \times 10^{-3}$ and $p_{th} \approx 1 \times 10^{-5}$, respectively.

Chapter 5 begins a particular program of research into quantum computer architecture. By constructing progressively more detailed and realistic system models of quantum computers, we can make predictions about the reliability of those systems under particular types of noise. Perhaps most importantly, we can begin to study system levels trade-offs associated with different layouts and communication methods [COI+03, MTC+05b, MTC+05a]. Such trade-offs can be explored through simulation for small codes as an extension of the existing body of work.

In order to study modestly complicated quantum computer systems, we need to enlist the aid of software design tools. Fault-tolerant quantum error-correction requires complicated quantum circuits and classical control, and the need for fault-tolerance introduces new problems when compared to classical computer architecture and circuit synthesis. How can we realize and evaluate a large-scale quantum computer, such as an ion-trap quantum computer described in Chapter 5, given the complexity of fault-tolerant design? Chapter 6 proposed a four phase design flow [SCA+04] that transforms a quantum algorithm written in a high level language into a sequence of physical operations. We have implemented the lowest phase of this design flow – software simulators for evaluating quantum fault-tolerance. The preceding chapters demonstrated these simulators, and we have provided the complete source code in the appendices of this thesis. The design flow gives a greater context to the simulators, placing them in a complete software architecture for quantum computer design tools, and gives insight into the synthesis problem looming just above them by exposing the physical operations interface.

The design flow is a scaffolding in which to organize future quantum architecture research. There is much work to do to realize the design flow as a set of interoperable software tools. There are still questions regarding the ideal placement of fault-tolerance within the flow, and precisely where the quantum computer software should end and the computer architecture begin. Indeed, there is a vast realm to explore between a concise description of Shor's algorithm and the pulse that implements the final measurement on a large-scale fault-tolerant quantum computer.

Appendix A

Notation

$[[n,k,d]]$	quantum code encoding k qubits in n that corrects $t = \lfloor \frac{d-1}{2} \rfloor$ errors
$\mathcal{H}$	single qubit Hilbert space
$\mathbb{Z}_2$	integers modulo 2
$SU(n)$	special unitary group of dimension n
$[X,Y]$	commutator bracket $XY - YX$
$\{X,Y\}$	anti-commutator bracket $XY + YX$
$\|0\rangle, \|1\rangle$	computational basis of $\mathcal{H}$
H	Hadamard gate $\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$, or parity check matrix
S	$\pi/2$-phase gate $\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$
X, Y, Z	single qubit Pauli operators
$\|0_L\rangle$	basis vector of a logical Hilbert space
C	classical error correcting code
$C^{\perp}$	dual space of a classical code
$\mathcal{C}$	quantum error correcting code
$\mathcal{C}_*$	quantum computation code with gates $\langle \mathcal{C}, \mathcal{B}, E, D, R \rangle$
E	encoding operation for a quantum computation code
D	decoding operation for a quantum computation code
$\mathcal{B}$	discrete universal set of fault-tolerant gates for a quantum computation code

R	fault-tolerant recovery operation for a quantum computation code
$\mathcal{G} = \mathcal{C}_1$	single qubit Pauli group $\langle X, Z, iI \rangle$
G	noiseless quantum circuit
$X \otimes Y$	tensor product of Pauli operators
S	stabilizer subgroup of the n-qubit Pauli group $\mathcal{G}^{\otimes n}$
$\vert\psi_S\rangle$	vector subspace stabilized by S
$\mathcal{O}(x)$	big-oh notation, $f(x) = \mathcal{O}(x) \iff f(x) \leq Cx \quad \forall x > x_0, \quad \text{some } C > 0$
$\mathcal{C}_k$	recursive hierarchy of sets $\{U \in SU(2^n) \mid UgU^\dagger \in \mathcal{C}_{k-1} \quad \forall g \in \mathcal{C}_{k-1}\}$
L	level of recursive simulation
p, p_0	failure probability of a basic component
p_L	failure probability of an L-simulated circuit
p_{th}	threshold failure probability
$\mathcal{E}(\rho)$	quantum operation
$\mathcal{N}$	noise model (sequence of quantum operations)
$G_{\mathcal{N}}$	noisy quantum circuit
P	set of failure probabilities
G_L	noiseless fault-tolerant quantum circuit obtained from G by L-simulation
$\mathcal{F}(\rho, \sigma)$	fidelity between density matrices, $F(\rho, \sigma) = \mathrm{tr}\sqrt{\sqrt{\rho}\sigma\sqrt{\rho}}$
$D(\rho, \sigma)$	trace distance between density matrices, $D(\rho, \sigma) = \mathrm{tr}(\vert\rho - \sigma\vert)$
$\mathcal{F}(U, \mathcal{E})$	gate fidelity
$\mathbb{P}(E)$	probability of event E

Appendix B

Ion-Trap Simulator (ITSIM) Source Code

B.1 Module Dependencies

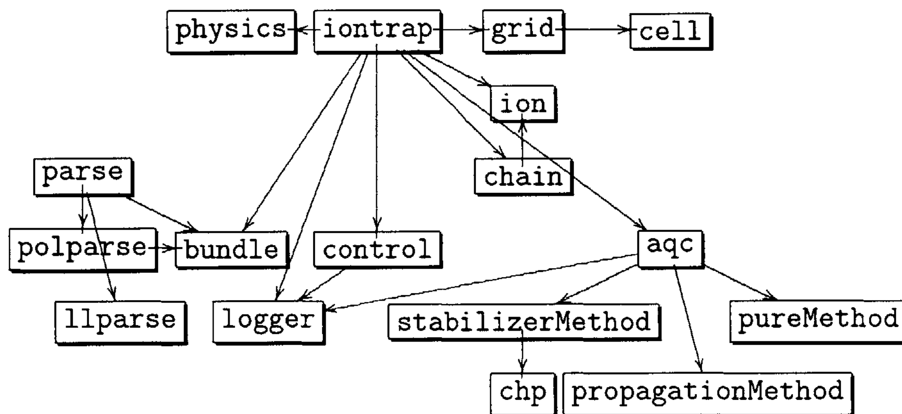


Figure B-1: The full ITSIM module dependence diagram. Arrows represent Python `import` dependence. The module at the arrow tail imports the module at the arrow head.

Figure B.1 shows how the modules of ITSIM depend on one another. The `iontrap` coordinates ITSIM by creating a graphical display (`grid`), ions (`ion, chain`), qubits (`aqc`), and a controller (`control`). Programs enter ITSIM through `parse`, which converts them to an intermediate representation called a `bundle`.

B.2 Source Listing

B.2.1 aqc.pt

```
# Name: aqc.py
# Authors: Andrew Cross <awcross@mit.edu>
# Last Modified: 26 March 2004
#
# Abstract quantum computer with a standard
     interface

import string
import random
from Numeric import *

import logger
from stabilizerMethod import *
from propagationMethod import *
from pureMethod import *

class aqc:

    """Abstract quantum computer"""
    def __init__(self,names,smode='stabilizer',
        log=None):

        if not type(names) is list: raise "
            qcInitFailed", names
        for q in names:
            if not type(q) is str: raise "
                qcInitFailed", q

            # see noise() for noise types
            self.noisy = False
            self.noiseType = "depolarize"
            self.method = None
            self.mode = smode

            if log == None:
                self.log = logger.logger('qc.
                    default.log',0)
            else: self.log = log

            if smode == 'stabilizer':
                self.method = stabilizerMethod(
                    names)
                self.method.reseed()
            elif smode == 'propagation':
                self.method = propagationMethod(
                    names)
            elif smode == 'pure':
                self.method = pureMethod(names)
            else: raise "qcInitFailed", smode


# Added By Setso
#
    def displayket(self):
        """ displays the ket of the system's
            state
        (NOTE: can be enourmous)
        """
        self.method.displayket()


    def message(self,msg):

        self.log.put("aqc.message","%s"%msg)


    def fidelity(self,other):

        if self.mode != other.mode: raise "
            incompatible"
        f = self.method.fidelity(other.method)
        self.log.put("aqc.fidelity","%f"%f)
        return f

    def same(self,names,initstring):

        c = self.method.same(names,initstring)
        self.log.put("aqc.same",["different","
            same"][c])
        return c

    def reset(self,names):
        """Noiseless reset"""
        self.log.put("aqc.reset",",".join(names)
            )
        self.method.reset(names)
```

```
    def noise(self,name,p):
        """Applies noise with probability p to
            the given qubit"""
        if random.random() < p:
            if self.noiseType == "depolarize":
                r = random.random()
                if r < 1.0 / 3.0:
                    self.log.put("aqc.noise","x
                        %s"%name)
                    self.method.gates(["x"],[
                        name])
                elif r < 2.0 / 3.0:
                    self.log.put("aqc.noise","y
                        %s"%name)
                    self.method.gates(["y"],[
                        name])
                else:
                    self.log.put("aqc.noise","z
                        %s"%name)
                    self.method.gates(["z"],[
                        name])
            elif self.noiseType == "bitflip":
                self.method.gates(["x"],[name])
                self.log.put("aqc.noise","x %s"%
                    name)
            elif self.noiseType == "phaseflip":
                self.method.gates(["z"],[name])
                self.log.put("aqc.noise","z %s"%
                    name)
            else: raise "badNoise", self.
                noiseType

    def gates(self,ops,qubits,pfails=None):
        """Applies a sequence of gates to a set
            of qubits

        For example, ops=['h','cnot'], qubits=['
            q1',['q2','q3']],
        pfails=[0.1,0.1]. The optional parameter
            pfails assigns
        failure probabilities to the list of
            gates. In the case of
        a two qubit gate, pfail gives the
            probability of a failure
        on each of the qubits involved. """
        if pfails==None: pfails=len(ops)*[0]
        if self.noisy:
            for g in range(len(ops)):
                if type(qubits[g]) is list:
                    for q in qubits[g]:
                        self.noise(q,pfails[g])
                    else: self.noise(qubits[g],
                        pfails[g])
            for g in range(len(ops)):
                if type(qubits[g]) is list:
                    self.log.put('aqc.gates','%s
                        %s'%
                                (ops[g],",".
                                    join(
                                    qubits[g])
                                    ))
                else:
                    self.log.put('aqc.gates','%s
                        %s'%
                                (ops[g],qubits[
                                    g]))
            self.method.gates(ops,qubits)

    def measure(self,qubits,pfail=None):
        """Projectively measure a set of qubits

        Measurement failure is equivalent to
        an error occurring just before the
        measurement itself.

        Returns a list of measurement results.""
            "
        if pfail == None: pfail = [0]*len(qubits
            )
        if self.noisy:
            for i in range(len(qubits)):
                self.noise(qubits[i],pfail[i])
        outcomes = self.method.measure(qubits)
        self.log.put('aqc.measure','%s %s'%(
            qubits,outcomes))
        return outcomes

    def add(self,names):
        """Adds qubits to the machine

        Takes a list of qubit objects."""
```

```
        self.log.put('aqc.add',names)
        self.method.add(names)

    def remove(self,names):
        self.log.put('aqc.remove',names)
        self.method.remove(names)

    def statestr(self):
        return(str(self.method))

    def __str__(self):
        sl = []
        sl.append("mode = %s\n"%self.mode)
        sl.append("noisy = %s\n"%self.noisy)
        sl.append("noiseType = %s\n"%self.
            noiseType)
        sl.append(str(self.method))
        # .join segfaults sometimes ...
        return "".join(sl)

# Test method : Fault-tolerant [7,1,3] quantum
    code without
#                  ancilla verification
if __name__ == "__main__":

    import time

    # Encode the logical |0>
    def encode7(myqc):
        # d0 is the qubit to be encoded
        # here, d0 is |0>
        for i in [4,5,6]:
            myqc.gates(["h"],["d%d"%i])
        for p in [(0,1),(0,2),(6,3),(6,1),(6,0)
            ,
                  (5,3),(5,2),(5,0),(4,3),(4,2)
                  ,
                  (4,1)]:
            myqc.gates(["cnot"],[["d%d"%p[0],"d
                %d"%p[1]]])

        for i in range(3*7):

            simmode = "stabilizer"
            q = aqc(["d0","d1","d2","d3","d4","
                d5","d6",
                    "a0","a1","a2","a3","v0"],
                        simmode)
            q0 = aqc(["d0","d1","d2","d3","d4",
                "d5","d6",
                        "a0","a1","a2","a3","
                            v0"],simmode)
            q.noisy = False
            S = ['IIIZZZZ','IZZIIZZ','ZIZIZIZ',
                'IIIXXXX','IXXIIXX','XIXIXIX']
            sX,sZ=[],[]
            nd,na = 7,4

            encode7(q)
            encode7(q0)

            # apply an error to the data
            if i < 7:
                print "x on qubit d%d"%i
                q.gates(["x"],["d%d"%(i)])
            elif i < 14:
                print "z on qubit d%d"%(i-7)
                q.gates(["z"],["d%d"%(i-7)])
            else:
                print "y on qubit d%d"%(i-14)
                q.gates(["y"],["d%d"%(i-14)])

            # detect and locate the error
            for s in S:
                s = string.upper(s)
                # initialize ancilla
                for a in ["a0","a1","a2","a3"]:
                    if q.measure([a])[0]: q.
                        gates(["x"],[a])
                # prepare shor state for "Z"
                    generators
                q.gates(['h'],['a0'])
                q.gates(['cnot'],[['a0','a1']])
                q.gates(['cnot'],[['a1','a2']])
                q.gates(['cnot'],[['a2','a3']])
                if 'Z' in s:
                    for a in ["a0","a1","a2","
                        a3"]:
                        q.gates(['h'],[a])
                # Data Interaction
                anc = 0
                for i in range(nd):
                    if s[i] == "Z":
                        q.gates(['cnot'],[["d"+
                            str(i),"a"+str(anc
                            ),]])
                        anc += 1
                    elif s[i] == "X":
                        q.gates(['cnot'],[["a"+
                            str(anc),"d"+str(i
                            )]])
                        anc += 1
                # Change measurement basis for
                    "X" generators
                if 'X' in s:
                    for a in ["a0","a1","a2","
                        a3"]:
                        q.gates(['h'],[a])
                # Measure parity of the four
                    ancilla qubits
                count, paritylist = 0, q.
                    measure(["a0","a1","a2","
                    a3"])
                print "%s -> %s"%(s,paritylist)
                for i in paritylist: count =
                    count + i
                if "Z" in s:
                    if (count % 2)==1: sX.
                        append(1)
                    else: sX.append(0)
                if "X" in s:
                    if (count % 2)==1: sZ.
                        append(1)
                    else: sZ.append(0)

            print "syndrome [X,Z]: %s"%(sX+sZ)
            if 1:
                # error correct
                xloc = sX[2] + 2*sX[1] + 4*sX
                    [0]
                zloc = sZ[2] + 2*sZ[1] + 4*sZ
                    [0]
                if xloc > 0: q.gates(['x'],['d%
                    d'%(xloc-1)])
                if zloc > 0: q.gates(['z'],['d%
                    d'%(zloc-1)])
            # reset ancilla
            for a in ["a0","a1","a2","a3"]: q.
                reset([a])
            print "machine fidelity:", q.
                fidelity(q0)
            print "machine compare:", q.same(q0
                )
            print q.method.generators()
            print q0.method.generators()
```

B.2.2 bundle.py

```
# File: bundle.py
# Author: Andrew Cross <awcross@mit.edu>
# Last Modified: 27 March 2004

import thread
import threading
import copy

# USE TYPECHECKING to simplify the interface!!

# There are special cases here
# that depend on action implementation in
# control.py:
#        "halt" raises the HaltAction exception
# And that depend on how we display data:
#        message updates

# The base class cannot be created, but it
# implements functions common to all bundles.
# Each derived bundle must implement __init__
# and start().
class bundle:

    def __init__(self):
        # bundle's self.time should be *total*
            time elapsed.
        self.time = 0
        self.halted = False
        self.sys = None


    def collapse(self):
```

```
        # easy version, replace single-edged
            vertices with their child
        if self.type == 's' or self.type == 'p':
            for b in self.bundles: b.collapse()
            for j in range(len(self.bundles)):
                b = self.bundles[j]
                if b.type == 's' or b.type == 'p
                    ':
                    if len(b.bundles) == 1:
                        self.bundles[j] = b.
                            bundles[0]

def __str__(self):
    sl = []
    if self.type == 's' or self.type == 'p':
        if self.type == 's': sl.append('\n<')
        else: sl.append('\n[')
        for b in self.bundles:
            if b.type == 's' or b.type == 'p':
                sl.append(str(b))
            elif b.type == 'a':
                if b.bundles[0][0] != '
                    condition':
                    sl.append(b.bundles[0][0])
                    sl.append(' ')
                    for t in b.bundles[0][1]:
                        if type(t) is list:
                            sl.append('(')
                            for x in t:
                                sl.append(str(
                                    x))
                                sl.append(',')
                            sl.pop()
                            sl.append(')')
                        else: sl.append(str(t)
                            )
                        sl.append(',')
                    sl.pop()
                    sl.append('\n')
                else:
                    sl.append("condition (")
                    for x in b.bundles
                        [0][1][0]:
                        sl.append(str(x))
                        sl.append(',')
                    sl.pop()
                    sl.append('),(')
                    for x in b.bundles
                        [0][1][1]:
                        sl.append(str(x))
                        sl.append(',')
                    sl.pop()
                    sl.append('), jump_bundle\
                        n') # b.bundles
                        [0][1][2] is jump
                        target bundle
            else: raise 'InvalidType', self.
                type
        if self.type == 's': sl.append('>')
        else: sl.append(']')
    elif self.type == 'a':
        if self.bundles[0][0] != 'condition':
            sl.append(self.bundles[0][0])
            sl.append(' ')
            for t in self.bundles[0][1]:
                if type(t) is list:
                    sl.append('(')
                    for x in t:
                        sl.append(str(x))
                        sl.append(',')
                    sl.pop()
                    sl.append(')')
                elif type(t) is str:
                    sl.append('"')
                    sl.append(str(t))
                    sl.append('"')
                else: sl.append(str(t))
                sl.append(',')
            sl.pop()
        else:
            sl.append("condition (")
            for x in self.bundles[0][1][0]:
                sl.append(str(x))
                sl.append(',')
            sl.pop()
            sl.append('),(')
            for x in self.bundles[0][1][1]:
                sl.append(str(x))
                sl.append(',')
            sl.pop()
            sl.append('), jump_bundle') # b.
                bundles[0][1][2] is jump
                target bundle
            sl.append('\n')
    else: raise 'InvalidType', self.type
    return "".join(sl)

def setActionMap(self):
    """Set the action map of all sub-bundles
        """
    # This is slow - call it once to
        initialize.
    if self.type == 's' or self.type == 'p':
        for b in self.bundles:
            if b.type == 's' or b.type == 'p
                ':
                b.actionmap = self.actionmap
                b.setActionMap()
            elif b.type == 'a':
                b.actionmap = self.actionmap
                if b.bundles[0][0] == '
                    condition':
                    subBundle = b.bundles
                        [0][1][2]
                    # this test should
                        prevent needless
                    # recursion and loops
                    if subBundle.actionmap
                        == []:
                        subBundle.actionmap
                            = self.
                            actionmap
                        subBundle.
                            setActionMap()
            else: raise "InvalidBundleType",
                b.type

def setSys(self,sys):
    """Set the ion trap reference of all sub
        -bundles"""
    # This is slow - call it one to
        initialize.
    if self.type == 's': self.sys = sys
    granlist,max = [],0
    if self.type == 'p':
        for i in range(len(self.bundles)):
            t = self.bundles[i].gran()
            granlist.append(t)
            if t > max: max = t
        for i in range(len(self.bundles)):
            if granlist[i] == max:
                self.bundles[i].setSys(sys)
                break
    elif self.type == 's':
        for b in self.bundles: b.setSys(sys)
    elif self.type == 'a':
        # need to set the system of
            condition bundles
        if self.bundles[0][0] == 'condition'
            :
            subBundle = self.bundles
                [0][1][2]
            # this test should prevent
                needless
            # recursion and loops
            if subBundle.sys == None:
                subBundle.sys = sys
                subBundle.setSys(sys)
    else: raise "InvalidBundleType",self.
        type

def start(self,debuglog,t0):
    raise "BaseStartUnimplemented"

def gran(self):
    """Return the 'granularity' of this
        object"""
    """Used to assign messaging priority to
        a thread"""
    if self.type == 'a': return 1
    elif self.type == 's':
        tot = 0
        for b in self.bundles: tot += b.gran
            ()
        return tot
    elif self.type == 'p':
        tot = 0
        for b in self.bundles:
            t = b.gran()
            if t > tot: tot = t
        return tot
    else: raise "InvalidBundleType", self.
        type
```

```
    def attach(self,bundle):
        """Attach a bundle to all of the
            condition actions
        in this bundle"""
        if self.type == 'a':
            # The indices depend on the
                arguments of
            # the 'condition' action
            # action = bundles[0]
            # actiontype = action[0]
            # actionargs = action[1]
            # TheBundle = actionargs[2]
            if self.bundles[0][0] == 'condition'
                :
                if self.bundles[0][1][2] is None
                    :
                    self.bundles[0][1][2] =
                        bundle
        elif self.type == 's':
            for b in self.bundles: b.attach(
                bundle)
        elif self.type == 'p':
            for b in self.bundles: b.attach(
                bundle)
        else:
            raise "InvalidBundleType", self.type

# Implements a bundle that is a single action.
# Single actions are the recursion endpoints
# when start() is called on a bundle.
# a = ["action",params]
class action(bundle):

    def __init__(self,a,actionmap=[]):
        bundle.__init__(self)
        self.type = 'a'
        self.actionmap = actionmap
        self.bundles = [a]

    # communicate the new total time to our
        caller via
    # the variable self.time
    def start(self,debuglog,t0):
        self.time = t0
        a = self.bundles[0]
#           debuglog.put('abundle','->%s t=%f'%(a
    [0],t0),6)
        try:
            actiontime = self.actionmap[a[0]](a
                [1],t0)
            # actiontime is total elapsed time (
                t0 + action's time)
#               debuglog.put('abundle','<-%s t=%f
    '%(a[0],actiontime),6)
            self.time = actiontime
        except "HaltAction",actiontime:
            debuglog.put('abundle','HALT t=%f'%
                actiontime,6)
            self.halted = True
            self.time = actiontime

# Implements a serial bundle. Each bundle in
# a serial bundle is executed after the prior
# bundle in the list finishes.
class sbundle(bundle):

    def __init__(self,bundles,asList=True,
        actionmap=[]):
        # The asList flag is unnecessary - use
            isinstance()
        bundle.__init__(self)
        self.type = 's'
        self.actionmap = actionmap
        if asList:
            alist = []
            for a in bundles:
                alist.append(action(a,self.
                    actionmap))
            self.bundles = alist
        else: self.bundles = bundles

    def start(self,debuglog,t0):
        self.time = t0
#       debuglog.put("sbundle","-> t=%f"%t0,6)
        for b in self.bundles:
            b.start(debuglog,self.time)
            self.time = copy.copy(b.time)
            b.time = 0 # reset the sub-bundle
                time
            if b.halted:
                debuglog.put("sbundle","HALT",6)
                b.halted = False    # reset the
                    sub-bundle flag
                self.halted = True
                break
        if self.sys != None:
            if self.sys.grid != None:
                self.sys.grid.addstatus(self
                    .time,self.sys.
                    lastFidelity)
                # self.sys.grid.addstate(
                    self.sys.qc.statestr())
#           debuglog.put("sbundle","<- t=%f"%self.
    time,6)

# Implements a parallel bundle. Each bundle in
# a parallel bundle executes in a separate
    thread.
# The start() function returns when all of the
# threads finish.
class pbundle(bundle):

    def __init__(self,bundles,asList=False,
        actionmap=[]):
        # The asList flag is unnecessary - use
            isinstance()
        bundle.__init__(self)
        self.type = 'p'
        self.actionmap = actionmap
        if asList:
            alist = []
            for a in bundles:
                alist.append(action([a],self.
                    actionmap))
            self.bundles = alist
        else:
            self.bundles = bundles

    def start(self,debuglog,t0):
        thdlist = []
        self.time = t0
#       debuglog.put("pbundle","-> t=%f"%t0,6)
        for b in self.bundles:
            thd = threading.Thread(target=b.
                start,args=(debuglog,self.time
                ))
            thdlist.append(thd)
        for thd in thdlist:
            thd.start()
        for thd in thdlist:
            thd.join()
        for b in self.bundles:
            if b.halted:
#               debuglog.put("pbundle","HALT t=%
    f"%b.time,6)
                self.halted = True
                b.halted = False   # reset the
                    sub-bundle halt flag
                if self.time < b.time:
                    self.time = copy.copy(b.time
                        )
            if self.time < b.time and not self.
                halted:
                self.time = copy.copy(b.time)
            b.time = 0 # reset the sub-bundle
                time
#       debuglog.put("pbundle","<- t=%f"%self.
    time,6)

if __name__ == "__main__":

    print "bundle.py is not directly
        executable"
```

B.2.3 cell.py

```
# cell.py
# Isaac Chuang <ichuang@mit.edu>
# Andrew Cross <awcross@mit.edu>

from visual import *
from Numeric import *

class cell:

    def __init__(self,xloc,yloc):

        self.type = 'empty'
        self.x = xloc
```

```
        self.y = yloc
        self.frame = frame()
        self.frame.pos = (xloc,yloc,0)        #
            frame position in grid
        self.eflags = array([0,0,0,0])        #
            electrode flags SENW
        self.color = color.blue
        self.laserflag = False                #
            True if visible, False o.w.
        self.detectflag = False
        self.lasercolor = color.green
        self.detectcolor = color.yellow

        # references for the laser and detector
            objects
        self.detector = None
        self.laser1 = None
        self.laser2 = None

        # frame for ions, so we can move them
            around

        self.ionframe = frame()
        self.ionframe.pos = (xloc,yloc,0)         #
            frame position in grid
        self.iondisplacement = vector([0,0,0])
        self.origpos = vector([xloc,yloc,0])
        self.ionobj = None

    def draw(self):

      if self.type=='plaquette': self.
          draw_plaquette()

      if self.type=='ion':
          self.ionobj = sphere(pos=vector
              ([0.5,0.5,0]) + self.
              iondisplacement, radius=0.25,
              frame=self.ionframe,color=self.
              color)

      # We say now that every cell has a laser
          and detector
      if 0:
          if self.type=='empty': return


      self.laser1 = cylinder(frame=self.frame,
          pos=(0.5,0.5,1),radius=0.05,length=8,
          axis=(1,0,0),color=self.lasercolor)
      if not self.laserflag: self.laser1.visible
          = False
      self.laser2 = cylinder(frame=self.frame,
          pos=(0.5,0.5,1),radius=0.05,length
          =-8,axis=(1,0,0),color=self.
          lasercolor)
      if not self.laserflag: self.laser2.visible
          = False
      self.detector = cone(frame=self.frame,pos
          =(0.5,0.5,-0.5),radius=0.4,axis
          =(0,0,-1),color=self.detectcolor)
      if not self.detectflag: self.detector.
          visible = False

    def redraw(self):
        if self.laserflag:
            self.laser1.visible = True
            self.laser2.visible = True
        else:
            self.laser1.visible = False
            self.laser2.visible = False
        if self.detectflag:
            self.detector.visible = True
        else:
            self.detector.visible = False


    # plaquette drawing function
    #
    # All plaquettes are 1x1 and have (0,0) as
        being their lower left corner

    def draw_plaquette(self):

        fp = self.frame
        mygold = (0.6,0.6,0.6)
        mygrey = (0,0.5,0.5)

        def mybox(pos,dx,dy,dz,col,f):
            mb = box(pos=pos+vector(dx/2.0,dy
                /2.0,0), length=dx, height=dy,
                width=dz, color=col,frame=f)
            return(mb)

        edx = 0.5
        edy = 0.2

        if self.eflags[0]==1:   # south
            ebox = mybox(vector((1-edx)/2,0,0.1)
                ,edx,edy,0.1,mygold,fp)
        if self.eflags[1]==1:   # east
            ebox = mybox(vector(1-edy,(1-edx)
                /2,0.1),edy,edx,0.1,mygold,fp)
        if self.eflags[2]==1:   # north
            ebox = mybox(vector((1-edx)/2,1-edy
                ,0.1),edx,edy,0.1,mygold,fp)
        if self.eflags[3]==1:   # west
            ebox = mybox(vector(0,(1-edx)/2,0.1)
                ,edy,edx,0.1,mygold,fp)
        pbox = mybox(vector(0,0,0),1,1,0.1,
            mygrey,fp)    # plaquette body
```

B.2.4 chain.py

```
# chain.py
# Andrew Cross <awcross@mit.edu>
# 6 Oct 2003

import thread
import ion
import copy
import math
from Numeric import *

class chain:
    """Ion chain

    Input is a dictionary of ion objects.
    """
    def __init__(self,ions):
        self.heat = 0          # <N> of
            oscillator mode
        self.ions = ions
        self.lock = thread.allocate_lock()
        self.orientation = 'unoriented'
        if not self.linear(): raise "
            NotLinearChain",str(self)

    def move(self,dx):
        # dx is an ordered pair displacement
            vector
        for key,val in self.ions.items():
            val.displacement = val.displacement
                + array(dx)

    def lockIons(self):
        for key,val in self.ions.items(): val.
            lock.acquire()

    def unlockIons(self):
        for key,val in self.ions.items(): val.
            lock.release()

    def isValidMove(self,dx):
        """Determines if move direction ordered
            pair dx is
        valid given orientation
        """
        if self.orientation == 'unoriented':
            if dx[0] != 0 and dx[1] == 0: return
                True
            if dx[0] == 0 and dx[1] != 0: return
                True
            return False # not vertical or
                horizontal
        elif self.orientation == 'horizontal':
            if dx[0] != 0 and dx[1] == 0: return
                True
            return False
        elif self.orientation == 'vertical':
            if dx[0] == 0 and dx[1] != 0: return
                True
            return False
        else: raise 'InternalError',self.
            orientation

    def __str__(self):
        s = "<"
#       if self.lock.locked(): s = "(L) "
#       else: s = "(U) "
```

```
#           s += "Chain:\n"
        for key,val in self.ions.items():
                s += val.__str__() + "\n"
#           s += "  Orientation: " + str(self.
    orientation)
#           s += "  Heating: " + str(self.heat)
        s += ">\n"
        return s

    def __getitem__(self,ionname):
        return self.ions[ionname]

    def __iter__(self):
       return self.ions.itervalues()

    def __len__(self):
        return len(self.ions)

    def __contains__(self,ionorobject):
        for key,val in self.ions.items():
            if ionorobject == val or ionorobject
                 == key: return True
        return False

    def __add__(self,other):
        """Concatenate two ion chains"""
        # ignore the possibility that two ions
             have the same name
        # or that an ion may be inserted into
             the chain under two
        # different names
        if self.orientation == "horizontal" and
             other.orientation == "vertical":
             raise "ChainTypeMismatch", [self.
             orientation, other.orientation]
        if self.orientation == "vertical" and
             other.orientation == "horizontal":
             raise "ChainTypeMismatch", [self.
             orientation, other.orientation]
        # create a new dictionary so we are
        # only holding a reference to the ions
        newdict = {}
        for key,val in self.ions.items():
             newdict[key] = val
        nc = chain(newdict)
        for key,val in other.ions.items():
                nc.ions[key] = val
        # average the heat
        nc.heat = (other.heat + self.heat)/2
        if not nc.linear(): raise "
             NotLinearChain",str(nc)
        return nc

    def split(self,i1,i2,heating):
        """Splits a chain, returning the split
             off-chain

        i1,i2 ion name strings
        Exceptions: NotAdjacent,InternalError
        """
        if not self.adjacent(i1,i2): raise "
             NotAdjacent"
        if self.orientation == 'horizontal': i
             = 0
        elif self.orientation == 'vertical': i
             = 1
        else: raise "InternalError"
        splitions = {}
        if self.ions[i1].location()[i]<self.ions
             [i2].location()[i]:
            cutpoint = self.ions[i2].location()[
                 i]-0.5
        else:
            cutpoint = self.ions[i1].location()[
                 i]-0.5
        for key,val in self.ions.items():
            if val.location()[i] < cutpoint:
                splitions[key] = val
                del self.ions[key]
        result = chain(splitions)
        if len(self.ions)==1:
            self.orientation = 'unoriented'
        self.heat += heating
        result.heat += heating
        return result

    def linear(self):
        """Test if the chain is a linear chain

        Set the chain _type to horizontal,
             vertical,or unoriented
```

```
        if the chain is linear. Return bool test
             result.
        """
        keys = self.ions.keys()
        if len(keys) <= 1:
            self.orientation = "unoriented"
            return True
        if self.sameRow(keys[0],keys[1]): self.
             orientation = "horizontal"
        else: self.orientation = "vertical"
        for i in range(1,len(keys)):
            if self.orientation == "horizontal":
                if not self.sameRow(keys[0],keys
                     [i]): return False
            else:
                if not self.sameCol(keys[0],keys
                     [i]): return False
        # Test that each ion is adjacent to some
             other ion
        for i in keys:
            found = False
            for j in keys:
                if i != j:
                    if self.adjacent(i,j): found
                         = True
            if not found: return False
        return True

    def validPair(self,i1,i2):
        if i1==i2: raise "SameName",i1
        if not i1 in self: raise "NotInChain",[
             i1,i2]
        if not i2 in self: raise "NotInChain",[
             i1,i2]

    def sameRow(self,i1,i2):
        self.validPair(i1,i2)
        if self.ions[i1].loc[1]+self.ions[i1].
             displacement[1] == \
           self.ions[i2].loc[1]+self.ions[i2].
                displacement[1]: return True
        return False

    def sameCol(self,i1,i2):
        self.validPair(i1,i2)
        if self.ions[i1].loc[0]+self.ions[i1].
             displacement[0] == \
           self.ions[i2].loc[0]+self.ions[i2].
                displacement[0]: return True
        return False

    def adjacent(self,i1,i2):
        self.validPair(i1,i2)
        x = self.ions[i2].loc[0]+self.ions[i2].
             displacement[0]
        y = self.ions[i2].loc[1]+self.ions[i2].
             displacement[1]
        if self.ions[i1].loc+self.ions[i1].
             displacement in \
                [array([x-1,y]),array([x+1,y]),
                    array([x,y-1]),array([x,y
                    +1])]:
            return True
        return False

if __name__ == "__main__":

    # run some consistency checks

    fail = False

    n = 5
    print "CHAIN TEST (", n, "ions )"
    iondicth = {}
    iondictv = {}
    for i in range(n): iondictv["i" + str(i+1)
         ] = \
        ion.ion("q" + str(i+1),"data",array([0,i
             ]))
    for i in range(n): iondicth["i" + str(i+1)
         ] = \
        ion.ion("q" + str(i+1),"data",array([i
             ,0]))
    print "creating horizontal and vertical
         chains ( ions 1 to", n, ")"
    c = chain(iondicth)
    c2 = chain(iondictv)
    print "adjacent tests - ",
    for i in range(n):
        for j in range(n):
            if math.fabs(i-j)==1 and not c.
                 adjacent("i"+str(i+1),"i"+str(j
```

```
                    +1)):
                    fail = True
                if math.fabs(i-j)==1 and not c2.
                    adjacent("i"+str(i+1),"i"+str(j
                    +1)):
                    fail = True
    if fail:
        print "FAILED"
        fail = False
    else: print "passed"
    print "split and merge tests - ",
    for i in range(n-1): # this gives the left
        ion of the split pair
#       print 'presplit c=',c
        cp = c.split("i"+str(i+1),"i"+str(i+2)
            ,100)
        cp2 = c2.split("i"+str(i+1),"i"+str(i+2)
            ,100)
        if len(cp) != i+1 or len(c) != n-i-1:
            fail=True
        if len(cp2) != i+1 or len(c2) != n-i-1:
            fail=True
#       print 'postsplit c=',c,'cp=',cp
        t = c + cp
        t2 = c2 + cp2
#       print 'postadd c=',c,'cp=',cp,'t-',t
        if len(t)!=n or len(cp)!=i+1 or len(c)!=
            n-i-1: fail = True
        if len(t2)!=n or len(cp2)!=i+1 or len(c2
            )!=n-i-1: fail = True
        c = t
        c2 = t2
#       print 'postassign c=',c,'t=',t
    if fail:
        print "FAILED"
        fail = False
    else: print "passed"

    print "movement tests -",
    htest = [[False,(0,1)],[True,(-1,0)],[False
        ,(0,-1)],[True,(1,0)],[False,(1,1)]]
    vtest = [[True,(0,1)],[False,(-1,0)],[True
        ,(0,-1)],[False,(1,0)],[False,(1,1)]]
    utest = [[True,(0,1)],[True,(-1,0)],[True
        ,(0,-1)],[True,(1,0)],[False,(1,1)]]
    cu = chain({"i":ion.ion("i","data",(0,0))})
    for x in utest:
        if cu.isValidMove(x[1]) != x[0]: fail =
            True
        elif cu.isValidMove(x[1]): cu.move(x[1])
    for x in htest:
        if c.isValidMove(x[1]) != x[0]: fail =
            True
        elif c.isValidMove(x[1]): c.move(x[1])
    for x in vtest:
        if c2.isValidMove(x[1]) != x[0]: fail =
            True
        elif c.isValidMove(x[1]): c2.move(x[1])

    for key, val in cu.ions.items():
        if val.displacement != array([0,0]):
            fail = True
    for key, val in c.ions.items():
        if val.displacement != array([0,0]):
            fail = True
    for key, val in c2.ions.items():
        if val.displacement != array([0,0]):
            fail = True

    if fail:
        print "FAILED"
        fail = False
    else: print "passed"
```

B.2.5 chp.c

```
// CHP: Stabilizer Quantum Computer Simulator (
    without scoreboarding)
// by Scott Aaronson
//
// Last modified April 13, 2004
//
// First modified by Andrew Cross and Tzvetan
    Metodiev March 24, 2004
// Pauli gates added March 24, 2004
            (tsmetodiev@ucdavis.edu)
// Controlled-Not corrected March 25, 2004
            (awcross@mit.edu)
// Fidelity function added March 26, 2004
            (awcross@mit.edu)
// Free function added March 31, 2004
            (awcross@mit.edu)
// String output added April 5, 2004
            (awcross@mit.edu)
// Reseed function added April 12, 2004
// Gauss-Jordan elimination added April 13, 2004
            (awcross@mit.edu)
// Comparison added April 13, 2004
            (awcross@mit.edu)
// Copy added April 13, 2004
            (awcross@mit.edu)

#include        <stdio.h>
#include        <stdlib.h>
#include        <string.h>
#include        <time.h>
#include        <math.h>

// include time.h for reseeding the random
    number generator
#include        <sys/time.h>

#include        "chp.h"

#define         CNOT            0
#define         HADAMARD        1
#define         PHASE           2
#define         XGATE           3
#define         ZGATE           4
#define         YGATE           5
#define         MEASURE         6

void reseed(void)

// seed rand() using gettimeofday

{

  struct timeval tv;
  gettimeofday(&tv,NULL);
  srand((unsigned int)tv.tv_sec);

}

void cnot(struct QState *q, long b, long c)

// Apply a CNOT gate with control b and target c

{

    long i;
    long b5;
    long c5;
    unsigned long pwb;
    unsigned long pwc;

    b5 = b>>5;
    c5 = c>>5;
    pwb = q->pw[b&31];
    pwc = q->pw[c&31];
    for (i = 0; i < 2*q->k; i++)
    {
        if (q->x[i][b5]&pwb) q->x[i][c5] ^= pwc;
        if (q->z[i][c5]&pwc) q->z[i][b5] ^= pwb;
        // This next line corrects the phase for
            two cases:
        // YY<-> -XZ and XZ<-> -YY
        // (the left-most qubit is the control).
        // Writing + for xor, the update rule
            for the phase is
        // r_i := r_i + ("YY" or "XZ")
        if ((q->x[i][b5]&pwb) && (q->z[i][c5]&
            pwc) &&
            (q->x[i][c5]&pwc) && (q->z[i][b5]&
                pwb))
            q->r[i] = (q->r[i]+2)%4;
        if ((q->x[i][b5]&pwb) && (q->z[i][c5]&
            pwc) &&
            !(q->x[i][c5]&pwc) && !(q->z[i][b5]&
                pwb))
            q->r[i] = (q->r[i]+2)%4;
    }

    return;

}



void hadamard(struct QState *q, long b)
```

```
// Apply a Hadamard gate to qubit b

{

    long i;
    unsigned long tmp;
    long b5;
    unsigned long pw;

    b5 = b>>5;
    pw = q->pw[b&31];
    for (i = 0; i < 2*q->k; i++)
    {
        tmp = q->x[i][b5];
        q->x[i][b5] ^= (q->x[i][b5] ^ q->z[i][b5
            ]) & pw;
        q->z[i][b5] ^= (q->z[i][b5] ^ tmp) & pw;
        if ((q->x[i][b5]&pw) && (q->z[i][b5]&pw)
            ) q->r[i] = (q->r[i]+2)%4;
    }

    return;

}

void phase(struct QState *q, long b)

// Apply a phase gate (|0>->|0>, |1>->i|1>) to
   qubit b

{

    long i;
    long b5;
    unsigned long pw;

    b5 = b>>5;
    pw = q->pw[b&31];
    for (i = 0; i < 2*q->k; i++)
    {
        if ((q->x[i][b5]&pw) && (q->z[i][b5]&pw)
            ) q->r[i] = (q->r[i]+2)%4;
        q->z[i][b5] ^= q->x[i][b5]&pw;
    }

    return;

}

void xgate(struct QState *q, long b)

// Apply an X gate to qubit b

{
    hadamard(q,b);
    phase(q,b);
    phase(q,b);
    hadamard(q,b);
}

void zgate(struct QState *q, long b)

// Apply a Z gate to qubit b

{
    phase(q,b);
    phase(q,b);
}

void ygate(struct QState *q, long b)

// Apply a Y gate to qubit b

{
    zgate(q,b);
    xgate(q,b);
}

void rowcopy(struct QState *q, long i, long k)

// Sets row i equal to row k

{

    long j;

    for (j = 0; j < q->over32; j++)
```

```
    {
        q->x[i][j] = q->x[k][j];
        q->z[i][j] = q->z[k][j];
    }
    q->r[i] = q->r[k];

    return;

}

void rowswap(struct QState *q, long i, long k)

// Swaps row i and row k

{

    rowcopy(q, 2*q->n, k);
    rowcopy(q, k, i);
    rowcopy(q, i, 2*q->n);

    return;

}

void rowset(struct QState *q, long i, long b)

// Sets row i equal to the bth observable (X_1
   ,...X_n,Z_1,...,Z_n)

{

    long j;
    long b5;
    unsigned long b31;

    for (j = 0; j < q->over32; j++)
    {
        q->x[i][j] = 0;
        q->z[i][j] = 0;
    }
    q->r[i] = 0;
    if (b < q->n)
    {
        b5 = b>>5;
        b31 = b&31;
        q->x[i][b5] = q->pw[b31];
    }
    else
    {
        b5 = (b - q->n)>>5;
        b31 = (b - q->n)&31;
        q->z[i][b5] = q->pw[b31];
    }

    return;

}

int clifford(struct QState *q, long i, long k)

// Return the phase (0,1,2,3) when row i is LEFT
   -multiplied by row k

{

    long j;
    long l;
    unsigned long pw;
    long e=0;    // Power to which i is raised

    for (j = 0; j < q->over32; j++)
        for (l = 0; l < 32; l++)
        {
            pw = q->pw[l];
            if ((q->x[k][j]&pw) && (!(q->z[k][j
                ]&pw)))  // X
            {
                if ((q->x[i][j]&pw) && (q->z[i][
                    j]&pw)) e++;          //
                    XY=iZ
                if ((!(q->x[i][j]&pw)) && (q->z[
                    i][j]&pw)) e--;       //
                    XZ=-iY
            }
            if ((q->x[k][j]&pw) && (q->z[k][j]&
                pw))
```

```
                        // Y
            {
                if ((!(q->x[i][j]&pw)) && (q->z[i][j]&pw)) e++;   // YZ=iX
                if ((q->x[i][j]&pw) && (!(q->z[i][j]&pw))) e--;   // YX=-iZ
            }
            if ((!(q->x[k][j]&pw)) && (q->z[k][j]&pw))
                        // Z
            {
                if ((q->x[i][j]&pw) && (!(q->z[i][j]&pw))) e++;   // ZX=iY
                if ((q->x[i][j]&pw) && (q->z[i][j]&pw)) e--;   // ZY=-iX
            }
        }

    e = (e+q->r[i]+q->r[k])%4;
    if (e>=0) return e;
    else return e+4;

}


void rowmult(struct QState *q, long i, long k)

// Left-multiply row i by row k

{

    long j;

    q->r[i] = clifford(q,i,k);
    for (j = 0; j < q->over32; j++)
    {
        q->x[i][j] ^= q->x[k][j];
        q->z[i][j] ^= q->z[k][j];
    }

    return;

}

void removerowpair(struct QState *q, unsigned long i)

// remove the ith generator and its corresponding
// destabilizer, note that i is >= 0 and < q->k

{
    unsigned long j;

    if( q->k == 1 )
    {
        freestate(q);
        return;
    }

    // shift the ith destabilizer through all the
    // destabilizers and stabilizers, into the 2 k - 1 position
    for( j=i+1; j<2*q->k; j++ ) rowswap(q,j-1,j);

    // shift the k+i-1 generator (the ith generator, now shifted
    // from the prior rowswaps) through all the elements and into
    // the 2*k - 1 position.
    for( j=q->k+i; j<2*q->k; j++ ) rowswap(q,j-1,j);

    // free the scratch space at 2*k and the last
    // destabilizer space at 2*k-1. 2*(k-1) is the new scratch space
    free(q->x[2*q->k]);
    free(q->x[2*q->k-1]);

    // adjust the pointer arrays, etc, to two fewer locations
    // by setting q->k to q->k - 1 and reallocating.
    q->k = q->k - 1;
    q->x = realloc(q->x, (2*q->k+1)*sizeof(unsigned long *));
    q->z = realloc(q->z, (2*q->k+1)*sizeof(unsigned long *));
    q->r = realloc(q->r, (2*q->k+1)*sizeof(int));

}

void addrowpair(struct QState *q)

// put new identity generators on the end of the stabilizer
// and destabilizer list

{
    unsigned long j;

    // grow the tableau by two rows
    q->k = q->k + 1;
    q->x = realloc(q->x, (2*q->k+1)*sizeof(unsigned long *));
    q->z = realloc(q->z, (2*q->k+1)*sizeof(unsigned long *));
    q->r = realloc(q->r, (2*q->k+1)*sizeof(int));
    q->x[2*q->k-1] = malloc(q->over32 * sizeof(unsigned long));
    q->x[2*q->k] = malloc(q->over32 * sizeof(unsigned long));
    q->z[2*q->k-1] = malloc(q->over32 * sizeof(unsigned long));
    q->z[2*q->k] = malloc(q->over32 * sizeof(unsigned long));

    // swap the new 2*k - 1 generator (last, before scratch) up
    // to the q->k-1 position, shifting all the stabilizer
    // generators down one row.
    for( j = 2*q->k - 1; j >= q->k; j-- )
        rowswap(q,j-1,j);

    // initialize the new destabilizer generator in position q->k-1 and
    // the new generator in position 2*q->k-1
    for( j = 0; j < q->over32; j++ )
    {
        q->x[q->k-1][j] = 0;
        q->x[2*q->k-1][j] = 0;
        q->z[q->k-1][j] = 0;
        q->z[2*q->k-1][j] = 0;
    }
    q->r[q->k-1] = 0;
    q->r[2*q->k-1] = 0;
}

void removecol(struct QState *q, unsigned long j)

// remove the jth column from every generator in q
// does not modify the phase
// j >= 0 and j <= q->n

{
    unsigned long FFFF = 0xFFFFFFFF; // 2^32 - 1 = 4294967295
    unsigned long jword = j>>5;   // the word containing the jth bit
    unsigned long leftj = FFFF << ((j&31) + 1);   // left mask
    unsigned long rightj = (FFFF ^ leftj) ^ q->pw[j&31];   // right mask
    unsigned long i, k;

    // a generator with enumerated terms 1 2 3 ... n
    // is organized in memory like this:
    // 32 31 ... 1 | 64 63 ... 33 | ...
    // where the vertical bars separate words.

    // FFFF << 32 == FFFF, so we have to correct that here
    if( (j&31) == 31 )
      {
        leftj = 0;
        rightj = (FFFF ^ leftj) ^ q->pw[j&31];
      }

    // iterate over all generators
    for( k = 0; k < 2*q->k +1; k++ )
    {
```

```
        // start at the word containing the jth
            bit
        for( i = jword; i < q->over32; i++ )
        {
            if( i == jword )
            {
                // if this is the word
                    containing the jth bit
                // right shift the part to the
                    left of the jth bit
                q->x[k][i] = ((q->x[k][i] &
                    leftj)>>1) |
                    (q->x[k][i] & rightj);
                q->z[k][i] = ((q->z[k][i] &
                    leftj)>>1) |
                    (q->z[k][i] & rightj);
            } else {
                // otherwise, just shift the
                    word right one
                // to make up for the missing
                    bit
                q->x[k][i] = q->x[k][i] >> 1;
                q->z[k][i] = q->z[k][i] >> 1;
            }
            // if there are still more words to
                the right
            // take the right-most bit of the
                next word
            // and place it at the left-most bit
                of this word
            if( i < q->over32 - 1 )
            {
                q->x[k][i] |= ((q->x[k][i
                    +1] & 1) << 31);
                q->z[k][i] |= ((q->z[k][i
                    +1] & 1) << 31);
            }
        }
    }
    // release some memory?
    q->n = q->n - 1;
    if( ((q->n>>5) + 1) < q->over32 )
    {
        // we need fewer words to represent each
            row
        q->over32 = (q->n>>5) + 1;
        for( i=0; i< 2*q->k + 1; i++ )
        {
            q->x[i] = realloc(q->x[i], q->over32
                * sizeof(unsigned long));
            q->z[i] = realloc(q->z[i], q->over32
                * sizeof(unsigned long));
        }
    }
}

void addcol(struct QState *q)

// add a column to the end of every generator in
    q.
// does not modify the phase.

{

    unsigned long i;

    q->n = q->n + 1;

    // should another word be allocated?
    if( (q->n>>5) + 1 > q->over32 )
    {
        q->over32++;
        for( i=0; i < 2*q->k + 1; i++ )
        {
            q->x[i] = realloc(q->x[i], q->over32
                * sizeof(unsigned long));
            q->z[i] = realloc(q->z[i], q->over32
                * sizeof(unsigned long));
            q->x[i][q->over32-1] = 0;
            q->z[i][q->over32-1] = 0;
        }
    }

}

void swapcol(struct QState *q, long i, long j)

// swap columns i and j (qubit relabel)

{
        long k;
        long i5 = i >> 5;
        long j5 = j >> 5;
        unsigned long pwi = q->pw[i&31];
        unsigned long pwj = q->pw[j&31];
        unsigned long xi, zi;

        for( k = 0; k < 2*q->k; k++ )
        {
                xi = q->x[k][i5]&pwi;    // copy
                    column i
                zi = q->z[k][i5]&pwi;
                q->x[k][i5] |= pwi;       // set
                    column i
                q->z[k][i5] |= pwi;
                // flip column i if column j not
                    set
                if( !(q->x[k][j5]&pwj) ) q->x[k
                    ][i5] ^= pwi;
                if( !(q->z[k][j5]&pwj) ) q->z[k
                    ][i5] ^= pwi;
                q->x[k][j5] |= pwj;       // set
                    column j
                q->z[k][j5] |= pwj;
                // flip column j if column i not
                    set
                if( !xi ) q->x[k][j5] ^= pwj;
                if( !zi ) q->z[k][j5] ^= pwj;
        }

}

char *statestring(struct QState *q)

// Return a string containing the stabilizer and
    destabilizer
// for state q

{
    long i;
    long j;
    long j5;
    unsigned long pw;

    char *result;
    int size;

    size = (q->n+2) * (2*q->k + 1) + 3; // 2
        extra bytes i think
    result = malloc( size * sizeof(char));
    strcpy(result,"");

    for (i = 0; i < 2*q->k; i++)
    {
        if (i == q->k)
        {
            strcat(result, "\n");
            for (j = 0; j < q->n+1; j++)
                strcat(result,"-");
        }
        if ( i > 0 )
        {
            if (q->r[i]==2) strcat(result,"\n-")
                ;
            else strcat(result,"\n+");
        }
        else
        {
            if (q->r[i]==2) strcat(result,"-");
            else strcat(result,"+");
        }
        for (j = 0; j < q->n; j++)
        {
            j5 = j>>5;
            pw = q->pw[j&31];
            if ((!(q->x[i][j5]&pw)) && (!(q->z[i
                ][j5]&pw)))               strcat(
                result,"I");
            if ((q->x[i][j5]&pw) && (!(q->z[i][
                j5]&pw)))                 strcat
                (result,"X");
            if ((q->x[i][j5]&pw) && (q->z[i][j5
                ]&pw))                    strcat
                (result,"Y");
            if ((!(q->x[i][j5]&pw)) && (q->z[i][
                j5]&pw))                  strcat(
                result,"Z");
        }
    }

    return result;
}

void printstate(struct QState *q)
```

```
// Print the destabilizer and stabilizer for
    state q

{

    long i;
    long j;
    long j5;
    unsigned long pw;

    for (i = 0; i < 2*q->k; i++)
    {
        if (i == q->k)
        {
            printf("\n");
            for (j = 0; j < q->n+1; j++)
                printf("-");
        }
        if (q->r[i]==2) printf("\n-");
        else printf("\n+");
        for (j = 0; j < q->n; j++)
        {
            j5 = j>>5;
            pw = q->pw[j&31];
            if ((!(q->x[i][j5]&pw)) && (!(q->z[i
                ][j5]&pw)))                printf(
                "I");
            if ((q->x[i][j5]&pw) && (!(q->z[i][
                j5]&pw)))                  printf
                ("X");
            if ((q->x[i][j5]&pw) && (q->z[i][j5
                ]&pw))                     printf
                ("Y");
            if ((!(q->x[i][j5]&pw)) && (q->z[i][
                j5]&pw))                   printf(
                "Z");
        }
    }
    printf("\n");

    return;

}



int measure(struct QState *q, long b, int sup)

// Measure qubit b
// Return          0 if outcome would always be 0
//                 1 if outcome would always be 1
//                 2 if outcome was random and 0
    was chosen
//                 3 if outcome was random and 1
    was chosen
// sup: 1 if determinate measurement results
    should be
//      suppressed, 0 otherwise

{

    int ran = 0;
    long i;
    long p;        // pivot row in stabilizer
    long m;        // pivot row in destabilizer
    long b5;
    unsigned long pw;

    b5 = b>>5;
    pw = q->pw[b&31];
    for (p = 0; p < q->k; p++)             // loop
        over stabilizer generators
    {
        if (q->x[p+q->k][b5]&pw) ran = 1;
                    // if a Zbar does NOT
            commute with Z_b (the
        if (ran) break;
                                        // operator
            being measured), then outcome is
            random
    }

    // If outcome is indeterminate
    if (ran)
    {
        rowcopy(q, p, p + q->k);
                            // Set Xbar_p :=
            Zbar_p
        rowset(q, p + q->k, b + q->k);
                    // Set Zbar_p := Z_b
        q->r[p + q->k] = 2*(rand()%2);
                    // moment of quantum
            randomness
        for (i = 0; i < 2*q->k; i++)
                    // Now update the Xbar's
            and Zbar's that don't commute with
            if ((i!=p) && (q->x[i][b5]&pw))
                            // Z_b
                rowmult(q, i, p);
        if (q->r[p + q->k]) return 3;
        else return 2;
    }

    // If outcome is determinate
    if ((!ran) && (!sup))
    {
        for (m = 0; m < q->k; m++)
                            // Before we were
            checking if stabilizer generators
            commute
            if (q->x[m][b5]&pw) break;
                        // with Z_b; now we're
                checking destabilizer
                generators
        rowcopy(q, 2*q->k, m + q->k);
        for (i = m+1; i < q->k; i++)
            if (q->x[i][b5]&pw)
                rowmult(q, 2*q->k, i + q->k);
        if (q->r[2*q->k]) return 1;
        else return 0;
        /*for (i = m+1; i < q->n; i++)
          if (q->x[i][b5]&pw)
          {
          rowmult(q, m + q->n, i + q->n);
          rowmult(q, i, m);
          }
          return (int)q->r[m + q->n];*/
    }

    return 0;

}


long gaussjordan(struct QState *q)

// Gauss-Jordan elimination to put the
    stabilizer generators into
// a unique form.
// (Return value = number of such generators =
    log_2 of number of nonzero basis states)

{
    long j, k;
    long l, l5;
    unsigned long pw;
    long row, g;

    g = gaussian(q);

    // jordan reduction

    // find the first row from the bottom with
        an X
    // (if it doesn't exist then row = q->n - 1)
    for( k = 2*q->k - 1; k > q->k - 1; k-- )
      {
        for( l = 0; l < q->n; l++ )
          {
            l5 = l >> 5;
            pw = q->pw[l&31];
            if( q->x[k][l5]&pw ) break;
          }
        if( l < q->n ) break;
      }

    // printf("first search for X: row = %ld,
        col = %ld\n",k,l);
    row = k;

    // for all of the Z rows
    for( k = 2*q->k-1; k > row; k-- )
      {
        // find the first Z from the left
        // (one should always exist)
        for( l = 0; l < q->n; l++ )
          {
            l5 = l >> 5;
            pw = q->pw[l&31];
            if( q->z[k][l5]&pw ) break;
          }
```

```
        // printf("found z at row = %ld col = %
             ld\n",k,l);

        // for all generators above with a Z
        // in the l position, reduce
        for( j = k - 1; j > q->k - 1; j-- )
          {
            if( q->z[j][l5]&pw )
              {
                // printf("z reducing row = %ld\
                     n",j);
                rowmult(q,j,k);
                rowmult(q,j-q->n,k-q->n);
              }
          }
      }

    // for all of the X rows
    for( k = row; k > q->k - 1; k-- )
      {
        // find the first X from the left
        // (one should always exist)
        for( l = 0; l < q->n; l++ )
          {
            l5 = l >> 5;
            pw = q->pw[l&31];
            if( q->x[k][l5]&pw ) break;
          }

        // printf("found x at row = %ld col = %
             ld\n",k,l);

        // for all generators above with an X
        // in the l position, reduce
        for( j = k - 1; j > q->k - 1; j-- )
          {
            if( q->x[j][l5]&pw )
              {
                // printf("z reducing row = %ld\
                     n",j);
                rowmult(q,j,k);
                rowmult(q,j-q->n,k-q->n);
              }
          }
      }

    return g;
}


long gaussian(struct QState *q)

// Do Gaussian elimination to put the stabilizer
      generators in the following form:
// At the top, a minimal set of generators
      containing X's and Y's, in "quasi-upper-
      triangular" form.
// (Return value = number of such generators =
      log_2 of number of nonzero basis states)
// At the bottom, generators containing Z's only
      in quasi-upper-triangular form.

{

    long i = q->k;
    long k;
    long k2;
    long j;
    long j5;
    long g;       // Return value
    unsigned long pw;

    for (j = 0; j < q->n; j++)
    {
        j5 = j>>5;
        pw = q->pw[j&31];
        for (k = i; k < 2*q->k; k++)    // Find
             a generator containing X in jth
             column
             if (q->x[k][j5]&pw) break;
        if (k < 2*q->k)
        {
             rowswap(q, i, k);
             rowswap(q, i-q->k, k-q->k);
             for (k2 = i+1; k2 < 2*q->k; k2++)
                  if (q->x[k2][j5]&pw)
                  {
                       rowmult(q, k2, i);
                             // Gaussian elimination
                             step
                       rowmult(q, i-q->k, k2-q->k);
                  }
                  i++;
        }
    }
    g = i - q->k;

    for (j = 0; j < q->n; j++)
    {
        j5 = j>>5;
        pw = q->pw[j&31];
        for (k = i; k < 2*q->k; k++)    // Find
             a generator containing Z in jth
             column
             if (q->z[k][j5]&pw) break;
        if (k < 2*q->k)
        {
             rowswap(q, i, k);
             rowswap(q, i-q->k, k-q->k);
             for (k2 = i+1; k2 < 2*q->k; k2++)
                  if (q->z[k2][j5]&pw)
                  {
                       rowmult(q, k2, i);
                       rowmult(q, i-q->k, k2-q->k);
                  }
             i++;
        }
    }

    return g;

}

void freestate(struct QState *q)

// free the memory associated with q

{

    long i;

    for( i = 0; i < 2*q->k + 1; i++ )
    {
        free( q->x[i] );
        free( q->z[i] );
    }
    free(q->r);
    free(q->z);
    free(q->x);
    free(q);

}

void addqubit(struct QState *q)

// append a new qubit to q in state z+

{
    addcol(q);
    addrowpair(q);
    // initialize the new generators
    q->x[q->k-1][(q->n-1)>>5] = q->pw[(q->n-1)
         &31];
    q->z[2*q->k-1][(q->n-1)>>5] = q->pw[(q->n-1)
         &31];
    q->r[q->k-1] = 0;
    q->r[2*q->k-1] = 0;
}

void removequbit(struct QState *q, unsigned long
      b)

// remove the bth qubit of q

{

    long j, k;
    long b5 = b>>5;
    unsigned long pw = q->pw[b&31];

    measure(q,b,1); // measure the bth qubit,
         neglecting outcome

    // find the first stabilizer generator with
         a Z in the bth term.
    // guaranteed to find one, because we
         measured Z_b.
    for( j = q->k; j < 2*q->k; j++ )
            if( q->z[j][b5]&pw ) break;

    // multiply all generators with weight in
         the bth term
    // by the generator we just found.
```

```
    for( k = q->k; k< 2*q->k; k++ )
        if( q->z[k][b5]&pw && k != j )
            rowmult(q,k,j);

    removerowpair(q,j-q->k);
    removecol(q,b);

    // rebuild the destabilizer generators
    // mkdestabilizers(q);
}

long invalid(struct QState *q)
// check the consistency of q
// returns 0 if good, otherwise calls exit()
{
  long j,k,l;

  // no identity elements
  for( j = 0; j < 2*q->k; j++ )
    {
      for( l = 0; l < q->over32; l++ )
        if( q->x[j][l] != 0 ||
            q->z[j][l] != 0 ) break;
      if( l == q->over32 )
        {
          printf("ASSERT: invalid QState -
              identity generator at row %ld\n"
              , j);
          exit(1);
        }
    }

  // no duplicate rows
  for( j = 0; j < 2*q->k; j++ )
    {
      for( k = 0; k < 2*q->k; k++ )
        {
          if( k != j )
            {
              for( l = 0; l < q->over32; l++ )
                if( q->x[j][l] != q->x[k][l] ||
                    q->z[j][l] != q->z[k][l] )
                  break;
              if( l == q->over32 )
                {
                  printf("ASSERT: invalid QState
                      - duplicate generators
                      at rows %ld, %ld\n", j, k
                      );
                  exit(1);
                }
            }
        }
    }

}

long commutes(struct QState *q, long i, long j)
// test if row i and row j commute
// return 1 if yes, 0 if no
// UNTESTED
{
        long l;
        unsigned long tip=0L;
        // commutes if "twisted" inner product
            is zero
        for( l = 0; l < q->over32; l++ )
                tip += weight(q->x[i][l] & q->z[
                    j][l])
                    + weight(q->z[i][l] & q->x[
                        j][l]);
        return ((tip%2)^1);
}

void mkdestabilizers(struct QState *q)
// create appropriate destabilizer generators
    for the
// stabilizer generators in q
// There is the possibility that this algorithm
    doesn't
// account for a situation like
// XX
// ZZ
// I think this algorithm will choose XI, ZI,
    but these don't commute.
// The correction choice is XI,IZ. Check this
    later.
{
    long j, k;
    long l, l5;
    unsigned long pw;
    long row;

    gaussjordan(q);

    // reinitialize the destabilizers to
        identity
    for( k = 0; k < q->k; k++ )
      {
        for( l=0; l < q->over32; l++ )
          {
            q->x[k][l] = 0;
            q->z[k][l] = 0;
          }
        q->r[k] = 0;
      }

    // find the first row from the bottom with
        an X
    // (if it doesn't exist then row = q->n - 1)
    for( k = 2*q->k - 1; k > q->k - 1; k-- )
      {
        for( l = 0; l < q->n; l++ )
          {
            l5 = l >> 5;
            pw = q->pw[l&31];
            if( q->x[k][l5]&pw ) break;
          }
        if( l < q->n ) break;
      }

    // printf("first search for X: row = %ld,
        col = %ld\n",k,l);
    row = k;

    // for all of the Z rows
    for( k = 2*q->k-1; k > row; k-- )
      {
        // find the first Z from the left
        // (one should always exist)
        for( l = 0; l < q->n; l++ )
          {
            l5 = l >> 5;
            pw = q->pw[l&31];
            if( q->z[k][l5]&pw ) break;
          }

        // printf("found z at row = %ld col = %
            ld\n",k,l);
        // set a destabilizer generator
            appropriately
        q->x[k-q->k][l>>5] |= q->pw[l&31];
      }

    // for all of the X rows
    for( k = row; k > q->k - 1; k-- )
      {
        // find the first X from the left
        // (one should always exist)
        for( l = 0; l < q->n; l++ )
          {
            l5 = l >> 5;
            pw = q->pw[l&31];
            if( q->x[k][l5]&pw ) break;
          }

        // printf("found x at row = %ld col = %
            ld\n",k,l);
        // set a destabilizer generator
            appropriately
        q->z[k-q->k][l>>5] |= q->pw[l&31];
      }

}

char *basisstatestring(struct QState *q)

// return a string containing the basis state
    corresponding to
// applying the scratch space generator to the
    zero ket.

{
    long j;
    long j5;
    unsigned long pw;
    int e = q->r[2*q->k];

    char *result;
    int size;

    size = 8 + q->n; // 2 extra bytes i think
    result = malloc( size * sizeof(char));
    strcpy(result,"");
```

```
    for (j = 0; j < q->n; j++)
    {
        j5 = j>>5;
        pw = q->pw[j&31];
        if ((q->x[2*q->k][j5]&pw) && (q->z[2*q->
            k][j5]&pw))              // Pauli
            operator is "Y"
            e = (e+1)%4;
    }
    if (e==0) strcat(result," +|");
    if (e==1) strcat(result,"+i|");
    if (e==2) strcat(result," -|");
    if (e==3) strcat(result,"-i|");

    for (j = 0; j < q->n; j++)
    {
        j5 = j>>5;
        pw = q->pw[j&31];
        if (q->x[2*q->k][j5]&pw) strcat(result,"
            1");
        else strcat(result,"0");
    }
    strcat(result,">");

    return result;
}

void printbasisstate(struct QState *q)

// Prints the result of applying the Pauli
    operator in the "scratch space" of q to
    |0...0>

{

    long j;
    long j5;
    unsigned long pw;
    int e = q->r[2*q->k];

    for (j = 0; j < q->n; j++)
    {
        j5 = j>>5;
        pw = q->pw[j&31];
        if ((q->x[2*q->k][j5]&pw) && (q->z[2*q->
            k][j5]&pw))              // Pauli
            operator is "Y"
            e = (e+1)%4;
    }
    if (e==0) printf("\n +|");
    if (e==1) printf("\n+i|");
    if (e==2) printf("\n -|");
    if (e==3) printf("\n-i|");

    for (j = 0; j < q->n; j++)
    {
        j5 = j>>5;
        pw = q->pw[j&31];
        if (q->x[2*q->k][j5]&pw) printf("1");
        else printf("0");
    }
    printf(">");

    return;

}


void seed(struct QState *q, long g)

// Finds a Pauli operator P such that the basis
    state P|0...0> occurs with nonzero
    amplitude in q, and
// writes P to the scratch space of q.  For this
     to work, Gaussian elimination must already
     have been
// performed on q.  g is the return value from
    gaussian(q).

{

    long i;
    long j;
    long j5;
    unsigned long pw;
    int f;
    long min;

    // the last row of q is scratch space
    q->r[2*q->k] = 0;
    for (j = 0; j < q->over32; j++)
    {
        q->x[2*q->k][j] = 0;              // Wipe
            the scratch space clean
        q->z[2*q->k][j] = 0;
    }
    for (i = 2*q->k - 1; i >= q->k + g; i--)
    {
        f = q->r[i];
        for (j = q->n - 1; j >= 0; j--)
        {
            j5 = j>>5;
            pw = q->pw[j&31];
            if (q->z[i][j5]&pw)
            {
                min = j;
                if (q->x[2*q->k][j5]&pw) f = (f
                    +2)%4;
            }
        }
        if (f==2)
        {
            j5 = min>>5;
            pw = q->pw[min&31];
            q->x[2*q->k][j5] ^= pw;
                // Make the seed consistent
                with the ith equation
        }
    }

    return;

}

int weight(unsigned long w)

// calculate the hamming weight of w

{
        int i=0;
        unsigned long j=1;
        do
        {
                if( w & j ) i++;
                j *= 2;
        } while( j != 0x80000000 );
        if( w & j ) i++;
        return i;
}

int same(struct QState *q1, struct QState *q2)

// Compare the reduced generators of q1 and q2,
// return 1 if same, 0 if different
// O(n^3)

{

    long j, k;

    if( q1->n != q2->n || q1->k != q2->k )
        return 0;

    gaussjordan(q1);
    gaussjordan(q2);

    // only compare the stabilizer generators
    for( j = q1->k; j < 2*q1->k; j++ )
    {
        // compare each word, assuming the words
        // are zero padded
        for( k = 0; k < q1->over32; k++ )
        {
            if( q1->x[j][k] != q2->x[j][k] )
                return 0;
            if( q1->z[j][k] != q2->z[j][k] )
                return 0;
        }
        if( q1->r[j] != q2->r[j] ) return 0;
    }

    return 1;

}

long scratchprod(struct QState *q1, struct
    QState *q2)

// calculates the inner product <0|HG|0> where G
    is the
// scratch space of q1 and H is the scratch
    space
```

```
// of q2. This is either 0, 1, i, -1, or -i, so
     we
// will return -1, 0, 1, 2, 3, respectively.
// Please make sure q1->n == q2->n.

{
    long j, j5;
    unsigned long pw1, pw2;
    int e1 = q1->r[2*q1->k];
    int e2 = q2->r[2*q2->k];

    for (j = 0; j < q1->n; j++)
    {
        // see printbasisstate
        j5 = j>>5;
        pw1 = q1->pw[j&31];
        pw2 = q2->pw[j&31];
        if ((q1->x[2*q1->k][j5]&pw1) && (q1->z
            [2*q1->k][j5]&pw1))
            e1 = (e1+1)%4;
        if ((q2->x[2*q2->k][j5]&pw2) && (q2->z
            [2*q2->k][j5]&pw2))
            e2 = (e2+1)%4;
    }
    // e2 is conjugated
    if( e2 == 1 || e2 == 3 ) e2 = (e2 + 2)%4;
    // are the kets equal?
    for (j = 0; j < q1->n; j++)
    {
        j5 = j>>5;
        pw1 = q1->pw[j&31];
        pw2 = q2->pw[j&31];
        // see printbasisstate
        if ((q1->x[2*q1->k][j5]&pw1) !=
            (q2->x[2*q2->k][j5]&pw2))
        {
            return -1;
        }
    }
    // yes, so return the product of the phases
    return (e1+e2);
}

long overlap(struct QState *q1, struct QState *
    q2)

// Returns -1 if <q2|G|0> == 0 for G in the
     scratch space
// of q1. Returns 0 if <q2|G|0> == 1/sqrt(g1*g2)
    ,
// 1 if <q2|G|0> == i/sqrt(g1*g2), 2 if <q2|G
    |0> == -1/sqrt(g1*g2),
// and 3 if <q2|G|0> == -i/sqrt(g1*g2).
// Please make sure q1->n == q2->n.

{
    long g, i, j, p;
    unsigned long t, t2;

    // see printket
    g = gaussian(q2);
    seed(q2, g);
    p = scratchprod(q1,q2);
    if( p > -1 ) return p;
    for (t = 0; t < q2->pw[g]-1; t++)
    {
        t2 = t ^ (t+1);
        for (i = 0; i < g; i++)
            if (t2 & q2->pw[i])
                rowmult(q2, 2*q2->k, q2->k + i);
        p = scratchprod(q1,q2);
        if( p > -1 ) return p;
    }
    return -1;
}

double fidelity(struct QState *q1, struct QState
    *q2)

// Returns a double that equals sqrt(|<q1|q2
    >|^2).
// Please make sure q1->n == q2->n.
// O(a^n) at worst, YUCK

{
    // Loop over 2^(g1+g2) states; g1, g2 are
       the
    // ranks of q1 and q2.
    long g, g2, i;
    unsigned long t, t2;
    long p=0;
    long real = 0;
    long imag = 0;

    // see printket
    g = gaussian(q1);
    g2 = gaussian(q2);
    g2 += g;    // g1 + g2 gives the denominator
    seed(q1, g);
    p = overlap(q1,q2);
    // calculate the sum
    if( p == 0 ) real += 1;
    if( p == 1 ) imag += 1;
    if( p == 2 ) real -= 1;
    if( p == 3 ) imag -= 1;
    for (t = 0; t < q1->pw[g]-1; t++)
    {
        t2 = t ^ (t+1);
        for (i = 0; i < g; i++)
            if (t2 & q1->pw[i])
                rowmult(q1, 2*q1->k, q1->k + i);
        p = overlap(q1,q2);
        // calculate the sum
        if( p == 0 ) real += 1;
        if( p == 1 ) imag += 1;
        if( p == 2 ) real -= 1;
        if( p == 3 ) imag -= 1;
    }

    return sqrt(((double)(real*real+imag*imag))/
        pow(2,g2));

}

char *ketstring(struct QState *q)

// returns a string containing the ket
     representation of q

{
    long g;              // log_2 of number of
         nonzero basis states
    unsigned long t;
    unsigned long t2;
    long i;

    char *result;
    int size;

    char *temp;

    g = gaussian(q);
    // printf("\n2^%ld nonzero basis states", g)
        ;
    if (g > 31)
    {
        result = malloc(100*sizeof(char));
        strcpy(result,"State is WAY too big to
            print");
        return result;
    }
    size = pow(2,g) * (8 + q->n) + 4; // 2 extra
        bytes i think
    result = malloc(size * sizeof(char));
    seed(q, g);
    temp = basisstatestring(q);
    strcpy(result,temp);
    free(temp);
    for (t = 0; t < q->pw[g]-1; t++)
    {
        t2 = t ^ (t+1);
        for (i = 0; i < g; i++)
            if (t2 & q->pw[i])
                rowmult(q, 2*q->k, q->k + i);
        temp = basisstatestring(q);
        strcat(result,temp);
        free(temp);
    }

    return result;
}

void printket(struct QState *q)

// Print the state in ket notation (warning:
     could be huge!)

{

    long g;              // log_2 of number of
         nonzero basis states
    unsigned long t;
    unsigned long t2;
```

```
    long i;

    g = gaussian(q);
    printf("\n2^%ld nonzero basis states", g);
    if (g > 31)
    {
        printf("\nState is WAY too big to print"
            );
        return;
    }

    seed(q, g);
    printbasisstate(q);
    for (t = 0; t < q->pw[g]-1; t++)
    {
        t2 = t ^ (t+1);
        for (i = 0; i < g; i++)
            if (t2 & q->pw[i])
                rowmult(q, 2*q->k, q->k + i);
        printbasisstate(q);
    }
    printf("\n");

    return;

}


void preparestate(struct QState *q, char *s)

// Prepare the initial state's "input"

{

    long l;
    long b;

    l = strlen(s);
    for (b = 0; b < l; b++)
    {
        if (s[b]=='Z')
        {
            // flip this bit
            hadamard(q,b);
            phase(q,b);
            phase(q,b);
            hadamard(q,b);
        }
        // rotate this bit to 0+1
        if (s[b]=='x') hadamard(q,b);
        if (s[b]=='X')
        {
            // rotate this bit to 0-1
            hadamard(q,b);
            phase(q,b);
            phase(q,b);
        }
        if (s[b]=='y')
        {
            // rotate this bit to 0+i1
            hadamard(q,b);
            phase(q,b);
        }
        if (s[b]=='Y')
        {
            // rotate this bit to 0-i1
            hadamard(q,b);
            phase(q,b);
            phase(q,b);
            phase(q,b);
        }
    }

    return;

}

struct QState *copy( struct QState *q )

// Copy the quantum state q

{
        struct QState *p;
        long i,j;

        p = malloc(sizeof(struct QState));

        p->n = q->n;
        p->k = q->k;
        p->x = malloc((2*p->k + 1) * sizeof(
            unsigned long *));
        p->z = malloc((2*p->k + 1) * sizeof(
            unsigned long *));
        p->r = malloc((2*p->k + 1) * sizeof(int)
            );
        p->over32 = (p->n>>5) + 1;
        p->pw[0] = 1;
        for( i = 1; i < 32; i++ )
            p->pw[i] = 2*p->pw[i-1];
        for( i = 0; i < 2*p->n + 1; i++ )
        {
                p->x[i] = malloc(p->over32 *
                    sizeof(unsigned long));
                p->z[i] = malloc(p->over32 *
                    sizeof(unsigned long));
                for( j = 0; j < p->over32; j++ )
                {
                        p->x[i][j] = q->x[i][j];
                        p->z[i][j] = q->z[i][j];
                }
                p->r[i] = q->r[i];
        }

        return p;
}

struct QState *initialize(long n, char *s)

// Initialize state q to have n qubits, and
    input specified by s

{

    long i;
    long j;

    struct QState *q;

    q = malloc(sizeof(struct QState));

    q->n = n;
    q->k = n;
    q->x = malloc((2*q->k + 1) * sizeof(unsigned
        long*));
    q->z = malloc((2*q->k + 1) * sizeof(unsigned
        long*));
    q->r = malloc((2*q->k + 1) * sizeof(int));
    q->over32 = (q->n>>5) + 1;
    q->pw[0] = 1;
    for (i = 1; i < 32; i++)
        q->pw[i] = 2*q->pw[i-1];
    for (i = 0; i < 2*q->n + 1; i++)
    {
        q->x[i] = malloc(q->over32 * sizeof(
            unsigned long));
        q->z[i] = malloc(q->over32 * sizeof(
            unsigned long));
        for (j = 0; j < q->over32; j++)
        {
            q->x[i][j] = 0;
            q->z[i][j] = 0;
        }
        if (i < q->n)
            q->x[i][i>>5] = q->pw[i&31];
        else if (i < 2*q->n)
        {
            j = i-q->n;
            q->z[i][j>>5] = q->pw[j&31];
        }
        q->r[i] = 0;
    }
    if (s) preparestate(q, s);

    return q;

}
```

B.2.6 chp.h

```
/* File: chp.i */
#ifndef __CHP__
#define __CHP__

struct QState

// Quantum state
```

```
{
        // To save memory and increase speed,
             the bits are packed 32 to an
             unsigned long
        long n; // # of qubits
        long k; // # of generators
        unsigned long **x;        // (2k+1)*n
             matrix for stabilizer/destabilizer
             x bits
                                  // plus one "
                                       scratch row
                                       " at the
                                       bottom
        unsigned long **z;        // (2k+1)*n
             matrix for z bits
        int *r;                   // Phase bits
             : 0 for +1, 1 for i, 2 for -1, 3
             for -i.
        unsigned long pw[32];     // pw[i] = 2^i
        long over32;              // floor(n/8)
             + 1
};

// Reseed the random number generator using the
     system clock
void reseed(void);

// Apply a CNOT gate with control b and target c
void cnot(struct QState *q, long b, long c);

// Apply a Hadamard gate to qubit b
void hadamard(struct QState *q, long b);

// Apply a phase gate to qubit b
void phase(struct QState *q, long b);

// Apply an X gate to qubit b
void xgate(struct QState *q, long b);

// Apply a Z gate to qubit b
void zgate(struct QState *q, long b);

// Apply a Y gate to qubit b
void ygate(struct QState *q, long b);

// Set row i equal to row k
void rowcopy(struct QState *q, long i, long k);

// Swap row i and row k
void rowswap(struct QState *q, long i, long k);

// Set row i equal to the bth observable (X_1
     ,..., X_n, Z_1,..., Z_n)
void rowset(struct QState *q, long i, long b);

// Return the phase (0,1,2,3) when row i is LEFT
     -multiplied by row k
int clifford(struct QState *q, long i, long k);

// Left-multiply row i by row k
void rowmult(struct QState *q, long i, long k);

// Remove the ith generator and its
     corresponding destabilizer
// (i.e. remove rows i and i + q->k)
void removerowpair(struct QState *q, unsigned
     long i);

// Add identity rows to the end of the
     stabilizer and destabilizer
void addrowpair(struct QState *q);

// Remove column j from every generator without
     modifying phase
void removecol(struct QState *q, unsigned long j
     );

// Add a column to the end of every generator
     without modifying phase
void addcol(struct QState *q);

// Swap columns i and j
void swapcol(struct QState *q, long i, long j);

// Create appropriate destabilizer generators
void mkdestabilizers(struct QState *q);

// Return the string containing the stabilizer
     and destabilizer
char *statestring(struct QState *q);

// Print the stabilizer and destabilizer
void printstate(struct QState *q);

// Measure qubit b
int measure(struct QState *q, long b, int sup);

// Gauss-Jordan elimination
long gaussjordan(struct QState *q);

// Gaussian elimination
long gaussian(struct QState *q);

// Free the memory associated with q
void freestate(struct QState *q);

// Append a new qubit in state |0>
void addqubit(struct QState *q);

// Remove the bth qubit of q
void removequbit(struct QState *q, unsigned long
     b);

// Test if q is valid
long invalid( struct QState *q );

// Commutes?
long commutes( struct QState *q, long i, long j
     );

// Return a string containing the basis state
     corresponding to
// applying the scratch space generator to the
     zero ket.
char *basisstatestring(struct QState *q);

// Print the result of applying the scratch
     space generator
// to the zero ket.
void printbasisstate(struct QState *q);

// Find a pauli operator p such that p|0> occurs
     with nonzero
// amplitude in q and writes p to the scratch
     space of q.
void seed(struct QState *q, long g);

// Hamming weight of w
int weight(unsigned long w);

// Compare two stabilizers, returning 1 if same
     , 0 otherwise
int same(struct QState *q1, struct QState *q2);

// Calculate inner product <0|HG|0> where G is
     the scratch space
// of q1 and H is the scratch space of q2.
long scratchprod(struct QState *q1, struct
     QState *q2);

// Calculate the partial overlap of q1 and q2
long overlap(struct QState *q1, struct QState *
     q2);

// Calculates the fidelity
double fidelity(struct QState *q1, struct QState
     *q2);

// Returns the ket representation
char *ketstring(struct QState *q);

// Prints the ket representation
void printket(struct QState *q);

// Prepare an initial state
void preparestate(struct QState *q, char *s);

// Copy a state
struct QState *copy(struct QState *q);

// Create an initial state
struct QState *initialize(long n, char *s);

#endif // __CHP__
```

B.2.7 chp.i

```
/* File: chp.i */
%module chp
%include typemaps.i
%{
struct QState
{
        long n;
        long k;
        unsigned long **x;
        unsigned long **z;
        int *r;
        unsigned long pw[32];
        long over32;
};
%}

extern void reseed(void);

extern struct QState *initialize(long n, char *s
    );
extern struct QState *copy(struct QState *q);
extern void freestate(struct QState *q);
extern void removequbit(struct QState *q,
    unsigned long b);
extern void addqubit(struct QState *q);
extern void swapcol(struct QState *q, long i,
    long j);
extern void printket(struct QState *q);
extern void printstate(struct QState *q);
extern void cnot(struct QState *q, long b, long
    c);
extern void hadamard(struct QState *q, long b);
extern void phase(struct QState *q, long b);
extern void xgate(struct QState *q, long b);
extern void zgate(struct QState *q, long b);
extern void ygate(struct QState *q, long b);
extern int measure(struct QState *q, long b, int
    sup);
extern void reseed(void);
extern double fidelity(struct QState *q1, struct
    QState *q2);
extern int same(struct QState *q1, struct QState
    *q2);

extern long gaussian(struct QState *q);
extern long gaussjordan(struct QState *q);

%typemap(python,ret) char * {
free($source);
}

char *statestring(struct QState *q);
char *ketstring(struct QState *q);
```

B.2.8 control.py

```
# File: control.py
# Author: Andrew Cross <awcross@mit.edu>
# Last Modified: 27 March 2004

import time
import copy
import thread
import threading
from math import *
from Numeric import *
import logger

# This module uses visual python when the
    iontrap object's
# visual flag is set to True. The visual module
    is imported
# in the block of code where it is needed,
    rather than the
# beginning, so that the simulation can be run
    without
# visualization without loading visual.

class control:

    def __init__(self,iontrap):

        # all actions return elapsed time, not
            duration
        self.actionmap = {"move":self.move,\
                          "split":self.split,\
                          "cool":self.cool,\
                          "readout":self.readout
                              ,\
                          "gate":self.gate,\
                          "join":self.join,\
                          "condition":self.
                              condition,\
                          "halt":self.halt,\
                          "fidelity":self.
                              fidelity,\
                          "test":self.test,\
                          "testequals":self.
                              testequals,\
                          "majority":self.
                              majority,\
                          "comparebits":self.
                              comparebits,\
                          "copybits":self.
                              copybits,\
                          "addbits":self.addbits
                              ,\
                          "setbitlist":self.
                              setbitlist,\
                          "displaybits":self.
                              displaybits,\
                          "message":self.message
                              ,\
                          "noise":self.noise,\
                          "same":self.same}

        # Control keeps a reference to its
            iontrap
        self.sys = iontrap
        # Times for visual display
        self.laserOnTime = 0.5  # in seconds
        self.cellMoveTime = 0.05 # in seconds
        self.sleepTime = 0.001   # in seconds
#       self.laserOnTime = 2  # in seconds
#       self.cellMoveTime = 1 # in seconds
#       self.sleepTime = 0.01    # in seconds
        self.date = None
        self.bits = {}
        self.bitlock = thread.allocate_lock()

    def reset(self):
        """reset the control object state"""
        self.date = None
        self.bits = {}
        self.bitlock = thread.allocate_lock()

    def execute(self,bundle):
        self.date = time.strftime('%d%b%H%M%S%Y'
            )
        self.sys.log.put("execute","START %s"%
            self.date,6)
        self.sys.log.put("execute","QC INITIAL\n
            %s"%str(self.sys.qc),6)
        bundle.start(self.sys.log,0)
        self.sys.log.put("execute",\
                         "END bundle.halted=%d"%
                             bundle.halted,6)
        self.sys.log.put("execute",\
                         "END bundle.time=%f"%
                             bundle.time,6)
#       self.sys.log.put("execute","QC FINAL\n%
s"%str(self.sys.qc),6)
        returnTime = copy.copy(bundle.time)
        returnHalt = copy.copy(bundle.halted)
        bundle.time = 0
        bundle.halted = False
        return [returnTime,returnHalt]

    def setup(self,bundle):
        """Set a bundle's actionmap and system
            to this control"""
        # call once, this is slow
#       bundle.reset()
        bundle.actionmap = self.actionmap
        bundle.setActionMap()
        bundle.setSys(self.sys)

    def setBit(self,cbit,val):
        self.bitlock.acquire()
        self.bits[cbit] = val
        self.bitlock.release()

    def getBit(self,cbit):
        self.bitlock.acquire()
        try:
            self.bits[cbit]
        except KeyError:
            self.bits[cbit] = 0
        c = copy.copy(self.bits[cbit])
        self.bitlock.release()
        return c
```

```
    def reportAction(self, msgstring):
        if self.sys.grid != None:
            self.sys.grid.addactionmessage(
                msgstring)

    def acquireChainLock(self, ion):
        """Acquire lock on the chain containing
        this ion object. This guards against
            deadlock"""
        notAcquired = True
        i = self.sys.nameToIon(ion)
        # self.sys.log.put("control.
            acquireChainLock",\
        #                   "getting %s's chain"%
            ion,6)
        while notAcquired:
            notAcquired = False
            i.lock.acquire()
            c = self.sys.nameToChain(ion)
            if not c.lock.acquire(False):
                # die and restart
                # self.sys.log.put("control.
                    acquireChainLock",\
                #                   "retrying %s'
                    s chain"%ion,6)
                i.lock.release()
                notAcquired = True
        i.lock.release()
        c.lockIons()
        # self.sys.log.put("control.
            acquireChainLock",\
        #                   "acquired %s's chain
            "%ion,6)

    def message(self, params, t0):
        # params = [messageString]
        messageString = params[0]
        self.sys.log.put("message", messageString
            ,6)
        if self.sys.grid != None:
            self.sys.grid.addcmdmessage(
                messageString)
        return t0

    def halt(self, params, t0):
        # params = []
        # the argument to HaltAction is the
            action time
        # DON'T CALL DIRECTLY IN A PBUNDLE
        self.sys.log.put("halt","raising
            HaltAction exception",6)
        self.reportAction("halt")
        raise 'HaltAction',t0



    def testequals(self, params, t0):
        """
                Test if the sum of the bits in a
                    bitlist equals any of
                the numbers in "nset"
        """
        # params = [cbitlist, nset, cbit]
        cbitlist = params[0]
        nset = params[1]
        cbit = params[2]
        self.sys.log.put("testequals",\
                            "cbitlist=%s"%cbitlist
                            ,6)
        setcount = 0
        for c in cbitlist:
            if self.getBit(c)==1: setcount += 1
        self.sys.log.put("testequals",\
                            "setcount=%d"%setcount
                            ,6)

        for i in nset:
                if setcount == i:
                        self.sys.log.put("
                            testequals", "%s->1
                            "%cbit,6)
                        self.reportAction("
                            testequals")
                        self.setBit(cbit,1)
                        break
                else:
                        self.sys.log.put("
                            testequals", "%s->0
                            "%cbit,6)
                        self.reportAction("
                            testequals")
                        self.setBit(cbit,0)
        return t0


    def majority(self, params, t0):
        """Performs majority voting between two
            lists"""
        # params = [cbitlist1, cbitlist2, cbit1
            , cbit2]
        # params = [cbitlist, cbit]
        if len(params) == 2:
            # we just want the majority of
                cbitlist
            cbitlist = params[0]
            cbit = params[1]
            self.reportAction("majority")
            self.sys.log.put("majority","
                cbitlist=%s"%cbitlist,6)
            count_1, count_0 = 0,0
            if len(cbitlist)%2 == 0: raise "
                Majority: evenCbitList"
            for c in cbitlist:
                if self.getBit(c)==1: count_1
                    += 1
                if self.getBit(c)==0: count_0
                    += 1
                self.sys.log.put("majority","
                    count_1=%d"%count_1,6)
                self.sys.log.put("majority","
                    count_0=%d"%count_0,6)
            if count_1 > count_0:
                self.setBit(cbit,1)
                self.sys.log.put("majority", "%s
                    ->1"%cbit,6)
            elif count_1 < count_0:
                self.setBit(cbit,0)
                self.sys.log.put("majority", "%s
                    ->0"%cbit,6)
            return t0

        elif len(params) == 4:
            # Take majority of two lists
            cbitlist1 = params[0]
            cbitlist2 = params[1]
            c1 = params[2]
            c2 = params[3]

            self.reportAction("majority")
            self.sys.log.put("majority","
                cbitlist1=%s"%cbitlist1,6)
            self.sys.log.put("majority","
                cbitlist2=%s"%cbitlist2,6)

            if len(cbitlist1) > 1:
                if self.getBit(cbitlist1[0])
                    == 0 and self.getBit(
                    cbitlist2[0]) == 0:
                    if self.getBit(cbitlist1[1])
                        == 0 and self.getBit(
                        cbitlist2[1]) == 1:
                        if self.getBit(cbitlist1
                            [2]) == 1 and self.
                            getBit(cbitlist2
                            [2]) == 1:
                            self.setBit(c1,1)
                            self.setBit(c2,1)
                            self.sys.log.put("
                                majority","
                                Special Case: %
                                s->1 and %s->1"
                                %(c1,c2),6)
                            return t0
            count1, count0 = 0,0
            for c in cbitlist1:
                if self.getBit(c)==1: count1
                    += 1
                if self.getBit(c)==0: count0
                    += 1
            self.sys.log.put("majority", "count1
                =%d"%count1,6)
            self.sys.log.put("majority", "count0
                =%d"%count0,6)
            if count1 > count0:
                self.setBit(c1,1)
                self.sys.log.put("majority", "%s
                    ->1"%c1,6)
            elif count1 < count0:
                self.setBit(c1,0)
                self.sys.log.put("majority", "%s
                    ->0"%c1,6)
            else:
```

```
                self.sys.log.put("majority","
                    unevenList1!",6)
                raise "unevenList1"

            count1,count0 = 0,0
            for c in cbitlist2:
                if self.getBit(c)==1: count1
                    += 1
                if self.getBit(c)==0: count0
                    += 1
            self.sys.log.put("majority", "count1
                =%d"%count1,6)
            self.sys.log.put("majority", "count0
                =%d"%count0,6)
            if count1 > count0:
                self.setBit(c2,1)
                self.sys.log.put("majority", "%s
                    ->1"%c2,6)
            elif count1 < count0:
                self.setBit(c2,0)
                self.sys.log.put("majority", "%s
                    ->0"%c2,6)
            else:
                self.sys.log.put("majority","
                    unevenList2!",6)
                raise "unevenList2"
            return t0

def comparebits(self,params,t0):
    """Compares two cbitlists and sets cbit
        to one if different"""
    # params = [[lists to compare],cbit]
    Lists = params[0]
    cbit = params[1]

    # It would be nice to display the three
        lists:
    for i in range(len(Lists)):
        tmpList = []
        for j in range(len(Lists[i])):
            tmpList.append(self.getBit(Lists
                [i][j]))
        self.sys.log.put("comparebits", "
            List %d = %s"%(i+1,tmpList))

    for i in range(len(Lists[0])):
        for j in range(len(Lists)-1):
            if self.getBit(Lists[j][i]) !=
                self.getBit(Lists[j+1][i]):
                self.setBit(cbit,1)
                self.sys.log.put("
                    comparebits","%s->1"%
                    cbit,6)
                return t0
    self.setBit(cbit,0)
    self.sys.log.put("comparebits","%s->0"%
        cbit,6)
    return t0



def copybits(self,params,t0):
    """Copy cbitlist1 to cbitlist2"""
    # params = [cbitlist1, cbitlist2]
    cbitlist1 = params[0]
    cbitlist2 = params[1]

    self.sys.log.put("copybits",\
                    "cbitlist1=%s"%
                        cbitlist1,6)
    self.sys.log.put("copybits",\
                    "cbitlist2=%s"%
                        cbitlist2,6)
    if len(cbitlist1) != len(cbitlist2):
            raise "copybits:
                UnequalLengthofLists"

    for i in range(len(cbitlist1)):
        value = self.getBit(cbitlist1[i])
        self.setBit(cbitlist2[i],value)

    return t0


def addbits(self, params,t0):
    """Adds Two binary lists"""
    # list1 is apparently cbit variables
    # list2 is cbit constants
    # params = [lsit1, list2, Sum(list1,
        list2)]
    list1 = params[0]
    list2 = params[1]
    Sum = params[2]

    L1,L2 = [],[]
    for i in range(len(list1)):
            L1.append(self.getBit(list1[i]))
    L2 = list2

    ## Add L1 and L2 into S
    diff = len(L1)-len(L2)
    if diff < 0: # append zeros to L1
            zeros = []
            for i in range(abs(diff)):zeros.
                append(0)
            L1 = zeros + L1
    elif diff > 0: # append zeros to L2
            zeros = []
            for i in range(diff):zeros.
                append(0)
            L2 = zeros + L2
    if len(L2) != len(L1): raise '
        Unequal_Lengths'

    S,C = [],0
    for i in range(len(L1)): S.append(0)
    for i in range(len(L1)-1, -1, -1):
            S[i] = (L1[i] ^ L2[i]) ^ C
            C = (C and (L1[i] ^ L2[i]))or(L1
                [i] and L2[i])
    ## Store the sum into Sum
    if len(Sum) != len(S): raise "
        BAD_SUM_LENGTH"

    for i in range(len(Sum)):
            value = S[i]
            self.setBit(Sum[i],value)
    self.sys.log.put("addbits",\
                "L1=%s,L2=%s,S=%s"%(L1,L2,S)
                    ,6)
    # print Sum, '--->', S
    return t0



def setbitlist(self,params,t0):
    """sets the bits in bitlist to the
        values
      given in the second input
    """
    # params = [cbitlist,bits]
    cbitlist = params[0]
    bits = params[1]
    self.sys.log.put("setbitlist","cbitlist
        =%s"%cbitlist,6)
    for i in range(len(cbitlist)):
        value = bits[i]
        self.setBit(cbitlist[i],value)
    return t0



def displaybits(self,params,t0):
    """display a list of bits"""
    cbitlist = params[0]
    self.sys.log.put("displaybits","cbitlist
        =%s"%cbitlist,6)
    hello = []
    for i in range(len(cbitlist)):
        value  = self.getBit(cbitlist[i])
        self.sys.log.put("displaybits","
            cbitlist_%d=%d"%(i,value),6)
        hello.append(value)
    print hello
    return t0



def test(self,params,t0):
    # params = [cbitlist,nset,cbit]
    cbitlist = params[0]
    nset = params[1]
    cbit = params[2]
    self.sys.log.put("test",\
                    "cbitlist=%s"%cbitlist
                        ,6)
    setcount = 0
    for c in cbitlist:
        if self.getBit(c)==1: setcount += 1
    self.sys.log.put("test",\
```

```
                                "setcount=%d"%setcount
                                    ,6)

        if setcount > nset:
                self.sys.log.put("test","%s->1"%
                        cbit,6)
                self.reportAction("test")
                self.setBit(cbit,1)
        else:
                self.sys.log.put("test","%s->0"%
                        cbit,6)
                self.reportAction("test")
                self.setBit(cbit,0)
        return t0

    def condition(self,params,t0):
        # params = [cbitlist,vallist,bundle]
        cbitlist = params[0]
        vallist = params[1]
        bundle = params[2]
        self.sys.log.put("condition",\
                         "%s ?= %s"%(cbitlist,
                              vallist),6)
        matched = True
        for i in range(len(cbitlist)):
                if self.getBit(cbitlist[i]) !=
                        vallist[i]: matched = False
        if matched:
            self.reportAction("condition")
            self.sys.log.put("condition","
                running",6)
#           bundle.reset()
#           bundle.actionmap = self.actionmap
#           bundle.setActionMap()
#           bundle.setSys(self.sys)
            bundle.start(self.sys.log,t0)
            returnTime = copy.copy(bundle.time)
            bundle.time = 0  # reset sub-bundle
                time
            if bundle.halted:
                self.sys.log.put("condition",\
                                  "HALTED t=%f"%
                                      bundle.
                                      time,6)
                bundle.halted = False  # reset
                    sub-bundle halt flag
                raise 'HaltAction',returnTime
#           self.sys.log.put("control.condition
    ",\
#                                "t=%f"%bundle.time
    ,6)
            return returnTime
        else:
            self.reportAction("condition")
            self.sys.log.put("condition","nop"
                ,6)
            return t0

    def fidelity(self,params,t0):
        """Calculate the fidelity between the
            current machine
        state and the state of qc0. Change the
            fidelity
        on the visual display and set a cbit if
            the fidelity
        is not equal to 1"""
        # params = [cbit]
        cbit = params[0]
        f = self.sys.qc.fidelity(self.sys.qc0)
        if f != 1.0: self.setBit(cbit,1)
        else: self.setBit(cbit,0)
        self.sys.lastFidelity = f
        return t0;

    def same(self,params,t0):
        """
                Test if some set of qubits has a
                    state described by
                the given string list. Just like
                    the strcmp() in C/C++
                same sets the cbit to 0 if the
                    states match and 1
                    otherwise.
        """
        # params = [ionlist,stringlist,cbit]

        ionlist = params[0]
        stringlist = params[1]
        cbit = params[2]
        f = self.sys.qc.same(ionlist,stringlist)
            #if True f = 1
        if f: self.setBit(cbit,0)
        else: self.setBit(cbit,1)
        return t0;


    def noise(self,params,t0):
        """
                Set ALL the System Noise ON or
                    OFF
        """
        switch = params[0]
        tmp_f = copy.deepcopy(self.sys.physics.
            tmpFailures)
        tmp_h = copy.deepcopy(self.sys.physics.
            tmpHeating)

        if switch == "OFF":

                self.sys.qc.noisy = False
                for key,val in self.sys.physics.
                    failures.items():
                        self.sys.physics.
                            failures[key] = 0
                for key,val in self.sys.physics.
                    heating.items():
                        self.sys.physics.heating
                            [key] = 0

        elif switch == "ON":

                self.sys.qc.noisy = True
                self.sys.physics.failures = copy
                    .deepcopy(tmp_f)
                self.sys.physics.heating  = copy
                    .deepcopy(tmp_h)

        else:
                raise "ON_or_OFF_PLEASE"

        return t0


    def readout(self,params,t0):
        """Measure an ion"""
        # params = [ion,cbit]
#           self.sys.log.put("control.readout
    ","-> %s"%params,6)
        ion = params[0]
        cbit = params[1]
        i = self.sys.nameToIon(ion)
        i.lock.acquire()
        meas = self.sys.qc.measure([ion],\
                                   [self.sys.
                                       physics.
                                       failures
                                       ['m']])
#           self.sys.log.put("readout","meas=%s"%
    meas,6)
        self.setBit(cbit,meas[0])
        self.reportAction("readout")
        self.sys.log.put("readout",\
                         "%s->%d"%(cbit,self.
                             getBit(cbit)),6)
        if self.sys.grid != None:
            import visual
            loc = i.location()
            self.sys.colorIonByState(i)
            self.sys.grid.moveion(i.loc[0]-1,i.
                loc[1]-1,visual.vector(loc[0]-i
                .loc[0],loc[1]-i.loc[1],1))
            self.sys.grid.laseron(int(loc[0])-1,
                int(loc[1])-1)
            time.sleep(self.laserOnTime)
            self.sys.grid.laseroff(int(loc[0])
                -1,int(loc[1])-1)
            self.sys.grid.moveion(i.loc[0]-1,i.
                loc[1]-1,visual.vector(loc[0]-i
                .loc[0],loc[1]-i.loc[1],0.5))
        i.lock.release()
#           self.sys.log.put("control.readout
    ","<- %s"%params,6)
        return t0+self.sys.physics.timescales["m
            "]

    def gate(self,params,t0):
        """Perform a logic gate
        gate should be a single string
        ionlist should be a list of strings
        THIS SHOULD EVENTUALLY CALL FUNCTIONS
        LIKE MS,GPHASE,CZ ...
        """
        # params = [gate,[ion,ion,...]]
```

```
        gate = params[0]
        ionlist = params[1]
        for ion in ionlist:
            i = self.sys.nameToIon(ion)
#           self.sys.log.put("control.gate","-> %s
    "%params,6)
        if len(ionlist)<2:
            # gate involves a single qubit
            i = self.sys.nameToIon(ionlist[0])
            i.lock.acquire()
            self.reportAction("%s gate"%gate)
            self.sys.qc.gates([gate],ionlist,\
                                [self.sys.physics.
                                    failures['1q'
                                    ]])
            if self.sys.grid != None:
                import visual
                loc = i.location()
                self.sys.colorIonByState(i)
                # move the ion up
                self.sys.grid.moveion(i.loc
                    [0]-1,i.loc[1]-1,visual.
                    vector(loc[0]-i.loc[0],loc
                    [1]-i.loc[1],1))
                self.sys.grid.laseron(int(loc
                    [0])-1,int(loc[1])-1)
                time.sleep(self.laserOnTime)
                self.sys.grid.laseroff(int(loc
                    [0])-1,int(loc[1])-1)
                # move the ion down
                self.sys.grid.moveion(i.loc
                    [0]-1,i.loc[1]-1,visual.
                    vector(loc[0]-i.loc[0],loc
                    [1]-i.loc[1],0.5))
            i.lock.release()
#           self.sys.log.put("control.gate
    ","<- %s"%params,6)
            return t0+self.sys.physics.
                timescales["1q"]
        else:
            # gate involves several qubits
            # we need to lock the chain
            self.acquireChainLock(ionlist[0])
            c = self.sys.nameToChain(ionlist[0])
            # check that the ionlist is in the
                chain
            for i in ionlist:
                if c != self.sys.nameToChain(i):
                    c.unlockIons()
                    c.lock.release()
                    raise "UnchainedIons",
                        ionlist
            i = c[ionlist[0]]
            self.reportAction("%s gate"%gate)
            self.sys.qc.gates([gate],[ionlist],
                len(ionlist)*[self.sys.physics.
                failures['2q']])
            if self.sys.grid != None:
                import visual
                for i in ionlist:
                    loc = c[i].location()
                    self.sys.colorIonByState(c[i
                        ])
                    self.sys.grid.moveion(c[i].
                        loc[0]-1,c[i].loc[1]-1,
                        visual.vector(loc[0]-c[
                        i].loc[0],loc[1]-c[i].
                        loc[1],1))
                    self.sys.grid.laseron(int(
                        loc[0])-1,int(loc[1])
                        -1)
                time.sleep(self.laserOnTime)
                for i in ionlist:
                    loc = c[i].location()
                    self.sys.grid.laseroff(int(
                        loc[0])-1,int(loc[1])
                        -1)
                    self.sys.grid.moveion(c[i].
                        loc[0]-1,c[i].loc[1]-1,
                        visual.vector(loc[0]-c[
                        i].loc[0],loc[1]-c[i].
                        loc[1],0.5))
            c.unlockIons()
            c.lock.release()
#           self.sys.log.put("control.gate
    ","<- %s"%params,6)
            return t0+self.sys.physics.
                timescales["2q"]

    def cool(self,params,t0):
        """Cool an ion chain"""
        # params = [ion]
        self.reportAction("cool")
        #       self.sys.log.put("control.cool
            ","-> %s"%params,9)
        ion = params[0]
        self.acquireChainLock(ion)
        c = self.sys.nameToChain(ion)
        if not self.sys.onlyGates: c.heat = 0
        if self.sys.grid != None:
            import visual
            loc = c[ion].location()
            self.sys.colorIonByState(c[ion])
            self.sys.grid.moveion(c[ion].loc
                [0]-1,c[ion].loc[1]-1,visual.
                vector(loc[0]-c[ion].loc[0],loc
                [1]-c[ion].loc[1],1))
            self.sys.grid.laseron(int(loc[0])-1,
                int(loc[1])-1)
            time.sleep(self.laserOnTime)
            self.sys.grid.laseroff(int(loc[0])
                -1,int(loc[1])-1)
            self.sys.grid.moveion(c[ion].loc
                [0]-1,c[ion].loc[1]-1,visual.
                vector(loc[0]-c[ion].loc[0],loc
                [1]-c[ion].loc[1],0.5))
        c.unlockIons()
        c.lock.release()
#       self.sys.log.put("control.cool","<- %s
    "%params,9)
        return t0+self.sys.physics.timescales["c
            "]

    def join(self,params,t0):
        """Join two ion chains"""
        # params = [ion1,ion2]
#       self.sys.log.put("control.join","-> %s
    "%params,9)
        self.reportAction("join")
        ion1 = params[0]
        ion2 = params[1]
        self.acquireChainLock(ion1)
        c1 = self.sys.nameToChain(ion1)
        if ion2 in c1:
            raise "InvalidJoin",ion1,ion2
        self.acquireChainLock(ion2)
        c2 = self.sys.nameToChain(ion2)
        d = c1 + c2
        d.lock.acquire()
        # self.sys.chains access is serialized
        ########################
        self.sys.chainslock.acquire()
        for i in d: self.sys.chains[i.name] = d
        self.sys.chainslock.release()
        ########################
        d.unlockIons()
        d.lock.release()
#       self.sys.log.put("control.join","<- %s
    "%params,9)
        return t0+self.sys.physics.timescales["j
            "]

    def split(self,params,t0):
        """Split ion chain into two chains"""
        # params = [ion1,ion2]
        ion1 = params[0]
        ion2 = params[1]
        self.reportAction("split")
        #       self.sys.log.put("control.split
            ","-> %s"%params,9)
        self.acquireChainLock(ion1)
        c = self.sys.nameToChain(ion1)
        # split() removes ions from one of
        # the chains in sys.chains without
            immediately
        # updating sys.chains, so we need to
            serialize
        # the access to sys.chains prior to the
            split
        ############
        self.sys.chainslock.acquire()
        if not self.sys.onlyGates:
                d = c.split(ion1,ion2,self.sys.
                    physics.heating['s'])
        else:
                d = c.split(ion1,ion2,0)
        for i in d: self.sys.chains[i.name] = d
        self.sys.chainslock.release()
        ############
        d.unlockIons() # locked by our last call
            to acquireChainLock
        c.unlockIons() # same
        c.lock.release()
```

```
#           self.sys.log.put("control.split","<- %s
    "%params,9)
        return t0+self.sys.physics.timescales["s
            "]

    def move(self,params,t0):
        """Move ion in a straight line to
            destination

        params = [ion,dest]
        ion is string, dest is an integer 2-
            tuple,speed is
        in cells/simtime.

        WARNING: Does not do collision detection
            right now!
        """
        ion = params[0]
        dest = params[1]
        self.reportAction("move")

        #           self.sys.log.put("control.move
            ","-> %s"%params,9)
        self.acquireChainLock(ion)
        c = self.sys.nameToChain(ion)
        i = c[ion]

        # The original locations are needed
            below
        origlocs = {}
        for key,val in c.ions.items():
            origlocs[key] = val.location()

        # Prepare to move
        dVec = array(dest) - i.location()
        mdVec = sqrt(innerproduct(dVec,dVec))
        if not c.isValidMove((dVec[0],dVec[1])):
            c.unlockIons()
            c.lock.release()
            raise "InvalidChainMove", params

        # Apply state changes
        c.move((dVec[0],dVec[1]))

        if not self.sys.onlyGates:
                # This is approximate ...
                # pfail = self.sys.physics.
                    failures["b"]*mdVec
                # This is exact

                pfail = 1 - exp(-self.sys.
                    physics.failures["b"]*mdVec
                    )
                for key,val in c.ions.items():
                    self.sys.qc.noise(val.name,
                        pfail)
                c.heat += self.sys.physics.
                    heating["m"]*mdVec

        # Display the ion motion
        if self.sys.grid != None:
            import visual
            tz = time.time()
            for key,val in c.ions.items(): self.
                sys.colorIonByState(val)
            while (time.time()-tz) < mdVec*self.
                cellMoveTime:
                time.sleep(self.sleepTime)
                for key,val in c.ions.items():
                    dVec = array(dest) -
                        origlocs[key]
                    delta = (time.time()-tz)*
                        dVec/(mdVec*self.
                        cellMoveTime)
                    delta = origlocs[key] - val.
                        loc + delta
                    self.sys.grid.moveion(val.
                        loc[0]-1,val.loc[1]-1,
                        visual.vector(delta[0],
                        delta[1],0.5))
            for key,val in c.ions.items():
                self.sys.grid.moveion(val.loc
                    [0]-1,val.loc[1]-1, visual.
                    vector(val.displacement
                    [0], val.displacement
                    [1],0.5))
        c.unlockIons()
        c.lock.release()
#           self.sys.log.put("control.move","<- %s
    "%params,9)
        return t0+self.sys.physics.timescales["b
            "]*mdVec
```

B.2.9 grid.py

```
# grid.py
# Isaac Chuang <ichuang@mit.edu>
# Andrew Cross <awcross@mit.edu>

from visual import *
from visual.text import *
import cell
import random

class grid:

    def __init__(self,gx,gy):

        self.NgridX = gx
        self.NgridY = gy
        self.drawemptygrid()
        self.time = []
        self.errors = []
        self.listofions = []
        self.statuslabel = None
        self.statelabel = None
        self.statusString = "Time-        %f s\
            nFidelity-    "%0.0
        self.messageString = "Message:\n"
        self.actionString = "Action:\n"
        self.stateString = ""

        self.celltab = []                   #
            celltab[x][y] = cell for x,y
        for kx in range(gx):
            ytab = []
            for ky in range(gy):
                # cell for each x,y grid
                    location
                ytab.append(cell.cell(kx,ky))
            self.celltab.append(ytab)

    def draw(self):
        for ytab in self.celltab:
            for acell in ytab:
                acell.draw()

    def colorion(self,gx,gy,color):
        self.celltab[gx][gy].ionobj.color =
            color

    def randomionmotion(self):
        for ions in self.listofions:
            self.moveion(ions[0],ions[1],\
                        0.1 * vector([random.
                            random(),random.
                            random(),0]))

    def drawemptygrid(self):                # draw
        the grid - empty rectangles

        fgrid = frame()                     # local
            frame for the grid rectangles
        gcolor = (0.1,0.1,0.1)              # dim
            grey color for the lines
        for k in range(0,self.NgridX+1):
            curve(frame=fgrid,pos=[(k,0,0),(k,
                self.NgridY,0)],color=gcolor)
        for k in range(0,self.NgridY+1):
            curve(frame=fgrid,pos=[(0,k,0),(self
                .NgridX,k,0)],color=gcolor)

        scene.center = vector(self.NgridX/2.0,
            self.NgridY/2.0,0) # center
        fgrid.pos = (0,0,0.1)
        self.fgrid = fgrid;

    def drawLegend(self,ioncolors):
        ## ioncolors = [datacolor,ancillacolor,
            sympatheticcolor,errorcolor]
        ## Draw the legend and initial status
            labels
#       ionlabelnames = ["Data","Ancilla","
    Sympathetic","Damaged"]
        ionlabelnames = ["Data","Ancilla","
            Sympathetic"]
        legloc = (self.NgridX+1,self.NgridY-1,0)
        msgloc = (self.NgridX+1,0.5,0)
        stateloc = (0,self.NgridY,0)
        for i in range(len(ionlabelnames)):
```

```
            thision = sphere(pos=vector(legloc)-
                vector([0,i,0]),\
                            radius=0.25,color=
                                ioncolors[i])
            label(pos=thision.pos,text=
                ionlabelnames[i],\
                    xoffset=10,yoffset=0,space
                        =0.3,height=10,\
                    box=0,line=1,opacity=0.33)
#          self.legendlabel = label(pos=legloc,\
#                                      text="Legend",
    xoffset=30,yoffset=20,space=0,\
#                                      height=10,box
    =0,line=0,opacity=0.33)
        self.statuslabel = label(pos=msgloc,\
                                    xoffset=10,
                                        yoffset=5,
                                        space=0,
                                        height
                                        =10,\
                                    box=1,line=0,\
                                    text="",opacity
                                        =0.33)
        self.statelabel = label(pos=stateloc,\
                                    xoffset=10,
                                        yoffset=5,
                                        space=0,
                                        height=10,\
                                    box=1,line=0,\
                                    text="",opacity
                                        =0.33)

    def addcmdmessage(self,msgstring):
        self.messageString = "Message:\n%s"%
            msgstring
        if self.statuslabel != None:
            self.statuslabel.text = self.
                statusString + "\n" + self.
                messageString + "\n" + self.
                actionString

    def addactionmessage(self,msgstring):
        self.actionString = "Action:\n%s"%
            msgstring
        if self.statuslabel != None:
            self.statuslabel.text = self.
                statusString + "\n" + self.
                messageString + "\n" + self.
                actionString

    def moveion(self,gx,gy,dr):
                                # displace ion
        # self.celltab[gx][gy].iondisplacement
            = dr
        self.celltab[gx][gy].ionframe.pos = self
            .celltab[gx][gy].origpos + dr

    def addplaq(self,gx,gy):
                                # add plaquette to
         grid
        self.celltab[gx][gy].type = 'plaquette'

    def electrode_s(self,gx,gy):
        # south electrode
        self.celltab[gx][gy].eflags[0] = 1

    def electrode_e(self,gx,gy):
        # east electrode
        self.celltab[gx][gy].eflags[1] = 1

    def electrode_n(self,gx,gy):
        # north electrode
        self.celltab[gx][gy].eflags[2] = 1

    def electrode_w(self,gx,gy):
        # west electrode
        self.celltab[gx][gy].eflags[3] = 1

    def addion(self,gx,gy,mycolor=color.red):
            # ion
        self.celltab[gx][gy].type = 'ion'
        self.celltab[gx][gy].color = mycolor
        self.listofions.append([gx,gy])

    def laseron(self,gx,gy,color=color.green):
            # laser
        self.celltab[gx][gy].laserflag = True
        self.celltab[gx][gy].lasercolor = color
        self.celltab[gx][gy].redraw()

    def laseroff(self,gx,gy):
        self.celltab[gx][gy].laserflag = False
        self.celltab[gx][gy].redraw()

    def detectoron(self,gx,gy,color=color.yellow
        ):      # detector
        self.celltab[gx][gy].detectflag = True
        self.celltab[gx][gy].detectcolor = color
        self.celltab[gx][gy].redraw()

    def detectoroff(self,gx,gy):
        self.celltab[gx][gy].detectflag = False
        self.celltab[gx][gy].redraw()

    def addstatus(self,thetime,thefidelity):
        # thetime should be a float
        # thefidelity is also a float
        self.statusString = "Time-        %f s\
            nFidelity -    "%thetime
        if self.statuslabel != None:
            self.statuslabel.text = self.
                statusString + "\n" + self.
                messageString + "\n" + self.
                actionString

    def addstate(self,thestate):
        self.statestring = thestate
        if self.statelabel != None:
                self.statelabel.text = self.
                    statestring

    def empty(self,gx,gy):
                                # empty
        self.celltab[gx][gy].type = 'empty'
```

B.2.10 ion.py

```
# File: ion.py
# Author: Andrew Cross <awcross@mit.edu>
# Last Modified: 27 March 2004
#
# Ion data structure

import thread
import copy
from Numeric import *

class ion:
    def __init__(self,name,type,loc):
        # loc is input as an ordered pair
        # name is string
        # type is string
        self.name = name
        self.type = type
        self.loc = array(loc)          # 2-array
            home location
        self.displacement = array([0,0]) # 2-
            array displacement from home
        self.lock = thread.allocate_lock()

    def __str__(self):
        s = ""
#        if self.lock.locked(): s = "(L) "
#        else: s = "(U) "
        s += "ion " + self.name + "," + '"' +
            self.type + '", ' \
            + "(%d,%d)"%(self.location()[0],self
                .location()[1])
        return s

    def location(self):
        """Returns the current location of the
            ion"""
        return self.loc + self.displacement

    def displacement(self):
        """Returns a copy of the current
            displacement"""
        return copy.copy(self.displacement)

    def nearby(self,location):
        """Returns true if ion is nearby
            location

        location is an ordered pair
        """
        ballRadius = 1e-6
        delta = self.location() - array(location
            )
        if innerproduct(delta,delta) <
            ballRadius**2: return True
```

```
        return False

    def passedThrough(self, location, direction):
        """Returns true if we're on the other
            side of
        a ball around location coming via
            direction
        both args are ordered pairs"""
        q = innerproduct(array(direction),
            location-self.location())
        if q<0 and not self.nearby(location):
            return True
        return False

if __name__ == "__main__":

    print "ION TEST"
    print "creating and printing ion:"
    i = ion("i","data",(0,0))
    print i
    print "nearby test:",
    if not i.nearby((-1e-8,1e-8)): print "FAILED
        "
    else: print "passed"
```

B.2.11 iontrap.py

```
# File: iontrap.py
# Author: Andrew Cross <awcross@mit.edu>
# Last Modified: 3 April 2004
#
# Object to coordinate a simulation

from Numeric import *
import cPickle, anydbm
import thread
import threading
import copy
from math import *
import random
import time
from string import *

# This module uses the visual and grid modules
    when
# the iontrap visual flag is True. These modules
    are
# imported just before they are needed so that
    when
# visual is False the code can run without
    loading
# visual.

from aqc import *
import ion
from chain import *
from logger import *
from bundle import *
from control import *
from physics import *

class iontrap:
    def __init__(self,trapfile,physics=physics()
        ,log=None):

        if log == None: self.log = logger('
            iontrap.default.log',0)
        else: self.log = log            # log for
            debug information
        self.rawchains = []
        self.chains = {}            # dictionary of
            chain objects
        # lock to serialize chains access
        self.chainslock = thread.allocate_lock()
        self.color = {} # place holder - defined
            in draw()
        self.dataqubitnames = []

        self.trap = []              # 2d list of
            string lists
        self.size = ()              # 2-tuple
        self.grid = None            # Grid object for
            visualization
        self.visual = False         # toggle visual
            display (vpython)
        self.cellsize = 10          # cell size in um
            (micrometers)
        self.qc = None              # initialize this
            in buildChains
        self.qc0 = None             # undamaged qc,
            also in buildChains
        self.method = "stabilizer"
        self.onlyGates = False # turn off all
            error sources except gates
        self.physics = physics
        self.control = control(self)
        self.lastFidelity = 1.0
        self.readTrap(trapfile)
        for c in self.chains.itervalues():
            for i in c:
                if i.type == 'data':
                    self.dataqubitnames.append(i
                        .name)

    def reset(self):
        """Reset parts of the iontrap state"""
        # too bad, some of this is slow (qc,
            buildChains)
        self.grid = None # allows it.draw() to
            rebuild grid
        self.qc = None    # initialize this in
            buildChains
        self.buildChains() # reinitializes qc,
            chains
        self.control.reset()

    def __str__(self):
        sl = []
        for c in self.chains.itervalues():
            for i in c:
                sl.append(str(i))
        sl.append(str(self.qc))
        return "".join(sl)

    def colorIonByState(self,iobj):
        # change this function to introduce
            state-dependent
        # display changes
        loc = iobj.loc
        type = iobj.type
        self.grid.colorion(int(loc[0])-1,int(loc
            [1])-1,self.colors[type])

    def draw(self):
      if self.visual and self.grid == None:
          import visual
          import grid
          self.colors = {"data":visual.color.
              green,"ancilla":visual.color.red,
              "sympathetic":visual.color.blue,"
              error":visual.color.yellow}
          visual.scene.autoscale = 1
          self.grid = grid.grid(self.size[0],
              self.size[1])
          for y in range(self.size[1]):
              for x in range(self.size[0]):
                  if 'plaquette' in self.trap[x
                      ][y]: self.grid.addplaq(x
                      ,y)
                  if 'enorth' in self.trap[x][y
                      ]: self.grid.electrode_n(
                      x,y)
                  if 'esouth' in self.trap[x][y
                      ]: self.grid.electrode_s(
                      x,y)
                  if 'ewest' in self.trap[x][y
                      ]: self.grid.electrode_w(
                      x,y)
                  if 'eeast' in self.trap[x][y
                      ]: self.grid.electrode_e(
                      x,y)
          for c in self.chains.itervalues():
              for ion in c:
                  x = ion.loc[0] - 1
                  y = ion.loc[1] - 1
                  iontype = ion.type
                  self.grid.addion(x,y,self.
                      colors[iontype])
          self.grid.draw()
          self.grid.drawLegend([self.colors["
              data"],self.colors["ancilla"],
              self.colors["sympathetic"],self.
              colors["error"]])
          visual.scene.autoscale = 0  # so
              lasers don't make the frame
              jitter

    def settitle(self,titlestring):
      if self.visual:
          import visual
          visual.scene.title = titlestring
```

```
    def setsize(self,width,height):
        if self.visual:
            import visual
            visual.scene.width = width
            visual.scene.height = height

    def readTrap(self,trapfile):
        """Reads trap structures from a file"""
        dbmIn = anydbm.open(trapfile)
        if not dbmIn.has_key("trap") or \
                not dbmIn.has_key("chains") or \
                not dbmIn.has_key("size"):
            raise "BadFormat"
        self.size = cPickle.loads(dbmIn["size"])
        self.trap = cPickle.loads(dbmIn["trap"])
        self.rawchains = cPickle.loads(dbmIn["
            chains"])
        self.buildChains()
        dbmIn.close()

    def buildChains(self):
        """Builds self.chains using self.qc and
            self.rawchains"""
        allions = []
        for c in self.rawchains:
            ions = {}
            for i in c:
                ions[i["name"]] = ion.ion(i["
                    name"],i["type"],i["loc"])
            tempchain = chain(ions)
            for i in c:
                self.chains[i["name"]] =
                    tempchain
            allions.extend(ions.keys())
        self.qc = aqc(allions,self.method,self.
            log)
        self.qc0 = aqc(allions,self.method)

    def heat(self):
        s = ""
        tot = 0.0
        for t in self.chains:
            for key,val in t.ions.items():
                s += " " + key
            s += "\t" + str(t.heat) + "\n"
            tot += t.heat
        s += "total heat = " + str(tot)
        return s

    def nameToIon(self,ion):
        """Takes ion name and returns assoc. ion
            object"""
        c = self.nameToChain(ion)
        c.lock.acquire()
        iono = c[ion]
        c.lock.release()
        return iono

    def nameToChain(self,ion):
        """Returns chain that ion name """
        self.chainslock.acquire()
        c = self.chains[ion]
        self.chainslock.release()
        return c
        # instead, a KeyError will result
        # raise 'NoChain',ion

if __name__ == "__main__":

    import sys
    import os

    if len(sys.argv) < 3:
        print "usage: iontrap bundle.bin layout.
            bin [visual? y/n]"
        sys.exit()

    bbinname = sys.argv[1]
    lbinname = sys.argv[2]
    if len(sys.argv) > 3:
            visualflag = lower(sys.argv[3])
    else:
            visualflag = "n"
    print "creating iontrap ..."
    myphys = physics()
    it = iontrap(lbinname,myphys,log=logger.
        logger("iontrap.log",9))
    it.qc.noiseType = "bitflip"
    it.qc.noisy = True
    it.onlygates = False
    if visualflag == 'y': it.visual = True
    else: it.visual = False
    it.setsize(800,600)
    print "drawing ..."
    it.draw()
    print "reading in bundle ..."
    dbmIn = anydbm.open(bbinname)
    if not dbmIn.has_key("main"): raise "
        BadFormat"
    mainprog = cPickle.loads(dbmIn["main"])
    dbmIn.close()
    print "simulating ..."
    it.control.setup(mainprog)
    print it.control.execute(mainprog)
    it.log.stop()
```

B.2.12 llparse.py

```
# File: llparse.py
# Author: Andrew Cross <awcross@mit.edu>
# Last Modified: 3 April 2004
#
# Layout Language lexer and parser

from string import *
import sys

import lex
import yacc

## Globals are not great, but they are ok
    because this module has
## its own namespace.

## my globals (not for ply lex/yacc)

lineoffset = 0          # make this nonzero if
    parsing starts mid-file
size = (0,0)            # trap size
trap = []               # trap contents
chains = []             # linear ion chains
namelist = []           # ion names

## lllexer

tokens = (
    'GRID','ION','EMPTY','FILL','LPAREN','RPAREN
        ','COMMA',
    'INTEGER','STRING','HYPHEN','LANGLE','RANGLE
        ','ID','COMMENT',
    )

t_GRID = r'grid'
t_ION = r'ion'
t_EMPTY = r'empty'
t_FILL = r'fill'
t_COMMA = r','
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LANGLE = r'\<'
t_RANGLE = r'\>'
t_HYPHEN = r'-'
t_ignore = " \t"

ll_reserved = {
    'grid' : 'GRID',
    'ion' : 'ION',
    'empty' : 'EMPTY',
    'fill' : 'FILL'
    }

def t_INTEGER(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_ID(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    t.type = ll_reserved.get(t.value,'ID')
    return t

def t_STRING(t):
    r'\"[a-zA-Z0-9 ]*\"'
    t.value = lstrip(t.value,'"')
    t.value = rstrip(t.value,'"')
    return t

def t_COMMENT(t):
    r'\#.*'
```

```
        pass

def t_newline(t):
        r'\n+'
        t.lineno += t.value.count("\n")

def t_error(t):
        global lineoffset
        print "Illegal character '%s' at %d"%(t.
            value[0],t.lineno+lineoffset)
        t.skip(1)

lllexer = lex.lex()

## llparser

def error(s,d):
        global lineoffset
        print "ERROR line %d: %s"%(d+lineoffset,
            s)
        sys.exit()

def p_llprog(t):
    'llprog : llprog statement'

def p_llprog_term(t):
    'llprog : statement'

def p_commentline(t):
    'statement : COMMENT'
    pass

def p_newgrid(t):
    'statement : GRID pair'
    global size, trap
    if not size == (0,0): error("grid
        redefinition", t.lineno)
    size = t[2]
    if size[0] < 1 or size[1] < 1: error("
        invalid grid size",t.lineno)
    # print "create grid", size
    for y in range(size[0]):
        trap.append([])
        for x in range(size[1]): trap[y].append
            (["empty"])

def p_fill(t):
    '''statement : FILL pairlist'''
    global size, trap
    if size == (0,0): error("grid undefined",t.
        lineno)
    for p in t[2]:
        if p[0] < 1 or p[0] > size[0] or \
           p[1] < 1 or p[1] > size[1]: error("
               out of bounds",t.lineno)
        # print 'fill',p
        if "ion" in trap[p[0]-1][p[1]-1]:
            trap[p[0]-1][p[1]-1] = ["plaquette",
                "ion"]
        else: trap[p[0]-1][p[1]-1] = ["plaquette
            "]

def p_fill_region(t):
    '''statement : FILL pair HYPHEN pair'''
    global size, trap
    if size == (0,0): error("grid undefined",t.
        lineno)
    p1 = t[2]
    p2 = t[4]
    if p1[0] < 1 or p1[0] > size[0] or \
       p1[1] < 1 or p1[1] > size[1]: error("out
           of bounds",t.lineno)
    if p2[0] < 1 or p2[0] > size[0] or \
       p2[1] < 1 or p2[1] > size[1]: error("out
           of bounds",t.lineno)
    if p2[0] < p1[0] or p2[1] < p1[1]:
        error("second pair must be > first",t.
            lineno)
    for x in range(p1[0],p2[0]+1):
        for y in range(p1[1],p2[1]+1):
            # print 'fill', (x,y)
            if "ion" in trap[x-1][y-1]:
                trap[x-1][y-1] = ["plaquette","
                    ion"]
            else: trap[x-1][y-1] = ["plaquette"]

def p_empty(t):
    '''statement : EMPTY pairlist'''
    global size, trap
    if size == (0,0): error("grid undefined",t.
        lineno)
    for p in t[2]:
        if p[0] < 1 or p[0] > size[0] or \
           p[1] < 1 or p[1] > size[1]: error("
               out of bounds",t.lineno)
        # print 'empty',p
        if "ion" in trap[p[0]-1][p[1]-1]:
            trap[p[0]-1][p[1]-1] = ["empty","ion
                "]
        else: trap[p[0]-1][p[1]-1] = ["empty"]

def p_empty_region(t):
    '''statement : EMPTY pair HYPHEN pair'''
    global size, trap
    if size == (0,0): error("grid undefined",t.
        lineno)
    p1 = t[2]
    p2 = t[4]
    if p1[0] < 1 or p1[0] > size[0] or \
       p1[1] < 1 or p1[1] > size[1]: error("out
           of bounds",t.lineno)
    if p2[0] < 1 or p2[0] > size[0] or \
       p2[1] < 1 or p2[1] > size[1]: error("out
           of bounds",t.lineno)
    if p2[0] < p1[0] or p2[1] < p1[1]:
        error("second pair must be > first",t.
            lineno)
    for x in range(p1[0],p2[0]+1):
        for y in range(p1[1],p2[1]+1):
            # print 'empty', (x,y)
            if "ion" in trap[x-1][y-1]:
                trap[x-1][y-1] = ["empty","ion"]
            else: trap[x-1][y-1] = ["empty"]

def p_chain(t):
    '''statement : LANGLE ionlist RANGLE'''
    global chains
    # we are guaranteed no duplicates and no
        overlaps by p_ion
    # only horizontal or vertical or vertical
        lines
    if len(t[2]) > 1:
            l = t[2]
            x1 = l[0]["loc"][0]; y1 = l[0]["loc"
                ][1]
            x2 = l[1]["loc"][0]; y2 = l[1]["loc"
                ][1]
            for i in range(len(l)-2):
                xp = l[i+2]["loc"][0]; yp = l[i
                    +2]["loc"][1]
                if y1 == y2:
                        if not yp == y2: error("
                            chain not linear",t
                            .lineno)
                else:
                        if not xp == x2: error("
                            chain not linear",t
                            .lineno)
    chains.append(t[2])

def p_ionlist(t):
    '''ionlist : ionlist ionstatement'''
    t[0] = t[1]
    t[0].append(t[2])

def p_ionlist_term(t):
    '''ionlist : ionstatement'''
    t[0] = [t[1]]

def p_ion(t):
    '''ionstatement : ION ID COMMA STRING COMMA
        pair'''
    global namelist, size, trap
    # must guarantee no overlaps and no
        duplicates
    if size == (0,0): error("grid undefined",t.
        lineno)
    if t[6][0] < 1 or t[6][0] > size[0] or \
       t[6][1] < 1 or t[6][1] > size[1]: error("
           out of bounds",t.lineno)
    if "ion" in trap[t[6][0]-1][t[6][1]-1]:
        error("ion overlaps another ion",t.
        lineno(1))
    trap[t[6][0]-1][t[6][1]-1].append("ion")
    if not t[2] in namelist:
        namelist.append(t[2])
        t[0] = {"loc":t[6], "name":t[2], "type":
            t[4]}
        # print 'ion',t[2],t[4],t[6]
    else: error("ion redeclared",t.lineno)

def p_pairlist_multi(t):
    '''pairlist : pairlist COMMA pair'''
    t[0] = t[1]
```

```
    t[0].append(t[3])

def p_pairlist_single(t):
    '''pairlist : pair'''
    t[0] = [t[1]]

def p_pair(t):
    '''pair : LPAREN INTEGER COMMA INTEGER
        RPAREN'''
    t[0] = (t[2],t[4])

def p_error(t):
        global lineoffset
        print "Syntax error '%s' at %d"%(t.value
            ,t.lineno+lineoffset)
        sys.exit()

llparser = yacc.yacc()
```

B.2.13 logger.py

```
#
# logger.py
# Andrew Cross <awcross@mit.edu>
# 27 Oct 2003
#

import Queue
import thread
import threading

# At level = 0 the logger is completely disabled
class logger:
    def __init__(self,filename,level=10):
        self._level = level
        self._alive = True
        if self._level > 0:
                self._file = file(filename,'w')
                self._queue = Queue.Queue()
                thd = threading.Thread(target=
                    self.logThread,args=())
                thd.start()

    def stop(self):
        self._alive = False

    def logThread(self):
        while self._alive:
            msg = self._queue.get()
            self._file.write(msg)
            self._file.flush()
        while self._queue.empty() == 0:
                self._file.write(self._queue.get
                    ())
        self._file.close()

    def put(self,function,message,level=5):
        if self._level > level and self._level
            != 0:
            if function!="":
                msg = function+": "+message+"\n"
            else:
                msg = message + "\n"
            self._queue.put(msg)
```

B.2.14 parse.py

```
# File: parse.py
# Author: Andrew Cross <awcross@mit.edu>
# Last Modified: 11 May 2004
#
# QCPOL parser

from bundle import *
from string import *
import cPickle, anydbm
import sys

import lex
import yacc
import polparse
import llparse

# automatically insert electrodes into the
    layout
def autoelectrodes(size,dat):
  x, y = size
  for j in range(y):
        for i in range(x):
                if dat[i][j][0] == "plaquette"
                    :
                        N, S, W, E
                            = 0, 0, 0, 0
                        if i > 0 and (dat[i
                            -1][j][0] == "
                            empty"): dat[i][j
                            ].append("ewest")
                        if i < x-1 and (dat[i
                            +1][j][0] == "
                            empty"): dat[i][j
                            ].append("eeast")
                        if j > 0 and (dat[i][j
                            -1][0] == "empty"
                            ): dat[i][j].
                            append("esouth")
                        if j < y-1 and (dat[i
                            ][j+1][0] == "
                            empty"): dat[i][j
                            ].append("enorth"
                            )
  return dat

# parse a qcpol file and write binary layout
    ltarg
# and binary program btarg
# the parser will cause the program to exit if
    the
# file cannot be parsed
def qcpolparse(fname,ltarg,btarg):

        # We have to reload the modules if this
            is the first
        # time the parse table was built
        reload(lex)
        reload(yacc)
        reload(llparse)
        reload(polparse)

        layouttag = "layout"
        qpoltag = "qcpol"

        # Read in the qcpol file, hope it's
            small
        f = file(fname,"r")
        s = f.read()
        f.close()

        # Get layout and pol code
        try:
                lstart = index(s,"<"+layouttag+"
                    >")
                lend = rindex(s,"</"+layouttag+"
                    >")
                pstart = index(s,"<"+qpoltag+">"
                    )
                pend = rindex(s,"</"+qpoltag+">"
                    )
        except ValueError:
                print "error: start/end tags
                    incorrect"
                sys.exit(1)

        # Set offsets into the file
        l = s[lstart+len("<"+layouttag+">"):lend
            ]
        llparse.lineoffset = count(s[0:lstart+
            len("<"+layouttag+">")],"\n")
        p = s[pstart+len("<"+qpoltag+">"):pend]
        polparse.lineoffset = count(s[0:pstart+
            len("<"+qpoltag+">")],"\n")

        # Parse
        llparse.llparser.parse(l,lexer=llparse.
            lllexer)
        llparse.trap = autoelectrodes(llparse.
            size,llparse.trap)
        lout = anydbm.open(ltarg,'n')
        lout['size'] = cPickle.dumps(llparse.
            size)
        lout['trap'] = cPickle.dumps(llparse.
            trap)
        lout['chains'] = cPickle.dumps(llparse.
            chains)
        lout.close()
        # set the valid ion names to those seen
            in the layout
```

```
        polparse.validionnames = llparse.
            namelist
        polparse.polparser.parse(p,lexer=
            polparse.pollexer)
        bout = anydbm.open(btarg,'n')
        bout['main'] = cPickle.dumps(polparse.
            terminalbundle)
        bout.close()

if __name__ == "__main__":

        if len(sys.argv) < 4:
                print "usage: parse source.pol
                    bundle.bin layout.bin"
                sys.exit()

        srcname = sys.argv[1]
        bbinname = sys.argv[2]
        lbinname = sys.argv[3]

        qcpolparse(srcname,lbinname,bbinname)
```

B.2.15 physics.py

```
# physics.py
# Andrew Cross
# 29 Oct 2003

import copy

class physics:

    def __init__(self,timescales=[],failures=[],
        heating=[]):

        # dictionary of times ('b','s' units of
            ms/um)
        self.cTimescales = {"1q":1e-6,"2q":10e
            -6,"m":100e-6,"c":10e-3,\
                            "b":1e-8,"s":1e-3,"
                            mem":100,"j":0}

        # dictionary of failure probabilities ('
            b' units of 1/um)
        self.cFailures = {"1q":1e-4,"2q":0.03,"m
            ":0.01,"b":0.005}

        # dictionary of heating amounts ('m' in
            units of <n>/um)
        self.cHeating = {"m":.01,"s":1}

        if timescales == []:
            self.timescales = self.cTimescales
        else:
            self.timescales = timescales
        if failures == []:
            self.failures = self.cFailures
        else:
            self.failures = failures
        if heating == []:
            self.heating = self.cHeating
        else:
            self.heating = heating

        self.tmpFailures = copy.deepcopy(self.
            failures)
        self.tmpHeating  = copy.deepcopy(self.
            heating)
```

B.2.16 polparse.py

```
# File: polparse.py
# Author: Andrew Cross <awcross@mit.edu>
# Last Modified: 5 April 2004
#
# Physical Operations Language lexer and parser

from string import *
import sys

import lex
import yacc

from bundle import *

## This file has its own namespace, so these
    globals are somewhat
## protected.

## my globals ( not for ply, lex/yacc )

lineoffset = 0                  # make nonzero
    if we start parsing mid-file
validionnames = {}              # valid ion
    names, passed in from elsewhere
terminalbundle = None           # the bundle we
    will output

## pollexer

tokens = (
        'MOVE','SPLIT','COOL','READOUT','GATE','
            JOIN','CONDITION',
        'HALT','SAME','TEST','TESTEQUALS','
            MAJORITY','COMPAREBITS',
        'COPYBITS','ADDBITS','SETBITLIST','
            DISPLAYBITS','MESSAGE','NOISE',
        'COMMA','INTEGER','ID','STRING','LPAREN'
            ,'RPAREN',
        'LANGLE','RANGLE','LBRACE','RBRACE','DEF
            ','LCURL','RCURL',
        'SELF','COMMENT',
        )

t_MOVE = r'move'
t_SPLIT = r'split'
t_COOL = r'cool'
t_READOUT = r'readout'
t_GATE = r'gate'
t_JOIN = r'join'
t_CONDITION = r'condition'
t_NOISE = r'noise'
t_HALT = r'halt'
t_SAME = r'same'
t_TEST = r'test'
t_TESTEQUALS = r'testequals'
t_MAJORITY = r'majority'
t_COMPAREBITS = r'comparebits'
t_COPYBITS = r'copybits'
t_ADDBITS = r'addbits'
t_SETBITLIST = r'setbitlist'
t_DISPLAYBITS = r'displaybits'
t_MESSAGE = r'message'
t_DEF = r'def'
t_SELF = r'self'
t_LANGLE = r'\<'
t_RANGLE = r'\>'
t_LBRACE = r'\['
t_RBRACE = r'\]'
t_LCURL = r'\{'
t_RCURL = r'\}'
t_COMMA = r','
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_ignore = " \t"

pol_reserved = {
        'move' : 'MOVE',
        'split' : 'SPLIT',
        'cool' : 'COOL',
        'readout' : 'READOUT',
        'gate' : 'GATE',
        'join' : 'JOIN',
        'condition' : 'CONDITION',
        'halt' : 'HALT',
        'same' : 'SAME',
        'test' : 'TEST',
        'noise' : 'NOISE',
        'testequals' : 'TESTEQUALS',
        'majority' : 'MAJORITY',
        'comparebits' : 'COMPAREBITS',
        'copybits' : 'COPYBITS',
        'addbits' : 'ADDBITS',
        'setbitlist' : 'SETBITLIST',
        'displaybits' : 'DISPLAYBITS',
        'message' : 'MESSAGE',
        'def' : 'DEF',
        'self' : 'SELF'}

def t_INTEGER(t):
        r'\d+'
        t.value = int(t.value)
        return t

def t_ID(t):
        r'[a-zA-Z_][a-zA-Z0-9_]*'
        t.type = pol_reserved.get(t.value,'ID')
```

```
        return t

def t_STRING(t):
        r'\"[a-zA-Z0-9 ]*\"'
        t.value = lstrip(t.value,'"')
        t.value = rstrip(t.value,'"')
        return t

def t_COMMENT(t):
        r'\#.*'

def t_newline(t):
        r'\n+'
        t.lineno += t.value.count("\n")

def t_error(t):
        global lineoffset
        print "Illegal character '%s' at %d"%(t.
            value[0],t.lineno+lineoffset)
        t.skip(1)

pollexer = lex.lex()

## polparser

# Global Variables to manage subroutines
subroutines = {}
selfreference = False

def error(s,d):
        global lineoffset
        print "ERROR line %d: %s"%(d+lineoffset,
            s)
        sys.exit()

def p_terminus(t):
        'terminus : program'
        global selfreference
        global terminalbundle
        if t[1] == 'def':
                print 'no code, only subroutines
                    '
                t[0] = t[1]
        elif type(t[1]) is list:
                t[0] = sbundle(t[1],False)
        elif isinstance(t[1],bundle):
                t[0] = t[1]
        else: error('bad type %s'%t[1],t.lineno
            (1))
        if selfreference: error('meta self-
            reference',t.lineno(1))
        # if t[1] != 'def': print 'final bundle
            :',t[0]
        # if len(subroutines)>0:
                # print 'subroutines'
                # print '----'
                # for k,v in subroutines.items()
                    :
                        # print "%s: %s"%(k,v)
        terminalbundle = t[0]

def p_program(t):
        '''program : program define
                   | program statement'''
        if t[2] == 'def':
                t[0] = t[1]
        else:
                if isinstance(t[1],bundle) and
                    isinstance(t[2],bundle):
                        t[0] = [t[1],t[2]]
                        # print ' prog bundle-
                            bundle implicit
                            serialization'
                elif isinstance(t[1],bundle) and
                    type(t[2]) is list:
                        t[0] = [t[1]]
                        t[0].extend(t[2])
                        # print ' prog bundle-
                            list implicit
                            serialization'
                elif type(t[1]) is list and
                    isinstance(t[2],bundle):
                        t[0] = t[1].append(t[2])
                        # print ' prog list-
                            bundle implicit
                            serialization'
                elif type(t[1]) is list and type
                    (t[2]) is list:
                        t[0] = t[1].extend(t[2])
                        # print ' prog list-list
                            implicit
                            serialization'
                elif t[1] == 'def':
                        t[0] = t[2]
                else: error('bad type %s'%[t[1],
                    t[2]],t.lineno(1))

def p_program_base(t):
        '''program : define
                   | statement
                   | statement COMMENT
                   | define COMMENT'''
        t[0] = t[1]

def p_def(t):
        'define : DEF ID LCURL statement RCURL'
        global selfreference
        if subroutines.has_key(t[2]):
                error('subroutine %s redefined'%
                    t[2],t.lineno(2))
        if type(t[4]) is list:
                subroutines[t[2]] = sbundle(t
                    [4],False)
                if selfreference:
                        # print ' attach self
                            reference in def'
                        subroutines[t[2]].attach
                            (subroutines[t[2]])
                        selfreference = False
                # print ' sbundle built from
                    bundle list in def:',t[2]
        elif isinstance(t[4],bundle):
                subroutines[t[2]] = t[4]
                # print ' single bundle not
                    serialized in def:',t[2]
                if selfreference:
                        # print ' attach self
                            reference in def'
                        subroutines[t[2]].attach
                            (subroutines[t[2]])
                        selfreference = False
        else: error('bad type %s'%t[4],t.lineno
            (4))
        t[0] = 'def'

def p_parallel(t):
        'statement : LBRACE statement RBRACE'
        if type(t[2]) is list:
                t[0] = pbundle(t[2])
                # print ' pbundle built from
                    bundle list:',t[0]
        elif isinstance(t[2],bundle):
                t[0] = t[2]
                # print ' single bundle not
                    parallelized:',t[0]
        else: error('bad type %s'%t[2],t.lineno
            (2))

def p_serial(t):
        'statement : LANGLE statement RANGLE'
        if type(t[2]) is list:
                t[0] = sbundle(t[2],False)
                # print ' sbundle built from
                    bundle list:',t[0]
        elif isinstance(t[2],bundle):
                t[0] = t[2]
                # print ' single bundle not
                    serialized:',t[0]
        else: error('bad type %s'%t[2],t.lineno)

def p_statement_list(t):
        'statement : statement statement'
        if isinstance(t[1],bundle) and
            isinstance(t[2],bundle):
                t[0] = [t[1],t[2]]
                # print ' bundle-bundle implicit
                    serialization'
        elif isinstance(t[1],bundle) and type(t
            [2]) is list:
                t[0] = [t[1]]
                t[0].extend(t[2])
                # print ' bundle-list implicit
                    serialization'
        elif type(t[1]) is list and isinstance(t
            [2],bundle):
                t[0] = t[1].append(t[2])
                # print ' list-bundle implicit
                    serialization'
        elif type(t[1]) is list and type(t[2])
            is list:
                t[0] = t[1].extend(t[2])
                # print ' list-list implicit
                    serialization'
```

```
        else: error('bad type %s'%[t[1],t[2]],t.
            lineno)

def p_statement_expression(t):
        'statement : expression'
        t[0] = t[1]
        # print ' terminating action:',t[0]

def p_expression_id(t):
        'expression : ID'
        if subroutines.has_key(t[1]):
                # defined subroutine
                t[0] = subroutines[t[1]]
        else:
                error('undefined subroutine %s'%
                    t[1],t.lineno(1))

def p_expression_halt(t):
        'expression : HALT'
        t[0] = action(["halt",[]])

def p_expression_message(t):
        'expression : MESSAGE STRING'
        t[0] = action(["message",[t[2]]])

def p_expression_testequals(t):
        'expression : TESTEQUALS idlist COMMA
            INTEGER COMMA ID'
        t[0] = action(["testequals",[t[2],t[4],t
            [6]]])

def p_expression_majority(t):
        '''expression : MAJORITY idlist COMMA
            idlist COMMA ID COMMA ID
                        | MAJORITY idlist COMMA ID'
                            ''
        if type(t[4]) is list:
                t[0] = action(["majority",[t[2],
                    t[4],t[6],t[8]]])
        else:
                t[0] = action(["majority",[t[2],
                    t[4]]])

def p_expression_comparebits(t):
        'expression : COMPAREBITS idlist COMMA
            idlist COMMA ID'
        t[0] = action(["comparebits",[[t[2],t
            [4]],t[6]]])

def p_expression_copybits(t):
        'expression : COPYBITS idlist COMMA
            idlist'
        t[0] = action(["copybits",[t[2],t[4]]])

def p_expression_addbits(t):
        'expression : ADDBITS idlist COMMA blist
            COMMA idlist'
        t[0] = action(["addbits",[t[2],t[4],t
            [6]]])

def p_expression_noise(t):
        'expression : NOISE STRING'
        t[0] = action(["noise",[t[2]]])

def p_expression_setbitlist(t):
        'expression : SETBITLIST idlist COMMA
            blist'
        t[0] = action(["setbitlist",[t[2],t
            [4]]])

def p_expression_displaybits(t):
        'expression : DISPLAYBITS idlist'
        t[0] = action(["displaybits",[t[2]]])

def p_expression_test(t):
        'expression : TEST idlist COMMA INTEGER
            COMMA ID'
        t[0] = action(["test",[t[2],t[4],t[6]]])

def p_expression_same(t):
        'expression : SAME idlist COMMA STRING
            COMMA ID'
        t[0] = action(["same",[t[2],t[4],t[6]]])

def p_expression_condition(t):
        '''expression : CONDITION idlist COMMA
            blist COMMA ID
                        | CONDITION idlist COMMA
                            blist COMMA SELF'''
        global selfreference
        if t[6] == 'self':
                # self reference in def, or
                    error
                selfreference = True
                t[0] = action(["condition",[t
                    [2],t[4],None]])
        elif subroutines.has_key(t[6]):
                t[0] = action(["condition",[t
                    [2],t[4],subroutines[t
                    [6]]]])
        else: error('undefined subroutine %s'%t
            [6],t.lineno(6))

def p_expression_readout(t):
        'expression : READOUT ID COMMA ID'
        global validionnames
        if not t[2] in validionnames: error("
            invalid ion name",t.lineno(2))
        t[0] = action(["readout",[t[2],t[4]]])

def p_expression_gate(t):
        'expression : GATE STRING COMMA idlist'
        global validionnames
        if len(t[4]) > 2:
                error('too many IDs %s'%t[4],t.
                    lineno)
        for n in t[4]:
                if not n in validionnames: error
                    ("invalid ion name",t.
                    lineno(4))
        t[0] = action(["gate",[t[2],t[4]]])

def p_expression_cool(t):
        'expression : COOL ID'
        global validionnames
        if not t[2] in validionnames: error("
            invalid ion name",t.lineno(2))
        t[0] = action(["cool",[t[2]]])

def p_expression_join(t):
        'expression : JOIN ID COMMA ID'
        global validionnames
        if not t[2] in validionnames: error("
            invalid ion name",t.lineno(2))
        if not t[4] in validionnames: error("
            invalid ion name",t.lineno(4))
        t[0] = action(["join",[t[2],t[4]]])

def p_expression_split(t):
        'expression : SPLIT ID COMMA ID'
        global validionnames
        if not t[2] in validionnames: error("
            invalid ion name",t.lineno(2))
        if not t[4] in validionnames: error("
            invalid ion name",t.lineno(4))
        t[0] = action(["split",[t[2],t[4]]])

def p_expression_move(t):
        'expression : MOVE ID COMMA LPAREN
            INTEGER COMMA INTEGER RPAREN'
        global validionnames
        if not t[2] in validionnames: error("
            invalid ion name",t.lineno(2))
        t[0] = action(["move",[t[2],(t[5],t[7])
            ]])

def p_idlist(t):
        'idlist : LPAREN csepid RPAREN'
        t[0] = t[2]

def p_csepid(t):
        '''csepid : csepid COMMA ID
                  | ID'''
        if type(t[1]) is list:
                t[0] = t[1]
                t[1].append(t[3])
        else:
                t[0] = [t[1]]

def p_blist(t):
        'blist : LPAREN csepbl RPAREN'
        t[0] = t[2]

def p_csepbl(t):
        '''csepbl : csepbl COMMA INTEGER
                  | INTEGER'''
        if type(t[1]) is list:
                t[0] = t[1]
                t[0].append(t[3])
        else:
                t[0] = [t[1]]

def p_error(t):
```

```
        global lineoffset
        try:
                print "Syntax error '%s' at %d"
                    %(t.value,t.lineno+
                    lineoffset)
        except AttributeError:
                print "Syntax error with
                    inaccessible token
                    information"
        sys.exit()

polparser = yacc.yacc()
```

B.2.17 propagationMethod.py

```
# File: propagationMethod.py
# Author: Andrew Cross <awcross@mit.edu>
# Last Modified: 26 March 2004
#
# Error propagation simulation method

import copy

class propagationMethod:

    """Classical error propagation simulation
        method"""
    def __init__(self,names):

        self.cValidOps = ["h","s","cnot","x","y"
            ,"z"]
        self.qubits = []
        self.state = {}
        if len(names)>0:
            for n in names:
                self.qubits = names
                self.state[n] = [0,0]   # [x,z]
        else: raise 'badInit', names

    def fidelity(self,method):

        if self.qubits != method.qubits: raise "
            badState"
        for n in self.qubits:
            if self.state[n] != method.state[n
                ]: return 0
        return 1

    def same(self,names,initstring):

            raise "unimplemented"

    def reset(self,names):

        for q in names:
            self.state[q] = [0,0]

    def gates(self,ops,qubits):

        """Example:
        gates(['h','cnot'],['q1',['q2','q3']])""
            "

        for o in ops:
            if not o in self.cValidOps: raise "
                badOp", o

        for g in range(len(ops)):
            if ops[g] == "h":
                t = copy.copy(self.state[qubits[
                    g]])
                self.state[qubits[g]] = [t[1],t
                    [0]]
            elif ops[g] == "s":
                t = copy.copy(self.state[qubits[
                    g]])
                if t == [1,1]: self.state[qubits
                    [g]] = [1,0]
                if t == [1,0]: self.state[qubits
                    [g]] = [1,1]
            elif ops[g] == "cnot":
                if self.state[qubits[g
                    ][0]][0] == 1:
                    self.state[qubits[g
                        ][1]][0] = (self.state[
                        qubits[g][1]][0] + 1)%2
                if self.state[qubits[g
                    ][1]][1] == 1:
                    self.state[qubits[g
                        ][0]][1] = (self.state[
                        qubits[g][0]][1] + 1)%2
            elif ops[g] == "x":
                self.state[qubits[g]][0] = (self
                    .state[qubits[g]][0] + 1)%2
            elif ops[g] == "y":
                self.state[qubits[g]][0] = (self
                    .state[qubits[g]][0] + 1)%2
                self.state[qubits[g]][1] = (self
                    .state[qubits[g]][1] + 1)%2
            elif ops[g] == "z":
                self.state[qubits[g]][1] = (self
                    .state[qubits[g]][1] + 1)%2
            else: raise "badOp", ops[g]

    def measure(self,qubits):

        outcomes = []
        for q in qubits:
            if self.state[q][0] == 1: outcomes.
                append(1)
            else: outcomes.append(0)
            if self.state[q][1] == 1: self.state
                [q][1] = 0
        return outcomes

    def add(self,names):

        if len(self.qubits)==0:
            self.__init__(names)
        else:
            self.qubits.extend(names)
            for q in names:
                self.state[q] = [0,0]

    def remove(self,names):

        for q in names:
            self.qubits.remove(q)
            del self.state[q]

    def __str__(self):

        sl = []
        for q in self.qubits:
            sl.append(q)
            sl.append(":%s\n"%self.state[q])
        return "".join(sl)
```

B.2.18 pureMethod.py

```
# File: pureMethod.py
# Author: Andrew Cross <awcross@mit.edu>
# Last Modified: 26 March 2004
#
# Pure state simulation method

from math import *
from Numeric import *
import random
import string

class pureMethod:

    """Pure state circuit simulation method"""
    def __init__(self,names):

        self.x = array([[0,1],[1,0]])
        self.y = array([[0,-1j],[1j,0]])
        self.z = array([[1,0],[0,-1]])
        self.s = array([[1,0],[0,1j]])
        self.cnot = array([[1,0,0,0],[0,1,0,0],\
                        [0,0,0,1],[0,0,1,0]])
        self.h = 1/sqrt(2)*array([[1,1],[1,-1]])
        self.cValidOps = ["h","s","cnot","x","y"
            ,"z"]

        self.qubits = names
        self.state = zeros(2**len(names))
        self.state[0] = 1

    def fidelity(self,method): raise "
        unimplemented"

    def same(self,names,initstring): raise "
        unimplemented"
```

```
    def reset(self,names): raise "unimplemented"

    def gates(self,ops,qubits): raise "
        unimplemented"

    def measure(self,qubits):

        epsilon = 1e-6
        n = len(self.qubits)
        outcomes = []
        if innerproduct(self.state,self.state)
            -1 > epsilon: raise "badState"
        for q in qubits:
            l = self.qubits.index(q)
            P = self.kron(self.kron(ones(2**(n-l
                )),array([1,0])),ones(2**(l-1))
                )
            p0 = innerproduct(P*self.state,P*
                self.state)
            r = random.random()
            if r<p0:
                outcomes.append(0)
                self.state = P*self.state/sqrt(
                    p0)
            else:
                outcomes.append(1)
                P = self.kron(self.kron(\
                    ones(2**(n-l)),array([0,1]))
                    ,\
                    ones(2**(l-1)))
        return outcomes

    def add(self,names):

        if len(self.qubits) == 0:
            self.__init__(names)
        else:
            self.qubits.extend(names)
            state = zeros(2**len(names))
            state[0] = 1
            self.state = self.kron(self.state,
                state)

    def remove(self,names): raise "unimplemented
        "

    def __str__(self):
        return "unimplemented"

    def tpmul(self,ops):
        """Matrix-vector multiply for kronecker
            products

        Takes a list of arrays specifying the
            individual
        terms in the tensor product matrix.
        """
        k = len(ops)
        n = len(self.state)
        # Compute the dimensions of the tensor
            product
        d = array([1,1])
        for i in range(0,k): d *= array(ops[i].
            shape)
        if n%d[1]!=0 or d[1]>n: raise "
            tpmulIncompatible"
        # If the tensor product is too small,
            assume the first
        # term is the identity (really should
            assume the last
        # term is the identity though).
        if d[1]<n: self.state = transpose(
            reshape(self.state,(n/d[1],d[1])))
        for i in range(k-1,-1,-1):
            t = ops[i].shape[1]
            self.state = reshape(self.state,(t,n
                /t))
            self.state = transpose(
                matrixmultiply(ops[i],self.
                state))
            self.state = reshape(self.state,(n
                ,-1))
            self.state = self.state.flat

    def kron(self,a1,a2):
        """Kronecker product of two matrices or
            two vectors"""
        # The try blocks and if statements
        # handle the case where the inputs are
        # vectors
        n1 = a1.shape[0]
        n2 = a2.shape[0]
        try: m1 = a1.shape[1]
        except IndexError: m1 = 1
        try: m2=a2.shape[1]
        except IndexError: m2=1
        if m1*m2 > 1:
            result = zeros((n1*n2,m1*m2))
        else:
            result = zeros(n1*n2)
        for i in range(0,n1):
            for j in range(0,m1):
                if m1==1: block = a1[i]*a2
                else:     block = a1[i,j]*a2
                if m1*m2==1: result[n2*i:n2*(i
                    +1)] = block
                else: result[n2*i:n2*(i+1), \
                          m2*j:m2*(j+1)] =
                              block
        return result
```

B.2.19 stabilizerMethod.py

```
# Name: stabilizerMethod.py
# Authors: Andrew Cross <awcross@mit.edu> (
    Python wrapper)
# Last Modified: 26 March 2004
#
# Stabilizer method implementation.

import chp      # Scott Aaronson's CHP simulator
    , modified heavily
import copy

class stabilizerMethod:

    """Stabilizer circuit simulation method"""
    def __init__(self,names):

        self.cValidOps = ["h","s","cnot","x","y"
            ,"z"]
        self.qubits = {}
        if len(names)>0:
            self.state = chp.initialize(len(
                names),"".join(['z']*len(names)
                ))
        else: raise 'badInit', names
        for q in names: self.qubits[q] = names.
            index(q)

    def reseed(self):

        chp.reseed()


    def same(self,names,initstring):
        """
        Test for equality of the states of a list
            of qubits
        against a perfect list of qubits.

        1 -- same
        0 -- different
        """
        if not type(initstring) is str: raise "
            badInitString"
        # create QState structures
        q = chp.initialize(len(initstring),
            initstring)
        p = chp.copy(self.state)
        # create name dictionaries
        pnames = copy.copy(self.qubits)
        qnames = {}
        for n in names: qnames[n] = names.index(n)
        # remove all the qubits from p not in
            names
        for n in self.qubits:
            if not n in names:
                chp.removequbit(p,pnames[n])
                rloc = pnames[n]
                for k in pnames.keys():
                    if pnames[k] > rloc:
                        pnames[k] = pnames[k] - 1
                del pnames[n]
        # reorder all the qubits in p
        for n in qnames.keys():
            if pnames[n] != qnames[n]:
                print "swapping %d with %d"%(
                    pnames[n],qnames[n])
                chp.swapcol(p,pnames[n],qnames[n])
```

```
            temp = pnames[n]
            for m in pnames.keys():
                if pnames[m] == qnames[n]:
                    pnames[m] = temp
            pnames[n] = qnames[n]

    # test equality
    sametest = chp.same(q,p)
    chp.freestate(q)
    chp.freestate(p)
    return sametest

def displayket(self):
    """ Displaying the Ket using CHP """
    chp.printket(self.state)


def fidelity(self,method):

    if self.qubits != method.qubits: raise "
        badState"
    return chp.fidelity(self.state,method.
        state)



def reset(self,names):

    for q in names:
        rbit = chp.measure(self.state,self.
            qubits[q],0)
        if rbit in [1,3]: chp.xgate(self.
            state,self.qubits[q])



def gates(self,ops,qubits):

    """Example:
    gates(['h','cnot'],['q1',['q2','q3']])""
    "
    for o in ops:
        if not o in self.cValidOps: raise "
            badOp", o
    for g in range(len(ops)):
        if ops[g] == "h":
            chp.hadamard(self.state,self.
                qubits[qubits[g]])
        elif ops[g] == "s":
            chp.phase(self.state,self.qubits
                [qubits[g]])
        elif ops[g] == "cnot":
            chp.cnot(self.state,self.qubits[
                qubits[g][0]],\
                    self.qubits[qubits[g
                        ][1]])
        elif ops[g] == "x":
            chp.xgate(self.state,self.qubits
                [qubits[g]])
        elif ops[g] == "y":
            chp.ygate(self.state,self.qubits
                [qubits[g]])
        elif ops[g] == "z":
            chp.zgate(self.state,self.qubits
                [qubits[g]])
        else: raise "badOp", ops[g]

def measure(self,qubits):
    """Projectively measure a list of qubits
    """

    outcomes = []
    for q in qubits:
        m = chp.measure(self.state,self.
            qubits[q],0)
        if m < 2: outcomes.append(m)
        else: outcomes.append(int(m)-2)
    return outcomes

def add(self,names):

    # there will be no duplicate names,
        because that is
    # enforced here and in __init__.
    iloc = len(self.qubits)
    for n in names:
            if self.qubits.has_key(n): raise
                'duplicateName', n
            chp.addqubit(self.state)
            self.qubits[n] = iloc
            iloc = iloc + 1

def remove(self,names):

    # if a name is invalid, a KeyError will
        get raised
    for n in names:
            chp.removequbit(self.state,self.
                qubits[n])
            rloc = self.qubits[n]
            for k in self.qubits.keys():
                    if self.qubits[k] > rloc
                        :
                            self.qubits[k
                                ] = self.
                                qubits[k
                                ] - 1
            del self.qubits[n]

def generators(self):
    return chp.statestring(self.state)

def ket(self):
    return chp.ketstring(self.state)

def __str__(self):
    return self.generators()
#       return self.ket()
```

Appendix C

Quantum Architecture (ARQ) Simulator Source Code

C.1 Module Dependencies

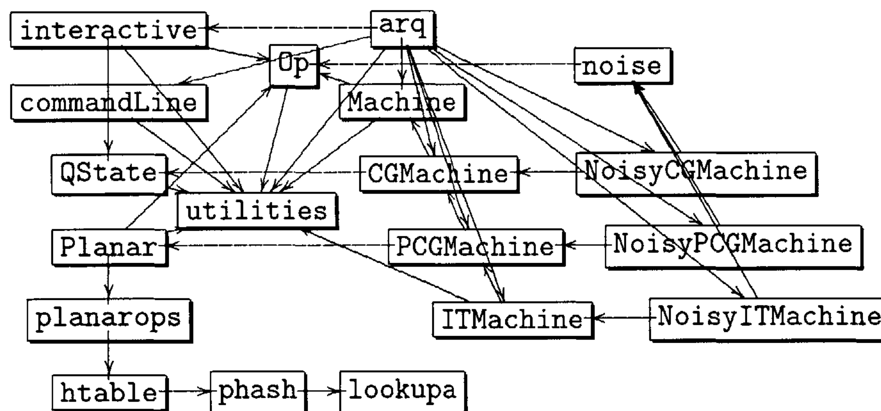


Figure C-1: This diagram shows ARQ source dependencies. Each source file at an arrow tail includes the source file at the arrow head.

The quantum architecture simulator (ARQ) is a standalone program written in C++ using the standard template library (STL). The main program, `arq`, creates an appropriate `Machine` from the hierarchy: `Machine`, `CGMachine`, `PCGMachine`, `ITMachine`, `NoisyCGMachine`, `NoisyPCGMachine`, and `NoisyITMachine`. Each machine may need a quantum state `QState` or a planar layout `Planar`. The user interface modules are `interactive`, `commandLine`, and `Op`.

C.2 Source Listing

C.2.1 CGMachine.cc

```
// CGMachine.cxx
// Added new logging scheme  07/19/04  ddthaker
// $Id: CGMachine.cc 223 2004-12-08 00:52:51Z
    awcross $
// Clifford group machine

#include "include/globals.hh"
#include "include/CGMachine.hh"
#include "include/loglevels.hh"

CGMachine::CGMachine()
{
        qubitCounter = 0;
}

void CGMachine::save(void)
{
        // backup our state
        qbak = qs;
        Machine::save();
}

void CGMachine::restore(void)
{
        // restore our state
        qs = qbak;
        // restore the machine within
        Machine::restore();
}

// Preprocess the instruction array. Do things
    like initialize
// information and set instruction times.
// Only quit if you recognize a critical error
    within your jurisdiction.
// Only warn of unknown operations in debug mode
    .
bool CGMachine::process(void)
{
    long i;
    vector<Op>::iterator j;
    vector<string >::iterator s;

    // First, process "seq" using Machine
    if( !Machine::process() ) return false;

    // If that didn't fail, go ahead and do our
        work.
    for( i = 0; i < seq.size(); i++ )
    {
        for( j = seq[i].begin(); j < seq[i].end
            (); j++ )
        {
            // QUBIT NAME, ..., NAME
            if( j->opcode == "qubit" )
            {
                if (log_level & DEBUG) cerr << "
                    DBG processing qubit " << i
                     << endl;

                for( s = j->args.begin(); s < j
                    ->args.end(); s++ )
                {
                    if( hasqubit(*s) )
                        cerr << "Warning: " << *
                            s << "redeclared."
                             << endl;
                    else
                    {
                        if (log_level & DEBUG)
                            cerr <<"DBG add
                            qubit "<< *s <<" "
                             << i << endl;

                        if (log_level & INIT)
                            cout << "qubit "
                             << *s << endl;

                        setqubit(*s);
                        qs.add();
                    }
                }
                j->opcode = "nop";
            }
            // MEASURE BIT,QUBIT
            else if( j->opcode == "measure" )
```

```
            {
                if (log_level & DEBUG) cerr << "
                    DBG processing measure "
                     << " " << i << endl;

                // FIX - check args.size() == 2
                    first
                if( !hasbit(j->args[0]) )
                {
                        if (log_level & INIT)
                            cout << "Adding bit
                             " << *s << endl;
                        setbit(*s,0);
                }
                if( !hasbit(j->args[0]) || !
                    hasqubit(j->args[1]))
                {
                    cerr << "Error! Incorrect
                        arguments for "
                         << "\"measure bit,qubit
                             \" at instruction
                             "
                         << i << endl;
                    return false;
                }
            }
            // GATE QUBIT
            else if( j->opcode == "x"  ||
                     j->opcode == "y"  ||
                     j->opcode == "z"  ||
                     j->opcode == "s"  ||
                     j->opcode == "h"  )
            {
                if (log_level & DEBUG) cerr << "
                    DBG processing "<< j->
                     opcode <<" " << i <<endl;

                // FIX check args.size()
                if( !hasqubit(j->args[0]) )
                {
                    cout << j->args[0] << endl;
                    cerr << "Error! Incorrect
                        arguments for "
                         << "\"" << j->opcode
                             << " qubit\" at
                             instruction "
                         << i << endl;
                    return false;
                }
            }
            // CNOT QUBIT, QUBIT
            else if( j->opcode == "cnot" )
            {

                    if (log_level & DEBUG) cerr
                         << "DBG processing
                        cnot " << " " << i <<
                        endl;

                // FIX check args.size()
                if( !hasqubit(j->args[0]) || !
                    hasqubit(j->args[1]) )
                {
                    cerr << "Error! Incorrect
                        arguments for "
                         << "\"cnot qubit,qubit
                             \" at instruction
                             "
                         << i << endl;
                    return false;
                }
            }
            // CZ QUBIT, QUBIT
            else if( j->opcode == "cz" )
            {
                if (log_level & DEBUG) cerr << "
                    DBG processing cz " << " "
                     << i << endl;

                // FIX check args.size()
                if( !hasqubit(j->args[0]) || !
                    hasqubit(j->args[1]) )
                {
                    cerr << "Error! Incorrect
                        arguments for "
                         << "\"cz qubit,qubit\"
                             at instruction "
                         << i << endl;
                    return false;
                }
            }
            // SEED INTEGER
```

```
else if( j->opcode == "seed" )
{
    if (log_level & DEBUG) cerr << "
        DBG processing seed " << "
        " << i << endl;

    // FIX check args.size()
    if( !isInteger(j->args[0]))
    {
        cerr << "Error! Incorrect
            arguments for "
            << "\"seed integer\" at
                instruction "
            << i << endl;
        return false;
    }
}
else if( j->opcode == "logtableau"
      || j->opcode == "logket"
            || j->opcode == "
                logstate")
{
    if (log_level & DEBUG) cerr << "
        DBG processing logtableau ,
        logket , or logstate" << " "
        << i << endl;
}
// SUBSET CBIT, N, QUBIT1, ... ,
    QUBITN,SGEN1, ... , SGENN
// We'll check that CBITS and QUBITS
    exist , are not
// written multiple times , and that
    the generators
// have the correct number of
    characters from the
// correct set of letters.
else if( j->opcode == "subset" )
{
    bool bail = false;
    // FIX - check args.size
    int n = str2int(j->args[1]); //
        FIX
    map<string,bool> seen;
    // check bit argument
    if( !hasbit(j->args[0]) )
    {
            if (log_level & INIT)
                cout << "Adding bit
                " << *s << endl;
            setbit(*s,0);
    }
    if( !hasbit(j->args[0]) ) bail
        = true;
    // check qubit arguments
    for( int i=2; i < 2+n; i++ )
    {
            if( !hasqubit(j->args[i
                ]) ) bail = true;
            if( seen[j->args[i]] )
                bail = true;
            seen[j->args[i]] = true;
    }
    // check number of arguments
    if( j->args.size() != 2 + 2*n )
        bail = true;
    // don't bother with the rest of
        now -
    // QState will issue warnings.
        FIX later.
    if( bail ) return false;
}
// SUBSETKET N, QUBIT1, ... , QUBITN
else if( j->opcode == "subsetket" )
{
    bool bail = false;
    // FIX - check args.size
    int n = str2int(j->args[0]); //
        FIX
    map<string,bool> seen;
    // check qubit arguments
    for( int i=1; i < 1+n; i++ )
    {
            if( !hasqubit(j->args[i
                ]) ) bail = true;
            if( seen[j->args[i]] )
                bail = true;
            seen[j->args[i]] = true;
    }
    // check number of arguments
    if( j->args.size() != 1 + n )
        bail = true;
```

```
            // don't bother with the rest of
                now -
            // QState will issue warnings.
                FIX later.
            if( bail ) return false;
        }
        // UNKNOWN
        else
        {
            if (log_level & DEBUG) cerr << "
                DBG Warning! Opcode " << j
                ->opcode << " unknown to
                CGMachine" << endl;
            continue; // don't set the
                number of clock cycles
        }

        // Set the number of clock cycles
            for the processed instruction
        j->nticks = times[j->opcode];
    }
  }
  return true;
}

bool CGMachine::exec(Op& op, bool log)
{

    if( op.opcode == "h" )
    {
        if(log && log_level & GATES) cout << op
            << endl;
        qs.h(getqubit(op.args[0]));
    }
    else if( op.opcode == "cnot" )
    {
        if(log && log_level & GATES) cout << op
            << endl;
        qs.cnot(getqubit(op.args[0]),getqubit(op
            .args[1]));
    }
    else if( op.opcode == "cz" )
    {
        if(log && log_level & GATES) cout << op
            << endl;

        qs.h(getqubit(op.args[1]));
        qs.cnot(getqubit(op.args[0]),getqubit(op
            .args[1]));
        qs.h(getqubit(op.args[1]));
    }
    else if( op.opcode == "s" )
    {
        if(log && log_level & GATES) cout << op
            << endl;
        qs.s(getqubit(op.args[0]));
    }
    else if( op.opcode == "x" )
    {
        if(log && log_level & GATES) cout << op
            << endl;
        qs.gx(getqubit(op.args[0]));
    }
    else if( op.opcode == "y" )
    {
        if(log && log_level & GATES) cout << op
            << endl;
        qs.gy(getqubit(op.args[0]));
    }
    else if( op.opcode == "z" )
    {
        if(log && log_level & GATES) cout << op
            << endl;
        qs.gz(getqubit(op.args[0]));
    }
    else if( op.opcode == "measure" )
    {
        if(log && log_level & GATES) cout << op
            << endl;
        setbit(op.args[0],qs.measure(getqubit(op
            .args[1]),0)%2);
    }
    else if( op.opcode == "seed" )
    {
        // important, log seed no matter what
        if (log && log_level & (GATES+MOVE+
            ERRORS)) cout << op << endl;
        // FIX - (-1) should choose random seed
        qs.reseed(str2int(op.args[0]));
    }
    else if( op.opcode == "subset" )
    {
```

```
        vector<string > cgens;
        vector<int> qlist;
        int n = str2int(op.args[1]);
        for( int i = 2; i < 2 + n; i++ ) qlist.
            push_back(qubits[op.args[i]]);
        for( int i = 2+n; i < 2 + 2*n ; i++ )
            cgens.push_back(op.args[i]);
        if(qs.subset(qlist,cgens)) setbit(op.
            args[0],1);
        else setbit(op.args[0],0);
    }
    else if( op.opcode == "subsetket" )
    {
        int i,j,k;
        if (1)
        //if (log_level & GATES)
        {
            vector<string > cgens;
            vector<int> qlist;
            int n = str2int(op.args[0]);
            for( i = 1; i < 1 + n; i++ ) qlist.
                push_back(qubits[op.args[i]]);
            cout << qs.subset_ket(qlist) << endl
                ;
        }
    }
    else if( op.opcode == "logtableau" )
    {
        if (log && log_level & GATES) cout << qs
            .raw() << endl;
    }
    else if( op.opcode == "logket" )
    {
        if (log && log_level & GATES) cout << qs
            .ket() << endl;
    }
    else if( op.opcode == "logstate" )
    {
        if (log && log_level & GATES) cout << qs
            .state() << endl;
    }
    else return Machine::exec(op,log);

    return true;
}

bool CGMachine::hasqubit(string name)
{
    map<string , int >::iterator loc;
    loc = qubits.find(name);
    if( loc != qubits.end() ) return true;
    return false;
}

void CGMachine::setqubit(string name)
{
    qubits[name] = qubitCounter;
    qubitCounter++;
}

int CGMachine::getqubit(string name)
{
    return qubits[name];
}

CGMachine::CGMachine(string fname) : Machine()
{
    qubitCounter = 0;

    map<string ,double> params = readParamFile(
        fname);
    map<string ,double>::iterator sdi;

    for( sdi=params.begin(); sdi != params.end()
        ; sdi++ )
    {
        string s = sdi->first;
        string t = Token(s,".");
        if( t == "times" )
        {
                times[s] = (int)sdi->second;
                cout << "param times." << s << "
                    " << (int)sdi->second <<
                    endl;
        } else
        {
                cerr << "Warning! Cannot
                    understand parameter ";
                cerr << t << "." << s << ".
                    Ignoring." << endl;
        }
    }
}
```

C.2.2 CGMachine.He

```
// CGMachine.hh
// Andrew Cross <awcross@mit.edu>
// $Id: CGMachine.hh 6 2004-07-16 03:48:34Z
    awcross $
// Clifford group machine

#include <cstdlib>

#include "Machine.hh"
#include "QState.hh"

#ifndef __CGMachine__
#define __CGMachine__

class CGMachine : public Machine
{
protected:
    // overridden functions
    bool process(void);
    bool exec(Op& op, bool log);
    // functions
    bool hasqubit(string name);
    void setqubit(string name);
    int getqubit(string name);
    // data
    map<string , int> qubits;
    QState qs, qbak;
    long qubitCounter;
public:
    CGMachine(string fname);
    CGMachine();
    void save(void);
    void restore(void);
};

#endif
```

C.2.3 ITMachine.cc

```
// ITMachine.cxx
// $Id: ITMachine.cc 149 2004-08-17 13:44:11Z
    awcross $

#include "include/ITMachine.hh"
#include "include/globals.hh"
#include "include/loglevels.hh"
using namespace std;

ITMachine::ITMachine()
{
        // Don't really call this - you won't
            have heating!
}

ITMachine::ITMachine(string fname)
{
    map<string ,double> params = readParamFile(
        fname);
    map<string ,double>::iterator sdi;

    for( sdi=params.begin(); sdi != params.end()
        ; sdi++ )
    {
        string s = sdi->first;
        string t = Token(s,".");
        if( t == "times" )
        {
                times[s] = (int)sdi->second;
                if (log_level & STARTUP)
                        cout << "param times."
                            << s << " "
                        << (int)sdi->second
                            << endl;
        } else if( t == "heating" )
        {
                heating_rates[s] = sdi->second;
                if (log_level & STARTUP)
                        cout << "param
                            heating_rates." <<
                            s
```

```
                                    << " " << sdi->
                                        second <<
                                        endl;
        } else
        {
                cerr << "Warning! Cannot
                     understand parameter ";
                cerr << t << "." << s << ".
                     Ignoring." << endl;
        }
    }
}

bool ITMachine::process(void)
{
    long i;
    vector<Op>::iterator j;
    vector<string>::iterator s;

    // First, process "seq" using PCGMachine
    if( !PCGMachine::process() ) return false;

    // PCGMachine's language intersects our
         language but
    // is not a subset of our language. We will
         have to
    // block some instructions now and add the
         new instructions

    // If that didn't fail, go ahead.

#ifdef TURBO
    map<string, int>::iterator msi;
    for( msi=qubits.begin(); msi != qubits.end()
         ; msi++ )
        heat[msi->first] = 0.0;
    return true;
#else // not TURBO
    for( i = 0; i < seq.size(); i++ )
    {
        for( j = seq[i].begin(); j < seq[i].end
             (); j++ )
        {
            // CNOT QUBIT, QUBIT - cnot is not
                 directly a valid operation
            // with trapped ions. Block it for
                 now. In the future we could
            // replace it with hadamards and
                 controlled-Z gates.
            if( j->opcode == "cnot" )
            {
                if (log_level & DEBUG) cerr << "
                     DBG processing cnot" <<
                     endl;
                cerr << "Error! Please use h and
                      cz instead of cnot at "
                      << i << endl;
                return false;
            }
            // COOL QUBIT, QUBIT, TICKS
            // Verify the arguments
            else if( j->opcode == "cool" )
            {
                if (log_level & DEBUG) cerr << "
                     DBG processing cool " << "
                     " << i << endl;
                // FIX check args.size()
                if( !hasqubit(j->args[0]) || !
                     hasqubit(j->args[1])
                              || !isInteger(j
                                  ->args[2]))
                {
                    cerr << "Error! Incorrect
                         arguments for "
                         << "\"cool qubit,qubit,
                              ticks\" at
                              instruction "
                         << i << endl;
                    return false;
                }
                j->nticks = str2int(j->args[2]);
                continue; // don't set the
                     number of ticks again
            }
            // UNKNOWN
            else
            {
                if (log_level & DEBUG) cerr << "
                     DBG Warning! Opcode " << j
                     ->opcode << " unknown to
                     ITMachine" << endl;
```

```
                continue; // don't set the
                     number of clock cycles
            }

            j->nticks = times[j->opcode]; //
                 unreachable for now
        }
    }

    // Build the heat map using the qubit names
         in CGMachine::qubits
    // If CGMachine::qubits changes then this
         code will break.
    // Assume that each qubit has <n> $\approx$
         0 for the center of mass mode.
    map<string, int>::iterator msi;
    for( msi=qubits.begin(); msi != qubits.end()
         ; msi++ )
        heat[msi->first] = 0.0;

    return true;

#endif // TURBO
}

bool ITMachine::exec(Op& op, bool log)
{
    if( op.opcode == "cool" )
    {
        // FIX - this is all messed up
        //if(nnqubits(op.args[0],op.args[1]))
        //{
                double oldheat1; double oldheat2
                    ;
                if (log_level & GATES) {
                        oldheat1 = heat[op.args
                            [0]];
                        oldheat2 = heat[op.args
                            [1]];
                }
                // The qubits first have to be
                     joined to share
                // their CM mode.
                heat[op.args[0]] +=
                     heating_rates["join"];
                heat[op.args[1]] +=
                     heating_rates["join"];
                // Cool both qubits using
                     heating_rates["cool"]
                // as the change in number of
                     quanta (of the center
                // of mass mode) per clock cycle
                     . This should be
                // < 0 if you want cooling.
                op.nticks = str2int(op.args[2]);
                double delta = heating_rates["
                     cool"] * op.nticks;
                heat[op.args[0]] += delta;
                heat[op.args[1]] += delta;
                if(heat[op.args[0]] < 0) heat[op
                     .args[0]] = 0;
                if(heat[op.args[1]] < 0) heat[op
                     .args[1]] = 0;
                // The qubits need to be
                     separated again.
                heat[op.args[0]] +=
                     heating_rates["split"];
                heat[op.args[1]] +=
                     heating_rates["split"];
                // Assume that the cooling laser
                      is applied to
                // the first qubit and causes
                     that qubit's state
                // to be destroyed. Model this
                     as depolarization.
                double p = ((double)rand())/((
                     double)RAND_MAX);
                Op o;
                if( p < 0.3333 )
                        o.opcode = "x";
                else if( p < 0.6666 )
                        o.opcode = "y";
                else
                        o.opcode = "z";
                o.args.push_back(op.args[0]);

                // The log variable will be
                     ignored here
                if (log_level & GATES) {
                        cout << "heat " << op.
                             args[0] << " " <<
                             oldheat1 << " "
```

```
                                                << heat[op.args
                                                    [0]] <<
                                                    endl;
                                cout << "heat " << op.
                                    args[1] << " " <<
                                    oldheat2 << " "
                                        << heat[op.args
                                            [1]] <<
                                            endl;
                        }

                        return CGMachine::exec(o,true);

// FIX - this is broken too
/*              else
                {
                        cerr << "Error! cooling applied
                            to distant qubits.";
                        cerr << "Surely you didn't
                            intend this. Check your
                            code.";
                        cerr << endl;
                        return false;
                }*/
        }
        else if( op.opcode == "move" )
        {
                double oldheat;
                if (log_level & GATES) oldheat = heat[op
                    .args[0]];
                // Heat up qubits as a result of the
                    move
                heat[op.args[0]] += heating_rates["move"
                    ] * str2int(op.args[2]);
                // The log variable will be ignored here
                if (log_level & GATES)
                        cout << "heat " << op.args
                            [0] << " " << oldheat << "
                            "
                                << heat[op.args[0]] <<
                                    endl;
                // Execute the move
                return PCGMachine::exec(op,log);
        }
        else if( op.opcode == "cz" )
        {
                double oldheat1; double oldheat2;
                if (log_level & GATES) {
                        oldheat1 = heat[op.args[0]];
                        oldheat2 = heat[op.args[1]];
                }
                // PCGMachine will check for locality
                    and apply the gate.
                // We need to apply heat for the split
                    and join here.
                heat[op.args[0]] += heating_rates["split
                    "] + heating_rates["join"];
                heat[op.args[1]] += heating_rates["split
                    "] + heating_rates["join"];
                // The log variable will be ignored here
                if (log_level & GATES) {
                        cout << "heat " << op.args
                            [0] << " " << oldheat1 << "
                            "
                                << heat[op.args[0]] <<
                                    endl;
                        cout << "heat " << op.args
                            [1] << " " << oldheat2 << "
                            "
                                << heat[op.args[1]] <<
                                    endl;
                }
                return PCGMachine::exec(op,log);
        }
        else return PCGMachine::exec(op,log);

        return true;
}
```

C.2.4 ITMachine.hh

```
// ITMachine.hh
// Andrew Cross <awcross@mit.edu>
// $Id: ITMachine.hh 6 2004-07-16 03:48:34Z
    awcross $

#ifndef __ITMachine__
#define __ITMachine__

#include <map>
#include <string>

#include "utilities.hh"
#include "PCGMachine.hh"

using namespace std;

class ITMachine : public PCGMachine
{

protected:

    // <n> for the center of mass mode
    // of each qubit
    map<string, double> heat;

    bool process(void);
    bool exec(Op& op, bool log);

public:
    map<string, double> heating_rates;
    ITMachine(string fname);
    ITMachine();
};

#endif
```

C.2.5 Machine.cc

```
// Machine.cxx
// $Id: Machine.cc 158 2004-08-26 22:29:24Z
    setso $
// Computing machine

#include "include/Machine.hh"
#include "include/globals.hh"
#include "include/loglevels.hh"

using namespace std;
Machine::Machine()
{
        skip = false;
}

void Machine::save(void)
{
}

void Machine::layout(string fname)
{

}

void Machine::restore(void)
{
        map<string,int>::iterator i;
        skip = false;
        // reinitialize bits
        for( i=bits.begin(); i != bits.end(); i
            ++ ) i->second = 0;
}

ostream& operator << (ostream& os, Machine& m)
{
    unsigned long iptr = 0;
    os << m.seq.size() << " instruction groups"
        << endl;
    while( iptr < m.seq.size() )
    {
        vector<Op> tmp = m.seq[iptr];    //
            parallel operations in this step
        os << iptr << " : " << tmp.size() << " "
            ;
        for( int i = 0; i < tmp.size(); i++ )
        {
            os << tmp[i].opcode << " (" << tmp[i
                ].nticks << ") ";
            for( int j = 0; j < tmp[i].args.size
                (); j++ )
                os << tmp[i].args[j] << " ";
        }
        os << endl;
        iptr++;
    }
    return os;
}
```

```
// I won't be careful here either, for now.
map<string,double> Machine::readParamFile(string
      filename)
{
        map<string,double> result;
        string s;
        char buffer[500];
        ifstream fin (filename.c_str(), ios::in)
            ;
        // FIX - what if file doesn't exist?

        fin.getline(&buffer[0],499);
        while( !fin.eof() )
        {
                string s(&buffer[0]);
                string t = Token(s,"=");
                Trim(s);
                result[t] = atof(s.c_str());
                fin.getline(&buffer[0],499);
        }
        fin.close();
        return result;
}

Machine::Machine(string fname)
{
    skip = false;

    // Instruction times that are not explicitly
        declared here
    // will be created the first time they are
        requested and
    // will be set to zero.

    map<string,double> params = readParamFile(
        fname);
    map<string,double>::iterator sdi;

    for( sdi=params.begin(); sdi != params.end()
        ; sdi++ )
    {
        string s = sdi->first;
        string t = Token(s,".");
        if( t == "times" )
        {
                times[s] = (int)sdi->second;
                cout << "param times." << s << "
                    " << (int)sdi->second <<
                    endl;
        } else
        {
                cerr << "Warning! Cannot
                    understand parameter ";
                cerr << t << "." << s << ".
                    Ignoring." << endl;
        }
    }
}

// Preprocess the instruction array. Do things
    like set labels,
// initialize information, and set instruction
    times.
// Only quit if you recognize a critical error
    within your jurisdiction.
// Only warn of unknown operations in debug mode
    .
bool Machine::process(void)
{
    long i;
    vector<Op>::iterator j;
    vector<string>::iterator s;

    for( i = 0; i < seq.size(); i++ )
    {
        // Count the number of jumps in this
            block of parallel
        // instructions. There cannot be more
            than one jump.
        // Calls count as jumps.
        int njumps = 0;
        // Count the number of labels in this
            block of parallel
        // instructions. There cannot be more
            than one label.
        int nlabels = 0;
        for( j = seq[i].begin(); j < seq[i].end
            (); j++ )
        {
            // NOP
            if( j->opcode == "nop" )
{
    if (log_level & DEBUG ) cerr <<
        "DBG processing nop " << i
         << endl;
}
// BIT NAME, ..., NAME
// Initialize each bit to zero,
    ignoring duplicates (warn),
// and replace the instruction with
    NOP
else if( j->opcode == "bit" )
{
    if (log_level & DEBUG ) cerr <<
        "DBG processing bit " << i
         << endl;
    for( s = j->args.begin(); s < j
        ->args.end(); s++ )
    {
        if( hasbit(*s) )
            cerr << "Warning: " << *
                s << " redeclared."
                 << endl;
        else
        {
            if (log_level & DEBUG )
                cerr<< "DBG add bit
                 " << *s << " " <<
                i << endl;
            if (log_level & INIT)
                cout << "bit " << *
                s << endl;
            setbit(*s,0);
        }
    }
    j->opcode = "nop";
}
// LABEL NAME
// Add the label and its target.
    Abort if the label exists.
// Replace the instruction with NOP.
else if( j->opcode == "label" )
{
    if (log_level & DEBUG) cerr << "
        DBG processing label " << i
         << endl;
    if( haslabel(j->args[0]) )
    {
        cerr << "Error! Cannot
            continue. Multiple
            labels \""
            << j->args[0] << "\" at
                instruction " <<
                i << endl;
        return false;
    }
    else
    {
        if (log_level & DEBUG)
                cerr << "DBG label "
                    << j->args
                   [0] << " refers
                    to "
                   << i << endl;
        setlabel(j->args[0],i);
    }
    j->opcode = "nop";        //
        replace with NOP
    nlabels++;                //
        increase label count
}
// JUMP NAME
// Do nothing. We cannot know if the
    jump target exists without
// doing another pass.
else if( j->opcode == "jump" )
{
    if (log_level & DEBUG)
            cerr << "DBG processing
                jump " << i << endl
                ;
    njumps++;                 //
        increase jump count
}
// AND C A B (c = a * b)
// Verify that the arguments are of
    the form
// bit,bit|[01],bit|[01].
else if( j->opcode == "and" )
{
    if (log_level & DEBUG)
            cerr << "DBG processing
                and " << i << endl;
```

```
        // FIX check args.size()
        for( s = j->args.begin(); s < j
            ->args.end(); s++ )
        {
            if( !hasbit(*s) )
            {
                if (log_level & INIT)
                    cout << "Adding bit
                    " << *s << endl;
                setbit(*s,0);
            }
        }
        if( !hasbit(j->args[0]) ||
            (j->args[1] != "0" && j->
                args[1] != "1" &&
             !hasbit(j->args[1])) ||
            (j->args[2] != "0" && j->
                args[2] != "1" &&
             !hasbit(j->args[2])))
        {
            cerr << "Error! Incorrect
                arguments for "
                << "\"and bit,bit|[01],
                    bit|[01]\" at
                    instruction "
                << i << endl;
            return false;
        }
    }
    // XOR C A B (c = a + b)
    // Verify that the arguments are of
        the form
    // bit,bit|[01],bit|[01].
    else if( j->opcode == "xor" )
    {
        if (log_level & DEBUG)
            cerr << "DBG processing
                xor " << i << endl;
        // FIX check args.size()
        for( s = j->args.begin(); s < j
            ->args.end(); s++ )
        {
            if( !hasbit(*s) )
            {
                if (log_level & INIT)
                    cout << "Adding bit
                    " << *s << endl;
                setbit(*s,0);
            }
        }
        if( !hasbit(j->args[0]) ||
            (j->args[1] != "0" && j->
                args[1] != "1" &&
             !hasbit(j->args[1])) ||
            (j->args[2] != "0" && j->
                args[2] != "1" &&
             !hasbit(j->args[2])))
        {
            cerr << "Error! Incorrect
                arguments for "
                << "\"xor bit,bit|[01],
                    bit|[01]\" at
                    instruction "
                << i << endl;
            return false;
        }
    }
    // OR C A B (c = a | b)
    // Verify that the arguments are of
        the form
    // bit,bit|[01],bit|[01].
    else if( j->opcode == "or" )
    {
        if (log_level & DEBUG)
            cerr << "DBG processing
                or " << i << endl;
        // FIX check args.size()
        for( s = j->args.begin(); s < j
            ->args.end(); s++ )
        {
            if( !hasbit(*s) )
            {
                if (log_level & INIT)
                    cout << "Adding bit
                    " << *s << endl;
                setbit(*s,0);
            }
        }
        if( !hasbit(j->args[0]) ||
            (j->args[1] != "0" && j->
                args[1] != "1" &&
             !hasbit(j->args[1])) ||
            (j->args[2] != "0" && j->
                args[2] != "1" &&
             !hasbit(j->args[2])))
        {
            cerr << "Error! Incorrect
                arguments for "
                << "\"or bit,bit|[01],
                    bit|[01]\" at
                    instruction "
                << i << endl;
            return false;
        }
    }
    // IF NAME, ..., NAME
    // Abort if any of the bits do not
        exist.
    else if( j->opcode == "if" )
    {
        if (log_level & DEBUG)
            cerr << "DBG processing
                if " << i << endl;

        for( s = j->args.begin(); s < j
            ->args.end(); s++ )
        {
            if( !hasbit(*s) )
            {
                setbit(*s,0);
                //cerr << "Error! Bit
                    " << *s << "
                    undeclared in ";
                //cerr << "\"if\" at
                    instruction " << i
                    << endl;
                //return false;
            }
        }
    }
    // SET C A (c = a)
    // Verify that the arguments are of
        the form
    // bit,bit|[01]
    else if( j->opcode == "set" )
    {
        if (log_level & DEBUG)
            cerr << "DBG processing
                set " << i << endl;
        // FIX check args.size()
        for( s = j->args.begin(); s < j
            ->args.end(); s++ )
        {
            if( !hasbit(*s) )
            {
                if (log_level & INIT)
                    cout << "Adding bit
                    " << *s << endl;
                setbit(*s,0);
            }
        }
        if( !hasbit(j->args[0]) ||
            (j->args[1] != "0" && j->
                args[1] != "1" &&
             !hasbit(j->args[1])))
        {
            cerr << "Error! Incorrect
                arguments for "
                << "\"set bit,bit
                    |[01]\" at
                    instruction "
                << i << endl;
            return false;
        }
    }
    // CALL NAME
    // Do nothing. We cannot know if the
        jump target exists without
    // doing another pass.
    else if( j->opcode == "call" )
    {
        if (log_level & DEBUG)
            cerr << "DBG processing
                call " << i << endl
                ;
        njumps++;                //
            increase jump count
    }
    // RETURN
    // Do nothing. We cannot know if the
        jump target exists without
    // doing another pass.
    else if( j->opcode == "return" )
    {
```

```
                if (log_level & DEBUG)
                        cerr << "DBG processing
                             return " << i <<
                             endl;
                njumps++;                    //
                     increase jump count
            }
            // MSG args
            // Do nothing.
            else if( j->opcode == "msg" )
            {
                if (log_level & DEBUG) cerr << "
                     DBG processing msg " << i
                      << endl;
            }
            // HALT
            // Do nothing.
            else if( j->opcode == "halt" )
            {
                if (log_level & DEBUG)
                        cerr << "DBG processed
                             halt " << i << endl
                             ;
            }
            // UNKNOWN
            else
            {
                if (log_level & DEBUG)
                        cerr << "DBG Warning!
                             Opcode " << j->
                             opcode
                        << " unknown to Machine"
                             << endl;
            }

            // Set the number of clock cycles
                 for the processed instruction
            j->nticks = times[j->opcode];
        }
        // Check that the number of jumps or
             labels in the parallel block
        // isn't greater than one.
        if( njumps > 1 || nlabels > 1 )
        {
            cerr << "Error! Multiple jumps or
                 labels at instruction "
                 << i << endl;
            return false;
        }
    }
    return true;
}

// The log argument doesn't do anything here,
     but does in other
// classes.
bool Machine::exec(Op& op, bool log)
{
    vector<string>::iterator i;

    if( op.opcode == "nop" )
    {
        return true;
    }
    if( op.opcode == "if" )
    {
        skip = false;
        for(i=op.args.begin(); i < op.args.end()
             ; i++ )
            if( getbit(*i) == 0 )
            {
                skip = true;
                break;
            }
        return true;
    }
    else if( op.opcode == "msg" )
    {
        if (log_level & MESSAGE) cout << op <<
             endl;
        return true;
    }
    else if( op.opcode == "jump" ||
             op.opcode == "call" ||
             op.opcode == "return" ||
             op.opcode == "halt" )
    {
        // handle jumps in run()
        // handle halts in run() too
        return true;
    }
    else if( op.opcode == "xor" )
    {
        int f,s;
        if( op.args[1] == "0" ) f = 0;
        else if( op.args[1] == "1" ) f = 1;
        else f = getbit(op.args[1]);
        if( op.args[2] == "0" ) s = 0;
        else if( op.args[2] == "1" ) s = 1;
        else s = getbit(op.args[2]);
        setbit(op.args[0],(f + s)%2);
        return true;
    }
    else if( op.opcode == "and" )
    {
        int f,s;
        if( op.args[1] == "0" ) f = 0;
        else if( op.args[1] == "1" ) f = 1;
        else f = getbit(op.args[1]);
        if( op.args[2] == "0" ) s = 0;
        else if( op.args[2] == "1" ) s = 1;
        else s = getbit(op.args[2]);
        setbit(op.args[0],f * s);
        return true;
    }
    else if( op.opcode == "or" )
    {
        int f,s;
        if( op.args[1] == "0" ) f = 0;
        else if( op.args[1] == "1" ) f = 1;
        else f = getbit(op.args[1]);
        if( op.args[2] == "0" ) s = 0;
        else if( op.args[2] == "1" ) s = 1;
        else s = getbit(op.args[2]);
        setbit(op.args[0],f | s);
        return true;
    }
    else if( op.opcode == "set" )
    {
        int f;
        if( op.args[1] == "0" ) f = 0;
        else if( op.args[1] == "1" ) f = 1;
        else f = getbit(op.args[1]);
        setbit(op.args[0],f);
        return true;
    }
    else return false;
    return true;
}

void Machine::load(string filename)
{
    unsigned long linecount = 0;

        if (log_level & DEBUG)
            cerr << "DBG load " << filename <<
                 endl;

    ifstream infile (filename.c_str());
    if (infile == NULL)
    {
        cerr << "Critical Error! Source file
             does not appear to exist."
             << "\nBailing Out!"
             << endl;
        exit(1);
    }
    while( !infile.eof())
    {
        vector<Op> vop;
        infile >> vop;
        if (log_level & DEBUG) cerr << "DBG got
             : " << vop << endl;
        if( vop.size() > 0 ) seq.push_back(vop);
        linecount++;
    }

        if (log_level & DEBUG)
            cerr << "DBG read " << linecount
                 -1 << " lines" << endl;

    // Process the instruction we've just read
    if( !process() )
    {
        cerr << "Critical Error! Failed to
             process instruction sequence"
             << endl;
        throw Error::Syntax_error("",0);
    }
}

void Machine::run(void)
{
```

```
unsigned int iptr = 0;  // instruction
     pointer
unsigned int ticks = 0; // clock cycles
stack<unsigned long> callstack; // stack of
     pointers for calls

    if (log_level & DEBUG)
        cerr << "DBG run - " << seq.size()
             << " parallel steps" << endl;

// Run instructions in "seq" until the
     instruction pointer points
// one instruction beyond the last.
// Each instruction in "seq" is a list of
     operations to be executed
// "in parallel" in the sense that they
     execute top-down and of all
// the operations that execute, only the
     longest operations adds
// to the cumulative time for the
     instruction.
while( iptr < seq.size() )
{
    vector<Op> tmp = seq[iptr];       //
         parallel operations in this step
    unsigned long next = iptr + 1;  // the
         next instruction location
    int tickCount = 0;               // clock
          increment for this instruction

    if (log_level & DEBUG) {
            cerr << "DBG " << iptr << " : ";
            cerr << tmp.size() << endl;
    }
    // If we have not been instructed to
         skip the next instruction
    if( !skip )
    {
        // Execute the list of operations
        for( int i = 0; i < tmp.size(); i
             ++ )
        {

            // Execute each instruction,
                 throwing a runtime error
            // if the instruction couldn't
                 execute. This could
            // happen if we execute a script
                  meant for another
            // machine, since the
                 preprocessor only warns if
            // instructions don't make sense
                  (and only does so
            // in DEBUG (log_level = DEBUG)
                  mode).
            if( !exec(tmp[i],true) )
            {
                cerr << "Critical Error!
                     Invalid instruction "
                      << i << " " << tmp[i
                           ] << endl;
                throw Error::Exec_error("");
            }
            // Use the longest operation
                 time as the instruction
                 time
            if( tmp[i].nticks > tickCount )
                tickCount = tmp[i].nticks;

            // Special handling: CALL,
                 RETURN, JUMP
            // CALL: push the return target
            if( tmp[i].opcode == "call" )
            {
                if (log_level & DEBUG)
                        cerr << "DBG pushed
                             " << iptr+1
                             << " for call
                                  return"
                                  << endl;
                callstack.push(iptr+1);
            }
            // RETURN: pop off the return
                 target
            if( tmp[i].opcode == "return" )
            {
                next = callstack.top();
                if (log_level & DEBUG)
                     cerr << "DBG return
                          popped " << next
                           << endl;
                callstack.pop();
            }
            // JUMP, CALL: get the jump
                 target
            if( tmp[i].opcode == "jump" ||
                 tmp[i].opcode == "call")
            {
                if( haslabel(tmp[i].args[0])
                     )
                {
                    next = getlabel(tmp[i].
                         args[0]);
                    if (log_level & DEBUG)
                            cerr << "DBG
                                 jump target
                                  \"" << tmp
                                 [i].args[0]
                            << "\" at " <<
                                 next <<
                                 endl;
                }
                else
                {
                    // Ouch, no target!
                    cerr << "Critical Error
                         ! No jump target
                         for jump "
                         << i << endl;
                    throw Error::Exec_error(
                         tmp[i].args[0]);
                }
            }
            // HALT: throw the Crash_error
                 exception, we died on
                 purpose
            if( tmp[i].opcode == "halt" )
            {
                string msg = to_string(iptr)
                     + " " + to_string(
                     ticks);
                throw Error::Crash_error(msg
                     );
            }
            // END OF SPECIAL HANDLING
        } // END OF INSTRUCTION LOOP
    }
    else // We have been instructed to skip
         the next operation
    {
        // The skip will be successful only
             if we have skipped
        // a significant instruction. Make
             sure that the instruction
        // consists of more than just a
             bunch of NOPs.
        for( int i = 0; i < tmp.size(); i
             ++ )
            if( tmp[i].opcode != "nop" )
                skip = false;
            if (log_level & DEBUG)
                cerr << "DBG instruction
                     skipped, skip is now "
                      << skip << endl;
    }

    // The instruction has now been executed
         .
    // Update the instruction pointer and
         the clock.
    iptr = next;
    ticks += tickCount;
    if (log_level & CLOCK)
            if(tickCount) cout << "clock "
                 << (ticks-tickCount)
            << " " << ticks << endl;
    if (log_level & DEBUG)
            cerr << "DBG clock=" << ticks
                 << "; pointer=" << iptr
                 << endl;
    }
}

void Machine::setbit(string name, int val)
{
    if (log_level & INIT) {
            if( hasbit(name) )
                cout << "set " << name << " "
                     << getbit(name)
                         << " " << val << endl;
            else
                cout << "set " << name << " - "
                     << val << endl;
    }
```

```
    bits[name] = val;
}

bool Machine::hasbit(string name)
{
    map<string , int >::iterator loc;
    loc = bits.find(name);
    if( loc != bits.end() ) return true;
    return false;
}

int Machine::getbit(string name)
{
    return bits[name];
}

bool Machine::haslabel(string name)
{
    map<string , unsigned long>::iterator loc;
    loc = labels.find(name);
    if( loc != labels.end() ) return true;
    return false;
}

void Machine::setlabel(string name, unsigned
     long line)
{
    labels[name] = line;
}

unsigned long Machine::getlabel(string name)
{
    return labels[name];
}
```

C.2.6 Machine.hh

```
// Machine.hh
// Andrew Cross <awcross@mit.edu>
// Computing machine
// $Id: Machine.hh 6 2004-07-16 03:48:34Z
     awcross $

#ifndef __Machine__
#define __Machine__

#include <iostream>
#include <fstream>
#include <map>
#include <vector>
#include <string>
#include <stack>

#include "Op.hh"
#include "utilities.hh"

using namespace std;

class Machine
{
protected:
    vector<vector<Op> > seq;              //
         instruction sequence
    virtual bool exec(Op& op, bool log);
              // execute an operation
    virtual bool process(void);        // process
         "seq"
    // private functions
    void setbit(string name, int val); // set a
         bit
    bool hasbit(string name);           // check
         if bit exists
    int getbit(string name);            // get a
         bit
    bool haslabel(string name);         // check
         if label exists
    void setlabel(string name, unsigned long
         line);  // set a label
    unsigned long getlabel(string name);
                   // get a label
    // private state
    bool skip;                          // flag:
         True to skip next instruction
    map<string , unsigned long> labels; // jump
         targets
    map<string , int> bits;             // bit
         names and values
    map<string ,double> readParamFile(string
         filename);
public:
    map<string , int> times;           // default
         operation times
    Machine(string fname);
    Machine();
    friend ostream& operator << (ostream& os,
         Machine& m);
    virtual void layout(string filename);
    void load(string filename);
    void run(void);
    virtual void save(void);     // save before
         run()
    virtual void restore(void); // restore after
         run()
};

ostream& operator << (ostream& os, Machine& m);

namespace Error
{

    struct Syntax_error
    {
        string msg;
        unsigned long lineno;
        Syntax_error(string imsg, unsigned long
             l)
            {
                msg = imsg; lineno = l;
            }
    };

    struct Exec_error
    {
        string msg;
        Exec_error(string m) {msg = m;}
    };

    struct Crash_error
    {
        string msg;
        Crash_error(string m) {msg = m;}
    };
}
#endif
```

C.2.7 NoisyCGMachine.cc

```
// NoisyCGMachine.cxx
// $Id: NoisyCGMachine.cc
     202 2004-10-02 00:00:47Z setso $
// Noisy Clifford group machine

#include "include/NoisyCGMachine.hh"
#include "include/globals.hh"
#include "include/loglevels.hh"

NoisyCGMachine::NoisyCGMachine(string fname)
{
    map<string ,double> params = readParamFile(
         fname);
    map<string ,double>::iterator sdi;

    noisyFlag = true;
    noiseType = DEPOL;           // default noise
         type = depolarization

    _pf1 = -1.0; _pf2 = -1.0; _pfm = -1.0;

    for( sdi=params.begin(); sdi != params.end()
         ; sdi++ )
    {
        string s = sdi->first;
        string t = Token(s,".");
        if( t == "times" )
        {
                times[s] = (int)sdi->second;
                cout << "param times." << s << "
                     " << (int)sdi->second <<
                     endl;
        } else if( t == "failures" )
        {
                if( s == "pf1" )
                {
                        cout << "param pf1 " <<
                             sdi->second << endl
                             ;
                        _pf1 = sdi->second;
```

```
                }
                else if( s == "pf2")
                {
                        cout << "param pf2 " <<
                             sdi->second << endl
                             ;
                        _pf2 = sdi->second;
                }
                else if( s == "pfm")
                {
                        cout << "param pfm " <<
                             sdi->second << endl
                             ;
                        _pfm = sdi->second;
                }
                else
                {
                        cerr << "Warning! Cannot
                              understand
                             parameter ";
                        cerr << t << "." << s
                              << ". Ignoring."
                              << endl;
                }
        }
        else
        {
                cerr << "Warning! Cannot
                     understand parameter ";
                cerr << t << "." << s << ".
                     Ignoring." << endl;
        }
    }
    if( _pf1 == -1.0 || _pf2 == -1.0 || _pfm
         == -1.0 )
    {
                cerr << "Warning! Parameters
                     failures.pf1, failures.pf2
                     , ";
                cerr << "and failures.pfm are
                     mandatory." << endl;
                exit(1);
    }
}

bool NoisyCGMachine::process(void)
{

    long i;
    vector<Op>::iterator j;
    vector<string>::iterator s;

    // First process using CGMachine
    if( !CGMachine::process() ) return false;

    for( i = 0; i < seq.size(); i++ )
    {
        for( j = seq[i].begin(); j < seq[i].end
             (); j++ )
        {
            // NOISE [type]
            if( j->opcode == "noise" )
            {
                if( log_level & DEBUG )
                    cerr << "DBG processing
                         noise" << endl;

                if( j->args[0] != "on" &&
                    j->args[0] != "off" &&
                    j->args[0] != "bitflip" &&
                    j->args[0] != "
                         bitflip_nopropagate" &&
                    j->args[0] != "depolarize"
                         &&
                    j->args[0] != "phaseflip" )
                {
                    cerr << "Error! Incorrect
                         arguments for "
                         << "\"noise [on|off|bitflip
                              |phaseflip|depolarize
                              |bitflip_nopropagate
                              ]\" at instruction
                              "
                         << i << ": {noise "<< j
                              ->args[0] << "}"
                         << endl;
                    return false;
                }
            }
            // UNKNOWN
            else
```

```
            {
                if( log_level & DEBUG )
                    cerr << "DBG Warning! opcode
                         " << j->opcode
                     << " unknown to
                         NoisyCGMachine" <<
                         endl;
            }
        }
    }
    return true;
}

bool NoisyCGMachine::exec(Op& op, bool log)
{
    vector<Op> fop;
    Op tmp;
    vector<Op>::iterator fit;

    if( op.opcode == "noise" )
    {
      noisyFlag = false;                    //
           default: turn off noise
      if( op.args[0] == "on" ){             // turn
           on noise - default type
        noisyFlag = true;
      }
      if (op.args[0] == "bitflip"){        // turn
           on bitflip noise
        noisyFlag = true;
        noiseType = BITFLIP;                //
             bitflip noise
      }
      if (op.args[0] == "bitflip_nopropagate"){
           // turn on bitflip noise
        noisyFlag = true;
        noiseType = BITFLIP_NOPROPAGATE;
             // bitflip noise
      }
      if (op.args[0] == "phaseflip"){      // turn
           on bitflip noise
        noisyFlag = true;
        noiseType = PHASEFLIP;  // phaseflip
             noise
      }
      return true;
    }

    if( noisyFlag )
    {
      if(noiseType==DEPOL){        //
           depolarization noise
        if( op.opcode == "x" || op.opcode == "y"
             || op.opcode == "z" ||
            op.opcode == "h" || op.opcode == "s"
              )
        {
            tmp = depolarize(op.args[0], _pf1);
            if(tmp.opcode != "nop") fop.
                 push_back(tmp);
        }
        else if( op.opcode == "cnot" || op.
             opcode == "cz" )
            fop = depolarize(op.args[0], op.args
                 [1], _pf2);
        else if( op.opcode == "measure" )
        {
            tmp = depolarize(op.args[1], _pfm);
            if(tmp.opcode != "nop") fop.
                 push_back(tmp);
        }
      }
      if(noiseType==BITFLIP){
        if( op.opcode == "x" || op.opcode == "y"
             || op.opcode == "z" ||
            op.opcode == "h" || op.opcode == "s"
              )
        {
            tmp = bitflip(op.args[0], _pf1);
            if(tmp.opcode != "nop") fop.
                 push_back(tmp);
        }
        else if( op.opcode == "cnot" || op.
             opcode == "cz" )
            fop = bitflip(op.args[0], op.args
                 [1], _pf2);
        else if( op.opcode == "measure" )
        {
            tmp = bitflip(op.args[1], _pfm);
            if(tmp.opcode != "nop") fop.
                 push_back(tmp);
        }
```

```
        }
        if(noiseType==BITFLIP_NOPROPAGATE){
          if( op.opcode == "x" || op.opcode == "y"
                || op.opcode == "z" ||
              op.opcode == "h" || op.opcode == "s"
                )
          {
              tmp = bitflip_nopropagate(op.args
                    [0], _pf1);
              if(tmp.opcode != "nop") fop.
                    push_back(tmp);
          }
          else if( op.opcode == "cnot" || op.
               opcode == "cz" )
              fop = bitflip_nopropagate(op.args
                    [0], op.args[1], _pf2);
          else if( op.opcode == "measure" )
          {
              tmp = bitflip_nopropagate(op.args
                    [1], _pfm);
              if(tmp.opcode != "nop") fop.
                    push_back(tmp);
          }
        }
      }

      if(log)
      {
          if(!fop.empty())
          {
                  if( log_level & ERRORS )
                  {
                          cout << "failure " << op
                                << endl;
                          for( fit = fop.begin();
                                fit != fop.end();
                                fit++ )
                                cout << "error "
                                      << *fit
                                      << endl;
                  }
          }
          else
          {
                  if( log_level & GATES )
                  {
                          if( op.opcode == "x" ||
                                op.opcode == "y" ||
                              op.opcode == "z" ||
                                    op.opcode == "h
                                    " ||
                              op.opcode == "s" ||
                                    op.opcode == "
                                    cnot" ||
                              op.opcode == "
                                    measure" || op.
                                    opcode == "cz"
                                    )
                                cout << op <<
                                      endl;
                  }
                  if( op.opcode == "subset" && (
                        log_level & ERRORS) ) cout
                        << op << endl;
                  if( op.opcode == "noise" && (
                        log_level & ERRORS) ) cout
                        << op << endl;
          }
      }

      fop.push_back(op);
      for( fit = fop.begin(); fit != fop.end();
            fit++ )
          if( !CGMachine::exec(*fit, false) )
                return false;
      return true;
}
```

C.2.8 NoisyCGMachine.hh

```
// NoisyCGMachine.hh
// Andrew Cross <awcross@mit.edu>
// $Id: NoisyCGMachine.hh
//     128 2004-08-04 18:08:21Z setso $
// Noisy Clifford group machine

#include "CGMachine.hh"
#include "noise.hh"

#ifndef __NoisyCGMachine__
#define __NoisyCGMachine__

class NoisyCGMachine : public CGMachine
{
protected:
      bool noisyFlag;
      enum { DEPOL, BITFLIP, BITFLIP_NOPROPAGATE,
            PHASEFLIP } noiseType;
      double _pf1;
      double _pf2;
      double _pfm;

      bool process(void);
      bool exec(Op& op, bool log);

public:
      NoisyCGMachine(string fname);
};

#endif
```

C.2.9 NoisyITMachine.cc

```
// NoisyITMachine.cxx
// $Id: NoisyITMachine.cc
//     149 2004-08-17 13:44:11Z awcross $

#include "include/NoisyITMachine.hh"
#include "include/globals.hh"
#include "include/loglevels.hh"

// When calculating errors due to scattering,
//   the simulator will be blind
// to qubits greater than this distance from the
//     measured qubit.
#define MEASURE_CUTOFF 5

using namespace std;

NoisyITMachine::NoisyITMachine(string fname)
{
      map<string,double> params = readParamFile(
            fname);
      map<string,double>::iterator sdi;

      // noisy by default
      noisyFlag = true;
      noiseType = DEPOL;              // default noise
            type = depolarization

      _pf1 = -1.0; _pf2 = -1.0; _pfm = -1.0; _pft
            = -1.0; _pfs = -1.0;

      for( sdi=params.begin(); sdi != params.end()
            ; sdi++ )
      {
          string s = sdi->first;
          string t = Token(s,".");
          if( t == "times" )
          {
                  times[s] = (int)sdi->second;
                  if (log_level & 1)
                          cout << "param times."
                                << s << " "
                                << (int)sdi->second
                                      << endl;
          } else if( t == "failures" )
          {
                  if( s == "pf1" )
                  {
                          if (log_level & 1)
                                  cout << "param
                                        pf1 " <<
                                        sdi->second
                                         << endl;
                          _pf1 = sdi->second;
                  }
                  else if( s == "pf2")
                  {
                          if (log_level & 1)
                                  cout << "param
                                        pf2 " <<
                                        sdi->second
                                         << endl;
                          _pf2 = sdi->second;
                  }
```

```
                else if( s == "pfm")
                {
                        if (log_level & 1)
                                cout << "param
                                      pfm " <<
                                      sdi->second
                                      << endl;
                        _pfm = sdi->second;
                }
                else if( s == "pft")
                {
                        if (log_level & 1)
                                cout << "param
                                      pft " <<
                                      sdi->second
                                      << endl;
                        _pft = sdi->second;
                }
                else if( s == "pfs")
                {
                        if (log_level & 1)
                                cout << "param
                                      pfs " <<
                                      sdi->second
                                      << endl;
                        _pfs = sdi->second;
                }
                else
                {
                        cerr << "Warning! Cannot
                              understand
                             parameter ";
                        cerr << t << "." << s
                             << ". Ignoring."
                             << endl;
                }
        }
        else if( t == "heating" )
        {
                heating_rates[s] = sdi->second;
                if (log_level & 1)
                        cout << "param
                             heating_rates." <<
                             s << " "
                             << sdi->second <<
                                  endl;
        } else
        {
                cerr << "Warning! Cannot
                     understand parameter ";
                cerr << t << "." << s << ".
                     Ignoring." << endl;
        }
    }

    if( _pf1 == -1.0 || _pf2 == -1.0 || _pfm
         == -1.0
                    || _pft == -1.0 || _pfs
                         == -1.0)
    {
                cerr << "Warning! Parameters
                     failures.pf1, failures.pf2
                     , ";
                cerr << "and failures.pfm are
                     mandatory." << endl;
                exit(1);
    }

}

bool NoisyITMachine::process(void)
{


    long i;
    vector<Op>::iterator j;
    vector<string>::iterator s;

    // first process with ITMachine
    if( !ITMachine::process() ) return false;
#ifdef TURBO
    return true;
#else // not TURBO
    for( i = 0; i < seq.size(); i++ )
    {
        for( j = seq[i].begin(); j < seq[i].end
             (); j++ )
        {
            // NOISE [type]
            if( j->opcode == "noise" )
            {
                if (log_level & DEBUG )
                        cerr << "DBG processing
                             noise" << endl;
                if( j->args[0] != "on" &&
                    j->args[0] != "off" &&
                    j->args[0] != "bitflip" &&
                    j->args[0] != "
                         bitflip_nopropagate" &&
                    j->args[0] != "depolarize"
                         &&
                    j->args[0] != "phaseflip" )
                {
                    cerr << "Error! Incorrect
                         arguments for "
                         << "\"noise [type]\" at
                              instruction "
                         << i << ": {noise "<< j
                              ->args[0] << "}"
                         << endl;
                    return false;
                }
            }
            // UNKNOWN
            else
            {
                if (log_level & DEBUG)
                        cerr << "DBG Warning!
                             opcode " << j->
                             opcode
                              << " unknown to
                                   NoisyITMachine
                                   " << endl;
            }
        }
    }
    return true;
#endif // TURBO
}

bool NoisyITMachine::exec(Op& op, bool log)
{
    Op tmp;
    vector<Op> fop;
    vector<Op>::iterator fit;

    if( op.opcode == "noise" )
    {
      noisyFlag = false;                    //
           default: turn off noise
      if( op.args[0] == "on" ){             // turn
           on noise - default type
        noisyFlag = true;
      }
      if (op.args[0] == "bitflip"){         // turn
           on bitflip noise
        noisyFlag = true;
        noiseType = BITFLIP;                //
             bitflip noise
      }
      if (op.args[0] == "bitflip_nopropagate"){
           // turn on bitflip noise
        noisyFlag = true;
        noiseType = BITFLIP_NOPROPAGATE;
             // bitflip noise
      }
      if (op.args[0] == "phaseflip"){    // turn
           on bitflip noise
        noisyFlag = true;
        noiseType = PHASEFLIP;   // phaseflip
             noise
      }
      return true;
    }

    if( noisyFlag )
    {
      if(noiseType==DEPOL){        //
           depolarization noise
        if( op.opcode == "move" )
        {
            double pf = 1.0 - exp(-(double)op.
                 nticks * _pft);
                 if (log_level & DEBUG)
                      cerr << "DBG moving " << op.
                           nticks
                           << " gives pf " << pf
                                << endl;
            tmp = depolarize(op.args[0], pf);
            if(tmp.opcode != "nop") fop.
                 push_back(tmp);
        }
```

```
        else if( op.opcode == "x" || op.opcode
              == "y" || op.opcode == "z" ||
                   op.opcode == "h" || op.opcode
                          == "s" )
        {
             tmp = depolarize(op.args[0], _pf1);
             if(tmp.opcode != "nop") fop.
                  push_back(tmp);
        }
        else if( op.opcode == "cnot" || op.
              opcode == "cz" )
             fop = depolarize(op.args[0], op.args
                  [1], _pf2);
        else if( op.opcode == "measure" )
        {
             vector<q_hentry *> qclose;
             int dist;
             int myloc = P.getloc(op.args[1]);
             tmp = depolarize(op.args[1], _pfm);
             if( tmp.opcode != "nop" ) fop.
                  push_back(tmp);
             if (log_level & DEBUG)
                  cerr << "DBG measured qubit
                       location " << myloc << endl
                       ;
             qclose = P.get_all(op.args[1],
                  MEASURE_CUTOFF);
             for( int i=0; i < qclose.size(); i
                  ++ )
             {
                  dist = P.distance(qclose[i]->pos
                       ->key, myloc);
                  if (log_level & DEBUG) {
                            cerr << "DBG scatter on
                                 " << qclose[i]->key
                                  << " at ";
                            cerr << qclose[i]->pos->
                                 key << endl;
                            cerr << "DBG distance to
                                  measured qubit = "
                                 << dist << endl;
                            cerr << "DBG
                                 corresponding pf =
                                 "
                                  << _pfs / (double)(
                                       dist*dist) <<
                                       endl;
                  }
                  // We will assume this relation
                       for failure probability
                  // due to scattering:
                  //                        p_f(d)
                       = pfs / d^2
                  tmp = depolarize(string(qclose[i
                       ]->key),
                                      _pfs / (double)
                                           (dist *
                                           dist));
                  if( tmp.opcode != "nop" ) fop.
                       push_back(tmp);
             }
        }
   }
   if(noiseType==BITFLIP){    // bitflip noise
        if( op.opcode == "move" )
        {
             double pf = 1.0 - exp(-(double)op.
                  nticks * _pft);
                  if (log_level & DEBUG)
                       cerr << "DBG moving " << op.
                            nticks
                            << " gives pf " << pf
                                 << endl;
             tmp = bitflip(op.args[0], pf);
             if(tmp.opcode != "nop") fop.
                  push_back(tmp);
        }
        else if( op.opcode == "x" || op.opcode
              == "y" || op.opcode == "z" ||
                   op.opcode == "h" || op.opcode
                          == "s" )
        {
             tmp = bitflip(op.args[0], _pf1);
             if(tmp.opcode != "nop") fop.
                  push_back(tmp);
        }
        else if( op.opcode == "cnot" || op.
             opcode == "cz" )
             fop = bitflip(op.args[0], op.args
                  [1], _pf2);
        else if( op.opcode == "measure" )
        {
             vector<q_hentry *> qclose;
             int dist;
             int myloc = P.getloc(op.args[1]);
             tmp = bitflip(op.args[1], _pfm);
             if( tmp.opcode != "nop" ) fop.
                  push_back(tmp);
             if (log_level & DEBUG)
                  cerr << "DBG measured qubit
                       location " << myloc << endl
                       ;
             qclose = P.get_all(op.args[1],
                  MEASURE_CUTOFF);
             for( int i=0; i < qclose.size(); i
                  ++ )
             {
                  dist = P.distance(qclose[i]->pos
                       ->key, myloc);
                  if (log_level & DEBUG) {
                            cerr << "DBG scatter on
                                 " << qclose[i]->key
                                  << " at ";
                            cerr << qclose[i]->pos->
                                 key << endl;
                            cerr << "DBG distance to
                                  measured qubit = "
                                 << dist << endl;
                            cerr << "DBG
                                 corresponding pf =
                                 "
                                  << _pfs / (double)(
                                       dist*dist) <<
                                       endl;
                  }
                  // We will assume this relation
                       for failure probability
                  // due to scattering:
                  //                        p_f(d)
                       = pfs / d^2
                  tmp = bitflip(string(qclose[i]->
                       key),
                                      _pfs / (double)
                                           (dist *
                                           dist));
                  if( tmp.opcode != "nop" ) fop.
                       push_back(tmp);
             }
        }
   }
   if(noiseType==PHASEFLIP){             //
        phaseflip noise
        if( op.opcode == "move" )
        {
             double pf = 1.0 - exp(-(double)op.
                  nticks * _pft);
                  if (log_level & DEBUG)
                       cerr << "DBG moving " << op.
                            nticks
                            << " gives pf " << pf
                                 << endl;
             tmp = phaseflip(op.args[0], pf);
             if(tmp.opcode != "nop") fop.
                  push_back(tmp);
        }
        else if( op.opcode == "x" || op.opcode
              == "y" || op.opcode == "z" ||
                   op.opcode == "h" || op.opcode
                          == "s" )
        {
             tmp = phaseflip(op.args[0], _pf1);
             if(tmp.opcode != "nop") fop.
                  push_back(tmp);
        }
        else if( op.opcode == "cnot" || op.
             opcode == "cz" )
             fop = phaseflip(op.args[0], op.args
                  [1], _pf2);
        else if( op.opcode == "measure" )
        {
             vector<q_hentry *> qclose;
             int dist;
             int myloc = P.getloc(op.args[1]);
             tmp = phaseflip(op.args[1], _pfm);
             if( tmp.opcode != "nop" ) fop.
                  push_back(tmp);
             if (log_level & DEBUG)
                  cerr << "DBG measured qubit
                       location " << myloc << endl
                       ;
             qclose = P.get_all(op.args[1],
                  MEASURE_CUTOFF);
             for( int i=0; i < qclose.size(); i
                  ++ )
```

```
			{
				dist = P.distance(qclose[i]->pos
					->key,myloc);
				if (log_level & DEBUG) {
					cerr << "DBG scatter on
						" << qclose[i]->key
						<< " at ";
					cerr << qclose[i]->pos->
						key << endl;
					cerr << "DBG distance to
						measured qubit = "
						<< dist << endl;
					cerr << "DBG
						corresponding pf =
						"
						<< _pfs / (double)(
							dist*dist) <<
							endl;
				}
				// We will assume this relation
					for failure probability
				// due to scattering:
				//                      p_f(d)
					= pfs / d^2
				tmp = phaseflip(string(qclose[i
					]->key),
						_pfs / (double)
							(dist *
							dist));
				if( tmp.opcode != "nop" ) fop.
					push_back(tmp);
			}
		}
	}
}
	if(!fop.empty())
	{
		if( log_level & ERRORS )
		{
			cout << "failure " << op
				<< endl;
			for( fit = fop.begin();
				fit != fop.end();
				fit++ )
				cout << "error "
					<< *fit
					<< endl;
		}
	}
	else
	{
		if( log_level & GATES )
		if( op.opcode == "x" || op.
			opcode == "y" ||
			op.opcode == "z" || op.
				opcode == "h" ||
			op.opcode == "s" || op.
				opcode == "cnot" ||
			op.opcode == "measure" || op
				.opcode == "cz" )
			cout << op << endl;

		if( log_level & MOVE )
		if( op.opcode == "move" || op.
			opcode == "cool" )
			cout << op << endl;

		if (log_level & ERRORS )
		if( op.opcode == "subset" || op.
			opcode == "noise")
			cout << op << endl;
	}
	fop.push_back(op);
	for( fit = fop.begin(); fit != fop.end();
		fit++ )
		if( !ITMachine::exec(*fit,false) )
			return false;
	return true;
}
```

C.2.10 NoisyITMachine.hh

```
// NoisyITMachine.hh
// Andrew Cross <awcross@mit.edu>
// $Id: NoisyITMachine.hh
	128 2004-08-04 18:08:21Z setso $

#ifndef __NoisyITMachine__
#define __NoisyITMachine__

#include <map>
#include <string>

#include "ITMachine.hh"
#include "noise.hh"

using namespace std;

class NoisyITMachine : public ITMachine
{

protected:

	bool noisyFlag;
	enum { DEPOL, BITFLIP, BITFLIP_NOPROPAGATE,
		PHASEFLIP } noiseType;
	bool process(void);
	bool exec(Op& op, bool log);
	double _pf1;
	double _pf2;
	double _pfm;
	double _pft;
	double _pfs;

public:
	NoisyITMachine(string fname);
};

#endif
```

C.2.11 NoisyPCGMachine.cc

```
// NoisyPCGMachine.cxx
// $Id: NoisyPCGMachine.cc
	202 2004-10-02 00:00:47Z setso $
// Noisy Planar Clifford group machine

#include "include/NoisyPCGMachine.hh"
#include "include/globals.hh"
#include "include/loglevels.hh"

using namespace std;

NoisyPCGMachine::NoisyPCGMachine(string fname)
{
	map<string,double> params = readParamFile(
		fname);
	map<string,double>::iterator sdi;

	noisyFlag = true;
	noiseType = DEPOL;		// default noise
		type = depolarization
	_pf1 = -1.0; _pf2 = -1.0; _pfm = -1.0; _pft
		= -1.0;

	for( sdi=params.begin(); sdi != params.end()
		; sdi++ )
	{
		string s = sdi->first;
		string t = Token(s,".");
		if( t == "times" )
		{
			times[s] = (int)sdi->second;
			if (log_level & 1)
				cout << "param times."
					<< s << " "
					<< (int)sdi->second
						<< endl;
		} else if( t == "failures" )
		{
			if( s == "pf1" )
			{
				if (log_level & 1)
					cout << "param
						pf1 " <<
						sdi->second
						<< endl;
				_pf1 = sdi->second;
			}
			else if( s == "pf2")
			{
				if (log_level & 1)
					cout << "param
						pf2 " <<
						sdi->second
						<< endl;
```

```
                        _pf2 = sdi->second;
                }
                else if( s == "pfm")
                {
                        if (log_level & 1)
                                cout << "param
                                        pfm " <<
                                        sdi->second
                                        << endl;
                        _pfm = sdi->second;
                }
                else if( s == "pft")
                {
                        if (log_level & 1)
                                cout << "param
                                        pft " <<
                                        sdi->second
                                        << endl;
                        _pft = sdi->second;
                }
                else
                {
                        cerr << "Warning! Cannot
                              understand
                              parameter ";
                        cerr << t << "." << s
                              << ". Ignoring."
                              << endl;
                }
        }
        else
        {
                cerr << "Warning! Cannot
                      understand parameter ";
                cerr << t << "." << s << ".
                      Ignoring." << endl;
        }
    }

    if( _pf1 == -1.0 || _pf2 == -1.0 || _pfm
         == -1.0 || _pft == -1.0)
    {
                cerr << "Warning! Parameters
                      failures.pf1, failures.pf2
                      , ";
                cerr << "and failures.pfm are
                      mandatory." << endl;
                exit(1);
    }
}

bool NoisyPCGMachine::process(void)
{
    long i;
    vector<Op>::iterator j;
    vector<string>::iterator s;
    // first process with PCGMachine
    if( !PCGMachine::process() ) return false;

#ifdef TURBO
    return true;

#else // Not TURBO
    for( i = 0; i < seq.size(); i++ )
    {
        for( j = seq[i].begin(); j < seq[i].end
             (); j++ )
        {
            // NOISE [type]
            if( j->opcode == "noise" )
            {
                if (log_level & DEBUG)
                        cerr << "DBG processing
                              noise" << endl;
                if( j->args[0] != "on" &&
                     j->args[0] != "off" &&
                     j->args[0] != "bitflip" &&
                     j->args[0] != "
                          bitflip_nopropagate" &&
                     j->args[0] != "depolarize"
                          &&
                     j->args[0] != "phaseflip" )
                {
                    cerr << "Error! Incorrect
                          arguments for "
                          << "\"noise [type]\" at
                                instruction "
                          << i << ": {noise "<< j
                                ->args[0] << "}"
                          << endl;
                    return false;
                }
            }
            // UNKNOWN
            else
            {
                if (log_level & DEBUG)
                cerr << "DBG Warning! opcode "
                      << j->opcode
                      << " unknown to
                            NoisyPCGMachine" <<
                            endl;
            }
        }
    }
    return true;
#endif // TURBO
}

bool NoisyPCGMachine::exec(Op& op, bool log)
{
    vector<Op> fop;
    Op tmp;
    vector<Op>::iterator fit;

    if( op.opcode == "noise" )
    {
      noisyFlag = false;                //
           default: turn off noise
      if( op.args[0] == "on" ){         // turn
           on noise - default type
        noisyFlag = true;
      }
      if (op.args[0] == "bitflip"){     // turn
           on bitflip noise
        noisyFlag = true;
        noiseType = BITFLIP;            //
             bitflip noise
      }
      if (op.args[0] == "bitflip_nopropagate"){
           // turn on bitflip noise
        noisyFlag = true;
        noiseType = BITFLIP_NOPROPAGATE;
             // bitflip noise
      }
      if (op.args[0] == "phaseflip"){   // turn
           on bitflip noise
        noisyFlag = true;
        noiseType = PHASEFLIP;  // phaseflip
             noise
      }
      return true;
    }

    if( noisyFlag )
    {
      if(noiseType==DEPOL){       //
           depolarization noise
        if( op.opcode == "move" )
        {
            double pf = 1.0 - exp(-(double)op.
                 nticks * _pft);
                if (log_level & DEBUG)
                    cerr << "DBG moving " << op.
                         nticks
                         << " gives pf " << pf
                              << endl;
            tmp = depolarize(op.args[0], pf);
            if(tmp.opcode != "nop") fop.
                 push_back(tmp);
        }
        else if( op.opcode == "x" || op.opcode
             == "y" || op.opcode == "z" ||
                op.opcode == "h" || op.opcode
                     == "s" )
        {
            tmp = depolarize(op.args[0], _pf1);
            if(tmp.opcode != "nop") fop.
                 push_back(tmp);
        }
        else if( op.opcode == "cnot" || op.
             opcode == "cz" )
            fop = depolarize(op.args[0], op.args
                 [1], _pf2);
        else if( op.opcode == "measure" )
        {
            tmp = depolarize(op.args[1], _pfm);
            if(tmp.opcode != "nop") fop.
                 push_back(tmp);
        }
      }
      if(noiseType==BITFLIP){
        if( op.opcode == "move" )
```

```
        {
            double pf = 1.0 - exp(-(double)op.
                nticks * _pft);
                if (log_level & DEBUG)
                    cerr << "DBG moving " << op.
                        nticks
                        << " gives pf " << pf
                            << endl;
            tmp = bitflip(op.args[0], pf);
            if(tmp.opcode != "nop") fop.
                push_back(tmp);
        }
        else if( op.opcode == "x" || op.opcode
            == "y" || op.opcode == "z" ||
                op.opcode == "h" || op.opcode
                    == "s" )
        {
            tmp = bitflip(op.args[0], _pf1);
            if(tmp.opcode != "nop") fop.
                push_back(tmp);
        }
        else if( op.opcode == "cnot" || op.
            opcode == "cz" )
            fop = bitflip(op.args[0], op.args
                [1], _pf2);
        else if( op.opcode == "measure" )
        {
            tmp = bitflip(op.args[1], _pfm);
            if(tmp.opcode != "nop") fop.
                push_back(tmp);
        }
    }
    if(noiseType==PHASEFLIP){
        if( op.opcode == "move" )
        {
            double pf = 1.0 - exp(-(double)op.
                nticks * _pft);
                if (log_level & DEBUG)
                    cerr << "DBG moving " << op.
                        nticks
                        << " gives pf " << pf
                            << endl;
            tmp = phaseflip(op.args[0], pf);
            if(tmp.opcode != "nop") fop.
                push_back(tmp);
        }
        else if( op.opcode == "x" || op.opcode
            == "y" || op.opcode == "z" ||
                op.opcode == "h" || op.opcode
                    == "s" )
        {
            tmp = phaseflip(op.args[0], _pf1);
            if(tmp.opcode != "nop") fop.
                push_back(tmp);
        }
        else if( op.opcode == "cnot" || op.
            opcode == "cz" )
            fop = phaseflip(op.args[0], op.args
                [1], _pf2);
        else if( op.opcode == "measure" )
        {
            tmp = phaseflip(op.args[1], _pfm);
            if(tmp.opcode != "nop") fop.
                push_back(tmp);
        }
    }
}

        if(!fop.empty())
        {
                if( log_level & ERRORS )
                {
                        cout << "failure " << op
                            << endl;
                        for( fit = fop.begin();
                            fit != fop.end();
                            fit++ )
                                cout << "error "
                                    << *fit
                                    << endl;
                }
        }
        else
        {
                if( log_level & GATES )
                if( op.opcode == "x" || op.
                    opcode == "y" ||
                    op.opcode == "z" || op.
                        opcode == "h" ||
                    op.opcode == "s" || op.
                        opcode == "cnot" ||
                        op.opcode == "measure" || op
                            .opcode == "cz" )
                                cout << op << endl;

                if( log_level & MOVE )
                if( op.opcode == "move" )
                                cout << op << endl;

                if( log_level & ERRORS )
                if( op.opcode == "subset" || op.
                    opcode == "noise")
                                cout << op << endl;
        }
    fop.push_back(op);
    for( fit = fop.begin(); fit != fop.end();
        fit++ )
        if( !PCGMachine::exec(*fit, false) )
            return false;
    return true;
}
```

C.2.12 NoisyPCGMachine.hh

```
// NoisyPCGMachine.hh
// Andrew Cross <awcross@mit.edu>
// $Id: NoisyPCGMachine.hh
    128 2004-08-04 18:08:21Z setso $
// Noisy Planar Clifford group machine

#ifndef __NoisyPCGMachine__
#define __NoisyPCGMachine__

#include <math.h>
#include "PCGMachine.hh"
#include "noise.hh"

class NoisyPCGMachine : public PCGMachine
{
protected:

    bool noisyFlag;
    enum { DEPOL, BITFLIP, BITFLIP_NOPROPAGATE,
        PHASEFLIP } noiseType;
    bool process(void);
    bool exec(Op& op, bool log);
    double _pf1;
    double _pf2;
    double _pfm;
    double _pft;

public:
    NoisyPCGMachine(string fname);
};

#endif
```

C.2.13 Op.cc

```
// Op.cxx
// $Id: Op.cc 84 2004-07-20 16:29:33Z awcross $
// Simple operation

#include "include/Op.hh"

using namespace std;

Op::Op()
{
        opcode = "nop";
        nticks = 0;
}

Op::Op(string o)
{
        opcode = o;
        nticks = 0;
}

ostream& operator << (ostream& os, Op& op)
{
        vector<string >::iterator i;
        os << op.opcode << " ";
        for(i = op.args.begin(); i < op.args.end
            (); i++ )
                os << *i << " ";
```

```
        return os;
}

ostream& operator << (ostream& os, vector<Op>&
    vop)
{
        vector<Op>::iterator i;
        for(i = vop.begin(); i < vop.end(); i
            ++)
                os << *i << "| ";
        return os;
}

istream& operator >> (istream& is, vector<Op>&
    vop)
{
        int SIZE = 32768/2;
        char inp[ SIZE ];
        // clear our operation vector ...
        vop.erase(vop.begin(),vop.end());
        is.getline(inp, SIZE);
        string input(inp);
        // won't compile without static cast
        // 20-Jul-04 AWC reintroduce lower-
            casing of everything
        //------------- made appropriate
            changes to QState.cxx
        //------------- stringToRow() function
            and the log parsers
        //------------- in ./av/ - quickly
            tested - 3bit_1.arq works.
        transform(input.begin(),input.end(),
            input.begin(),static_cast<int (*)(
            int)>(std::tolower));
        Trim(input);

        // 20-Jul-04 ILC remove trailing comment
            , if any
        {
          size_t idx = input.find_first_of("#");
          if (idx < input.size()){
#ifdef DEBUG
            cout << "[Op] Found comment in '"
                    << input << "'\n";
#endif
            input = input.substr(0,idx);
#ifdef DEBUG
            cout << "[Op] input modified to '"
                    << input << "'\n";
#endif
          }
        }

        while( input.size() > 0 && input[0] != '
            #')
        {
                string sop = Token(input,";");
                string arg; Op op;
                op.opcode = Token(sop," \t");
                while( sop.size() > 0 )
                {
                        arg = Token(sop,",");
                        op.args.push_back(arg);
                }
                vop.push_back(op);
        }
        return is;
}

bool operator < (const Op& opl, const Op& opr)
{
        return (opl.opcode < opr.opcode);
}

bool operator > (const Op& opl, const Op& opr)
{
        return (opl.opcode > opr.opcode);
}
```

C.2.14 Op.hh

```
// Op.hh
// Andrew Cross <awcross@mit.edu>
// Simple operation
// $Id: Op.hh 10 2004-07-16 06:37:13Z awcross $

#ifndef __Op__
#define __Op__

#include <vector>
#include <string>
#include <algorithm>
#include <cctype>

#include "utilities.hh"

using namespace std;

class Op
{
public:
    string opcode;
    int nticks;
    vector<string> args;

    Op();
    Op(string o);
};

ostream& operator << (ostream& os, Op& op);
ostream& operator << (ostream& os, vector<Op>&
    vop);
istream& operator >> (istream& is, vector<Op>&
    vop);
bool operator < (const Op& opl, const Op& opr);
bool operator > (const Op& opl, const Op& opr);

#endif
```

C.2.15 PCGMachine.cc

```
// PCGMachine.cxx
// $Id: PCGMachine.cc 223 2004-12-08 00:52:51Z
    awcross $

#include "include/PCGMachine.hh"
#include "include/globals.hh"
#include "include/loglevels.hh"

using namespace std;

void PCGMachine::save(void)
{
        CGMachine::save();
}


void PCGMachine::restore(void)
{
        CGMachine::restore();
}


PCGMachine::PCGMachine()
{
}


PCGMachine::PCGMachine(string fname) : CGMachine
    (fname)
{
}


// Return true if qubits a and b are next to
    each other
bool PCGMachine::nnqubits(string& a, string& b)
{
        int l0 = P.getloc(a);
        int l1 = P.getloc(b);
        //cout << P.getloc(a) << "," << P.
            getloc(b) << endl;
        if( l0-l1 == xseparator || l0-l1 == -
            xseparator ||
            l0-l1 == 1   || l0-l1 == -1 ) return
                true;
        return false;
}

bool PCGMachine::process(void)
{
    long i;
    vector<Op>::iterator j;
    vector<string>::iterator s;

    // First, process "seq" using CGMachine
```

```
if( !CGMachine::process() ) return false;

// If that didn't fail, go ahead and do our
    work.
for( i = 0; i < seq.size(); i++ )
{
    for( j = seq[i].begin(); j < seq[i].end
        (); j++ )
    {
        // INIT qubit,location_integer
        // FIX check args.size()
        if( j->opcode == "init" )
        {
            if (log_level & DEBUG)
                cerr << "DBG processing
                    init" << endl;
            if( !hasqubit(j->args[0]) ||
                !isInteger(j->args[1]))
            {
                cerr << "Error! Incorrect
                    arguments for "
                    << "\"init qubit,loc\"
                        at instruction "
                    << i << endl;
                return false;
            }

            int location_integer = str2int(j
                ->args[1]);
            P.init(j->args[0],
                location_integer);
            if (log_level & MOVE) {
                int y = location_integer
                    % xseparator;
                int x = (
                    location_integer -
                    y) / xseparator;
                cout << "place " << j->
                    args[0] << " ";
                cout << x << " " << y
                    << endl;
            }
            if (log_level & DEBUG)
                cerr << "DBG place qubit
                    " << j->args[0]
                    << " " << i << endl
                    ;
            j->opcode = "nop";
        }
        // MOVE QUBIT,DIR,STEPS
        // FIX check args.size()
        else if( j->opcode == "move" )
        {
            if (log_level & DEBUG)
                cerr << "DBG processing
                    move" << endl;
            if( !hasqubit(j->args[0]) ||
                (j->args[1] != "n" && j->
                    args[1] != "e" &&
                 j->args[1] != "s" && j->
                    args[1] != "w") ||
                !isInteger(j->args[2]))
            {
                cerr << "Error! Incorrect
                    arguments for "
                    << "\"move qubit,[nesw
                        ],steps\" at
                        instruction "
                    << i << endl;
                return false;
            }
            j->nticks = str2int(j->args[2]);
            if( j->nticks == 0 )
            {
                cerr << "Error! Moving qubit
                    0 steps at instruction
                    "
                    << i << endl;
                return false;
            }
            continue; // don't set the clock
                ticks again
        }
        // UNKNOWN
        else
        {
            if (log_level & DEBUG)
                cerr << "DBG Warning!
                    Opcode " << j->
                    opcode
                                << " unknown to
                                    PCGMachine"
                                    << endl;
                continue; // don't set the
                    number of clock cycles
            }

            j->nticks = times[j->opcode];
        }
    }
    return true;
}
```

```
bool PCGMachine::exec(Op& op, bool log)
{

    if( op.opcode == "init" )
    {
        int location_integer = str2int(op.args
            [1]);
        if (log && log_level & MOVE) {
            int y = location_integer %
                xseparator;
            int x = (location_integer - y)
                / xseparator;
            cout << "place " << op.args
                [0] << " ";
            cout << x << " " << y << endl;
        }
        P.init(op.args[0],location_integer);
        op.opcode = "nop";
    }
    else if( op.opcode == "move" )
    {
        if(log && log_level & MOVE) cout << op
            << endl;
        P.moveq(op.args[0], toupper(op.args
            [1][0]), str2int(op.args[2]));
    }
    else if( op.opcode == "cnot" )
    {
        if(nnqubits(op.args[0],op.args[1]))
            return CGMachine::exec(op,log);
        else
        {
            cerr << "Error! cnot applied to
                distant qubits.";
            cerr << "Surely you didn't
                intend this. Check your
                code.";
            cerr << endl;
            return false;
        }
    }
    else if( op.opcode == "cz" )
    {
// FIX - this is also broken!
//      if(nnqubits(op.args[0],op.args[1]))
        return CGMachine::exec(op,log);
/*      else
        {
            cerr << "Error! cz applied to
                distant qubits.";
            cerr << "Surely you didn't
                intend this. Check your
                code.";
            cerr << endl;
            return false;
        }*/
    }
    else return CGMachine::exec(op,log);

    return true;
}

void PCGMachine::layout(string filename)
{
    ifstream Layout (filename.c_str(), ios::in);
    int location;

    // FIX - careful layout file parsing?
        if (log_level & DEBUG)
            cerr << "DBG load layout " <<
                filename << endl;

    while( Layout >> location) P.add(location);
}
```

C.2.16 PCGMachine.hh

```
// PCGMachine.hh
// $Id: PCGMachine.hh 6 2004-07-16 03:48:34Z
     awcross $
// Hash table internals by Darshan Thaker <
     ddthaker@ucdavis.edu>
// Tzvetan Metodiev <tsmetodiev@ucdavis.edu>
// Andrew Cross <awcross@mit.edu>

#ifndef __PCGMachine__
#define __PCGMachine__

#include "CGMachine.hh"
#include "Planar.hh"

using namespace std;

class PCGMachine : public CGMachine
{

protected:
    Planar P;

    static const int xseparator = 1000;
    bool process(void);
    bool exec(Op& op, bool log);

    bool nnqubits(string& a, string& b);   //
         test if a and b are near

public:
    PCGMachine();
    PCGMachine(string fname);
    void layout(string filename);
    void save(void);
    void restore(void);
};

#endif
```

C.2.17 Planar.cc

```
// Planar.cxx
// $Id: Planar.cc 142 2004-08-15 04:59:44Z
     awcross $
// Implements the ion-trap layout instructions.

#include "include/Planar.hh"

using namespace std;

// Constructor: Create hash tables of a given
     size
Planar::Planar(int ts)
{
    if(!ts) TAB_SIZE = 2048;
    else TAB_SIZE = ts;
    ptab = create_p_table(TAB_SIZE);
    qtab = create_q_table(TAB_SIZE);
}

// taxicab distance from location l1 to location
     l2,
// where both are given by 1000*x+y
int Planar::distance(int l1, int l2)
{
        int x1, y1, x2, y2;
        int d;
        y1 = l1 % 1000;
        y2 = l2 % 1000;
        x1 = (l1 - y1)/1000;
        x2 = (l2 - y2)/1000;
        d = (x2>x1)?x2-x1:x1-x2;
        d += (y2>y1)?y2-y1:y1-y2;
        return d;
}

// Call the Hash table move function.
void Planar::moveq(string name, char dir, int
     steps)
{
    move(name.c_str(), dir, steps, ptab, qtab
         , 0);
}

// Get a qubit's location
int Planar::getloc(string name)
{
    int cell = get_loc(name.c_str(),ptab,qtab);
    return cell;
}

// Return the qubit object sitting at some
     location
string Planar::getqid(int loc)
{
    string name = get_name(loc,ptab);
    return name;
}


// Return the qubit object with the given name
q_hentry * Planar::get_qobj (string name)
{
    q_hentry *q;
    q = find_in_qhtable(name.c_str(), qtab);
    return q;
}

// Return the physical qubit object with the
     given location
p_hentry * Planar::get_pobj (int loc)
{
    p_hentry *p;
    if( (p = find_in_phtable(loc, ptab)) != NULL
         )
    {
    }
    return p;
}

// Return the adjacent qubit if it exists and is
// located in the specified direction ("N","S","
     E","W")
// NULL if there are no adjacent qubits or if
     electrode is adjacent
q_hentry * Planar::get_qadj (string name, char
     dir)
{
    q_hentry *q;
    q = get_adj( name.c_str(), dir, ptab, qtab);
    return q;

}

// Return the nearest qubit to our qubit in any
     of
// of the four cardinal directions. Returns NULL
// if we run into an electrode (i.e. out of
     legal qubit locs).
q_hentry * Planar::get_qnear (string name, char
     dir)
{
    q_hentry *q;
    q = get_near( name.c_str(), dir, ptab, qtab)
         ;
    return q;
}

// Return the qubit a distance d from the name-
     qubit
// is some prespecified direction.
//
// Return NULL if nothing there or we hit an
     electrode.
q_hentry * Planar::get_qdist (string name, char
     dir, int d)
{
    q_hentry *q;
    q = get_dist( name.c_str(), dir, d, ptab,
         qtab);
    return q;
}

// Get all ion-qubits a distance d from the
     input qubit.
// Returns a vector container with qubit
     pointers.
// If the returned vector is empty, then there
     are no ions a
// distance d from the input qubit in any
     direction.
//
// d = 1 if you want all adjacent qubits in all
     directions.
vector< q_hentry* > Planar::get_all (string name
     , int d)
{
```

```
    vector< q_hentry * > qubits;
    q_hentry *tmp;

    string dirs = "NSEW";
    string::const_iterator i1 = dirs.begin();
    char dir;
    int i;

    while (i1 != dirs.end())
    {
        dir = *i1;
        for (i = d; i > 0; i--)
        {
            tmp = get_qdist (name,*i1,i);
            if (tmp != NULL)
            {
                qubits.push_back(tmp);
            }
        }
        ++i1;
    }
    return qubits;
}

// Return TRUE if a set of Qubits Q are a chain
    and FALSE otherwise.
bool Planar::is_chain(vector< string > Q)
{
    vector< string >::const_iterator itr1 = Q.
        begin();
    vector< q_hentry* > adj_qubits;
    int samecount;

    while (itr1 != Q.end())
    {
        samecount = 0;
        adj_qubits = get_all(*itr1,1);
        // if no adj qubits are part of Q, then
            Q is not a chain.
        if (adj_qubits.size() == 0) return FALSE
            ;
        for(int i = 0; i < adj_qubits.size(); i
            ++)
        {
            for (int j = 0; j < Q.size(); j++)
            {
                if (Q[j] == adj_qubits[i]->key)
                {
                    samecount++;
                }
            }
        }
        if (samecount == 0) return FALSE;
        ++itr1;
    }
    return TRUE;
}
```

C.2.18 Planar.hh

```
// Planar.hh
// $Id: Planar.hh 6 2004-07-16 03:48:34Z awcross
    $
// Hash table internals by Darshan Thaker <
    ddthaker@ucdavis.edu>
// Tzvetan Metodiev <tsmetodiev@ucdavis.edu>
// Andrew Cross <awcross@mit.edu>
//
// Implements planar layout and movement.

#ifndef __Planar__
#define __Planar__

#include <iostream>
#include <fstream>
#include <stdio.h>
#include <stdlib.h>
#include <map>
#include <vector>
#include <string>

#include "Op.hh"
#include "utilities.hh"

extern "C"
{
#include "planarops.h"
}
```

```
using namespace std;

class Planar
{

private:
    int TAB_SIZE;          // size of the table
    p_htable *ptab;        // pointer to a
        physical locations hash table
    q_htable *qtab;        // pointer to a hash
        table of ions by name

//    p_htable * getPtable (void) { return ptab
    ; }
//    q_htable * getQtable (void) { return qtab
    ; }

    q_hentry * get_qobj (string);
    p_hentry * get_pobj (int);
    q_hentry * get_qadj (string, char);
    q_hentry * get_qnear (string, char);
    q_hentry * get_qdist (string, char, int);

public:

    void add(int loc) { add_loc(loc,ptab); }
    void init(string id, int pos)
        { init_position(id.c_str(),pos, ptab,
            qtab); }
    void moveq(string, char, int);
    int getloc(string);
    string getqid (int);
    bool is_chain ( vector< string >);
    int distance(int l1, int l2);
    // perhaps hide more later ....
    vector<q_hentry*> get_all (string, int);

    Planar(int ts = 0);
    void loadLayout(string filename);
};

#endif
```

C.2.19 QState.cc

```
// QState.cxx
// Derived from Scott Aaronson's CHP simulator
// Andrew Cross <awcross@mit.edu>
// $Id: QState.cc 152 2004-08-20 18:34:08Z setso
    $
// Quantum state (or subspace) given by a set
// of stabilizer generators in the binary
// representation.

#include "include/QState.hh"

// Return the number of qubits
long QState::size(void)
{
    return n;
}

// Test if a subset of qubits of this QState
    equals the given
// stabilizer state.
// The vector qlist tells what qubits map from
    left to right to each
// of the generators elements.
// WARNING: I don't check to make sure qlist and
    sgens are valid vectors
// of strings.
bool QState::subset(vector<int> qlist, vector<
    string> sgens)
{
        QState sstate(sgens.size());
        QState temp = (*this); // make a copy of
            this
        vector<pair<int,int> > plist;
        vector<pair<int,int> >::iterator pit;
        int endnum = qlist.size();
        int bounce;

        // reorder temp
        for( int j = 0; j < qlist.size(); j++ )
        {
                // imagine pairs (qlist[j],j) --
```

```
                // if qlist[j]>j permute those
                    columns and add this pair
                // to plist; if qlist[j]==j do
                    nothing; if qlist[k]<j
                // then we may have already
                    moved the qubit at qlist[k]
                // somewhere else, so search
                    plist from the beginning
                    and
                // replace qlist[k] to apply the
                    transpositions
                if( qlist[j] > j )
                {
                        temp.swapcol(qlist[j],j)
                            ;
                        plist.push_back(pair<int
                            ,int>(qlist[j],j));
                } else if( qlist[j] < j )
                {
                        bounce = qlist[j];
                        for( pit = plist.begin()
                            ; pit != plist.end
                            (); pit++ )
                                if( bounce ==
                                    pit->second
                                    ) bounce
                                    = pit->
                                    first;
                        if( bounce != j )
                        {
                                temp.swapcol(
                                    bounce,j);
                                plist.push_back(
                                    pair<int,
                                    int>(bounce
                                    ,j));
                        }
                }
        }
        // remove extra qubits in temp
        for( long j = endnum; j < n; j++ ) temp.
            remove(endnum);
        // Build up the QState corresponding to
            sgens
        for( long i = 0; i < sstate.k; i++ )
                sstate.stringToRow(sgens[i], i+
                    sstate.k);

        // Now check if they are equivalent
        bool eq = (temp == sstate);
        return eq;
}

//
//
//****************** END SUBSET
    *******************************//


// SUBSET_KET:
// Print the ket of a given subset of qubits.
    Should be
// non-destructive.
string QState::subset_ket(vector<int> qlist)
{
        QState temp = (*this); // make a copy of
            this
        vector<pair<int,int> > plist;
        vector<pair<int,int> >::iterator pit;
        int endnum = qlist.size();
        int bounce;

        // reorder temp (identical as the
            routine is subset.
        for( int j = 0; j < qlist.size(); j++ )
        {
                if( qlist[j] > j )
                {
                        temp.swapcol(qlist[j],j)
                            ;
                        plist.push_back(pair<int
                            ,int>(qlist[j],j));
                } else if( qlist[j] < j )
                {
                        bounce = qlist[j];
                        for( pit = plist.begin()
                            ; pit != plist.end
                            (); pit++ )
                                if( bounce ==
                                    pit->second
                                    ) bounce
                                    = pit->
                                    first;
                        if( bounce != j )
                        {
                                temp.swapcol(
                                    bounce,j);
                                plist.push_back(
                                    pair<int,
                                    int>(bounce
                                    ,j));
                        }
                }
        }
        // remove extra qubits in temp
        for( long j = endnum; j < n; j++ ) temp.
            remove(endnum);
        return temp.destructiveKet();
}

//
//
//****************** END SUBSETKET
    *******************************//


void QState::stringToRow(string s, long row)
{
    r[row] = 0;
    if(s[0] == '-')
    {
        r[row] += 2;
        s.erase(0,1);
    }
// comment this part of the phase out so that
// lowercase generators are not ambiguous ...
// if "i" needed, make changes here and below.
//    if(s[0] == 'i')
//    {
//        r[row] += 1;
//        s.erase(0,1);
//    }

    for( int j = 0; j < s.size(); j++ )
    {
        int word = j >> 5;
        int pwj = pw[j%32];
        int npwj = 0xFFFFFFFF ^ pwj;
        switch(s[j])
        {
        case 'x':
            x[row][word] |= pwj;
            z[row][word] &= npwj;
            break;
        case 'y':
            x[row][word] |= pwj;
            z[row][word] |= pwj;
            break;
        case 'z':
            x[row][word] &= npwj;
            z[row][word] |= pwj;
            break;
        case 'i':
            x[row][word] &= npwj;
            z[row][word] &= npwj;
            break;
        default:
            break;
        }
    }
}

// copy constructor
QState::QState(const QState& q)
{
    n = q.n;
    k = q.k;
    if( n > 0 && k > 0 )
    {
        x = new unsigned long *[2*k+1];
        z = new unsigned long *[2*k+1];
        r = new int [2*k+1];
        over32 = (n>>5) + 1;
        pw[0] = 1;
        for( int i = 1; i < 32; i++ )
            pw[i] = 2*pw[i-1];
        for( int i = 0; i < 2*k + 1; i++ )
        {
            x[i] = new unsigned long[over32];
            z[i] = new unsigned long[over32];
            for( int j = 0; j < over32; j++ )
            {
                x[i][j] = q.x[i][j];
                z[i][j] = q.z[i][j];
```

```
            }
            r[i] = q.r[i];
        }
    }
    else
    {
        x = NULL;
        z = NULL;
        r = NULL;
        over32 = 0;
    }
}

// assignment
QState& QState::operator=(const QState& q)
{
    if( *this == q ) return *this;

    // not efficient
    sfree();
    n = q.n;
    k = q.k;
    x = new unsigned long *[2*k+1];
    z = new unsigned long *[2*k+1];
    r = new int [2*k+1];
    over32 = (n>>5) + 1;
    pw[0] = 1;
    for( int i = 1; i < 32; i++ )
        pw[i] = 2*pw[i-1];
    for( int i = 0; i < 2*k + 1; i++ )
    {
        x[i] = new unsigned long[over32];
        z[i] = new unsigned long[over32];
        for( int j = 0; j < over32; j++ )
        {
            x[i][j] = q.x[i][j];
            z[i][j] = q.z[i][j];
        }
        r[i] = q.r[i];
    }
    return *this;
}

// compare
// two states can be equal even if their tableau
    are different
bool QState::operator==(const QState& rhs)
{
    if( k != rhs.k || n != rhs.n ) return false;

    // Work with copies since GE and GJ can
        destroy the
    // commutation relations between stabilizer
        and destabilizer
    QState tlhs = (*this); // use copy
        constructors
    QState trhs = rhs;

    tlhs.gaussjordan();
    trhs.gaussjordan();

    // only compare stabilizer generators
    for( int j = trhs.k; j < 2*trhs.k; j++ )
    {
        // we can assume the words are zero
            padded because it is
        // ensured in the constructors
        for( int l = 0; l < trhs.over32; l++ )
        {
            if( tlhs.x[j][l] != trhs.x[j][l] )
                return false;
            if( tlhs.z[j][l] != trhs.z[j][l] )
                return false;
        }
        if( tlhs.r[j] != trhs.r[j] ) return
            false;
    }
    return true;
}

// returns true if row i commutes with row j
bool QState::rowcommutes(long i, long j)
{
    int w = 0;
    for( int l = 0; l < over32; l++ )
        w += weight(x[i][l] & z[j][l]) + weight(
            z[i][l] & x[j][l]);
    if( w%2 == 0 ) return true;
    return false;
}

// calculate the hamming weight of w
int QState::weight(unsigned long w)
{
    int i=0;
    unsigned long j=1;
    do
    {
        if( w & j ) i++;
        j *= 2;
    } while( j != 0x80000000 );
    if( w & j ) i++;
    return i;
}

// Print the raw tableau
string QState::raw(void)
{
    string result;
    long i, j, m, p;

    result = "cgraw " + to_string(k)+" "+
        to_string(n)+" "
            +to_string(over32)+"\n";

    result += "cgraw Z\n";
    for( i = 0; i < (k>0?(2*k + 1):0); i++ )
    {
        result += "cgraw " + to_string(r[i]) + "
            ";
        for( j = 0; j < over32; j++ )
        {
            if( j == over32 - 1 ) p = (n-1)%32;
            else p = 31;
            for( m = 0; m <= p; m++ )
            {
                int bit = ((z[i][j] >> m) & 1);
                result += to_string(bit);
            }
            if( over32 > 1 && j < over32 - 1)
                result += "\ncgraw  ";
        }
        result += "\n";
    }
    result += "cgraw X\n";
    for( i = 0; i < (k>0?(2*k + 1):0); i++ )
    {
        result += "cgraw " + to_string(r[i]) + "
            ";
        for( j = 0; j < over32; j++ )
        {
            if( j == over32 - 1 ) p = (n-1)%32;
            else p = 31;
            for( m = 0; m <= p; m++ )
            {
                int bit = ((x[i][j] >> m) & 1);
                result += to_string(bit);
            }
            if( over32 > 1 && j < over32 - 1)
                result += "\ncgraw  ";
        }
        if( i < 2*k ) result += "\n";
    }

    return result;
}

// CNOT from control b to target c
void QState::cnot(long b, long c)
{
    long i;
    long b5;
    long c5;
    unsigned long pwb;
    unsigned long pwc;

    b5 = b>>5;
    c5 = c>>5;
    pwb = pw[b&31];
    pwc = pw[c&31];
    for (i = 0; i < 2*k; i++)
    {
        if (x[i][b5]&pwb) x[i][c5] ^= pwc;
        if (z[i][c5]&pwc) z[i][b5] ^= pwb;
        if ((x[i][b5]&pwb) && (z[i][c5]&pwc) &&
            (x[i][c5]&pwc) && (z[i][b5]&pwb))
            r[i] = (r[i]+2)%4;
        if ((x[i][b5]&pwb) && (z[i][c5]&pwc) &&
            !(x[i][c5]&pwc) && !(z[i][b5]&pwb))
            r[i] = (r[i]+2)%4;
    }
}
```

```
// Hadamard on b
void QState::h(long b)
{
    long i;
    unsigned long tmp;
    long b5;
    unsigned long pwt;

    b5 = b>>5;
    pwt = pw[b&31];
    for (i = 0; i < 2*k; i++)
    {
        tmp = x[i][b5];
        x[i][b5] ^= (x[i][b5] ^ z[i][b5]) & pwt;
        z[i][b5] ^= (z[i][b5] ^ tmp) & pwt;
        if ((x[i][b5]&pwt) && (z[i][b5]&pwt)) r[
            i] = (r[i]+2)%4;
    }
}

// Phase gate on b
void QState::s(long b)
{
    long i;
    long b5;
    unsigned long pwt;

    b5 = b>>5;
    pwt = pw[b&31];
    for (i = 0; i < 2*k; i++)
    {
        if ((x[i][b5]&pwt) && (z[i][b5]&pwt)) r[
            i] = (r[i]+2)%4;
        z[i][b5] ^= x[i][b5]&pwt;
    }
}

// Sets row i equal to row m
void QState::rowcopy(long i, long m)
{
    long j;

    for (j = 0; j < over32; j++)
    {
        x[i][j] = x[m][j];
        z[i][j] = z[m][j];
    }
    r[i] = r[m];
}

// Sets row i equal to the bth observable (X_1
    ,...X_n,Z_1,...,Z_n)
void QState::rowset(long i, long b)
{
    long j;
    long b5;
    unsigned long b31;

    for (j = 0; j < over32; j++)
    {
        x[i][j] = 0;
        z[i][j] = 0;
    }
    r[i] = 0;
    if (b < n)
    {
        b5 = b>>5;
        b31 = b&31;
        x[i][b5] = pw[b31];
    }
    else
    {
        b5 = (b - n)>>5;
        b31 = (b - n)&31;
        z[i][b5] = pw[b31];
    }
}

// Return the phase (0,1,2,3) when row i is LEFT
    -multiplied by row m
int QState::clifford(long i, long m)
{
    long j;
    long l;
    unsigned long pwt;
    long e=0;    // Power to which i is raised

    for (j = 0; j < over32; j++)
        for (l = 0; l < 32; l++)
        {
            pwt = pw[l];
```

```
            if ((x[m][j]&pwt) && (!(z[m][j]&pwt)
                ))        // X
            {
                if ((x[i][j]&pwt) && (z[i][j]&
                    pwt)) e++;           // XY=iZ
                if ((!(x[i][j]&pwt)) && (z[i][j
                    ]&pwt)) e--;// XZ=-iY
            }
            if ((x[m][j]&pwt) && (z[m][j]&pwt))
                          // Y
            {
                if ((!(x[i][j]&pwt)) && (z[i][j
                    ]&pwt)) e++;// YZ=iX
                if ((x[i][j]&pwt) && (!(z[i][j]&
                    pwt))) e--;// YX=-iZ
            }
            if ((!(x[m][j]&pwt)) && (z[m][j]&pwt
                ))        // Z
            {
                if ((x[i][j]&pwt) && (!(z[i][j]&
                    pwt))) e++;       // ZX=iY
                if ((x[i][j]&pwt) && (z[i][j]&
                    pwt)) e--;           // ZY=-iX
            }
        }

    e = (e+r[i]+r[m])%4;
    if (e>=0) return e;
    else return e+4;
}

// Swap rows i and k
void QState::rowswap(long i, long m)
{
    rowcopy(2*n, m); rowcopy(m, i); rowcopy(i
        , 2*n);
}

// Left-multiply row i by row m
void QState::rowmult(long i, long m)
{
    long j;

    r[i] = clifford(i,m);
    for (j = 0; j < over32; j++)
    {
        x[i][j] ^= x[m][j];
        z[i][j] ^= z[m][j];
    }
}

// Free the memory associated with the state
void QState::sfree(void)
{
    long i;

    if( n > 0 )
    {
        for(i = 0; i < 2*k + 1; i++)
        {
            delete [] x[i];
            delete [] z[i];
        }
        delete [] r;
        delete [] z;
        delete [] x;
        r = NULL;
        z = NULL;
        x = NULL;
        n = k = 0;
        over32 = 0;
        pw[0] = 1;
        for (i = 1; i < 32; i++)
            pw[i] = 2*pw[i-1];
    }
}

// Create an n-qubit |0> state
void QState::sInit(long ni)
{
    long i;
    long j;

    n = ni;
    k = ni;
    x = new unsigned long *[2*k + 1];
    z = new unsigned long *[2*k + 1];
    r = new int [2*k + 1];
    over32 = (n>>5) + 1;
    pw[0] = 1;
    for (i = 1; i < 32; i++)
        pw[i] = 2*pw[i-1];
```

```
    for (i = 0; i < 2*n + 1; i++)
    {
        x[i] = new unsigned long [over32];
        z[i] = new unsigned long [over32];
        // zero the words
        for (j = 0; j < over32; j++)
        {
            x[i][j] = 0;
            z[i][j] = 0;
        }
        if (i < n)
            x[i][i>>5] = pw[i&31];
        else if (i < 2*n)
        {
            j = i-n;
            z[i][j>>5] = pw[j&31];
        }
        r[i] = 0;
    }
}

// remove the ith generator and its
    corresponding
// destabilizer , note that i is >= 0 and < q->k
void QState::removerow(long i)
{
    unsigned long j, m;

    if( k == 1 )
    {
        sfree();
        return;
    }

    // shift the ith destabilizer through all
        the
    // destabilizers and stabilizers , into the 2
        k - 1 position
    for( j=i+1; j<2*k; j++ ) rowswap(j-1,j);

    // shift the k+i-1 generator (the ith
        generator , now shifted
    // from the prior rowswaps) through all the
        elements and into
    // the 2*k - 1 position.
    for( j=k+i; j<2*k; j++ ) rowswap(j-1,j);

    // Free the last two generators
    delete [] x[2*k];
    delete [] z[2*k];
    delete [] x[2*k-1];
    delete [] z[2*k-1];

    // adjust the pointer arrays, etc, to two
        fewer locations
    // by setting q->k to q->k - 1 and
        reallocating.
    k = k - 1;

    // Allocate and copy while we free the old
        space
    unsigned long **tx = new unsigned long *[2*k
        +1];
    unsigned long **tz = new unsigned long *[2*k
        +1];
    int *tr = new int [2*k+1];
    for( m = 0; m < 2*k + 1; m++ )
    {
        tx[m] = new unsigned long [over32];
        tz[m] = new unsigned long [over32];
        for( j = 0; j < over32; j++ )
        {
            tx[m][j] = x[m][j];
            tz[m][j] = z[m][j];
        }
        tr[m] = r[m];
        delete [] x[m];
        delete [] z[m];
    }
    delete [] x;
    delete [] z;
    delete [] r;
    // Ok, copied and deleted. Point to the copy
        .
    x = tx;
    z = tz;
    r = tr;
}

// put new identity generators on the end of the
    stabilizer
// and destabilizer list
void QState::addrow(void)
{
    unsigned long j;

    // grow the tableau by two rows, creating
        new space
    // while freeing the old space
    k = k + 1;
    unsigned long **tx = new unsigned long * [2*
        k+1];
    unsigned long **tz = new unsigned long * [2*
        k+1];
    int *tr = new int [2*k+1];
    for( long i = 0; i < 2*k + 1; i++ )
    {
        tx[i] = new unsigned long [over32];
        tz[i] = new unsigned long [over32];
        for( long j = 0; j < over32; j++ )
        {
            if( i < 2*k - 1 )
            {
                tx[i][j] = x[i][j];
                tz[i][j] = z[i][j];
            }
        }
        if( i < 2*k - 1 )
        {
            delete [] x[i];
            delete [] z[i];
            tr[i]= r[i];
        }
    }
    delete [] x;
    delete [] z;
    delete [] r;

    x = tx;
    z = tz;
    r = tr;
    x[2*k-1] = new unsigned long [over32];
    x[2*k] = new unsigned long [over32];
    z[2*k-1] = new unsigned long [over32];
    z[2*k] = new unsigned long [over32];

    // set these new rows to zero
    for( j = 0; j < over32; j++ )
    {
        x[2*k-1][j] = 0;
        z[2*k-1][j] = 0;
        x[2*k][j] = 0;
        z[2*k][j] = 0;
    }
    r[2*k-1] = 0;
    r[2*k] = 0;

    // swap the new 2*k - 1 generator (last ,
        before scratch) up
    // to the k-1 position , shifting all the
        stabilizer
    // generators down one row.
    for( j = 2*k - 1; j >= k; j-- ) rowswap(j-1,
        j);

}

// remove the jth column from every generator in
    q
// does not modify the phase
// j >= 0 and j <= q->n
void QState::removecol(long j)
{
    unsigned long FFFF = 0xFFFFFFFF; //
        2^32 - 1 = 4294967295
    unsigned long jword = j>>5;   // the word
        containing the jth bit
    unsigned long leftj = FFFF << ((j&31) + 1);
        // left mask
    unsigned long rightj = (FFFF ^ leftj) ^ pw[j
        &31];    // right mask
    unsigned long i, m;

    // a generator with enumerated terms
        1 2 3 ... n
    // is organized in memory like this:
    // 32 31 ... 1 | 64 63 ... 33 | ...
    // where the vertical bars separate words.

    // FFFF << 32 == FFFF, so we have to correct
        that here
    if( (j&31) == 31 )
    {
        leftj = 0;
```

```
        rightj = (FFFF ^ leftj) ^ pw[j&31];
    }

    // iterate over all generators
    for( m = 0; m < 2*k +1; m++ )
    {
        // start at the word containing the jth
            bit
        for( i = jword; i < over32; i++ )
        {
            if( i == jword )
            {
                // if this is the word
                    containing the jth bit
                // right shift the part to the
                    left of the jth bit
                x[m][i] = ((x[m][i] & leftj)>>1)
                    |
                  (x[m][i] & rightj);
                z[m][i] = ((z[m][i] & leftj)>>1)
                    |
                  (z[m][i] & rightj);
            } else {
                // otherwise, just shift the
                    word right one
                // to make up for the missing
                    bit
                x[m][i] = x[m][i] >> 1;
                z[m][i] = z[m][i] >> 1;
            }
            // if there are still more words to
                the right
            // take the right-most bit of the
                next word
            // and place it at the left-most bit
                of this word
            if( i < over32 - 1 )
            {
                x[m][i] |= ((x[m][i+1] & 1)
                    << 31);
                z[m][i] |= ((z[m][i+1] & 1)
                    << 31);
            }
        }
    }
    // release some memory?
    n = n - 1;
    if( ((n>>5) + 1) < over32 )
    {
        // we need fewer words to represent each
            row
        over32 = (n>>5) + 1;
        for( i=0; i< 2*k + 1; i++ )
        {
            unsigned long *tx = new unsigned
                long [over32];
            unsigned long *tz = new unsigned
                long [over32];
            for( m=0; m < over32; m++ )
            {
                tx[m] = x[i][m];
                tz[m] = z[i][m];
            }
            delete [] x[i];
            delete [] z[i];
            x[i] = tx;
            z[i] = tz;
        }
    }
}

// add a column to the end of every generator in
    q.
// does not modify the phase.
void QState::addcol(void)
{

    unsigned long i, j;

    n = n + 1;
    // should another word be allocated?
    if( (n>>5) + 1 > over32 )
    {
        over32++;
        for( i=0; i < 2*k + 1; i++ )
        {
            unsigned long *tx = new unsigned
                long [over32];
            unsigned long *tz = new unsigned
                long [over32];
            for( j=0; j < over32; j++ )
            {
                if( j == over32 - 1 )
                {
                    tx[j] = 0;
                    tz[j] = 0;
                }
                else
                {
                    tx[j] = x[i][j];
                    tz[j] = z[i][j];
                }
            }
            delete [] x[i];
            delete [] z[i];
            x[i] = tx;
            z[i] = tz;
        }
    }
}

// swap columns i and j (qubit relabel)
void QState::swapcol(long i, long j)
{
    long m;
    long i5 = i >> 5;
    long j5 = j >> 5;
    unsigned long pwi = pw[i&31];
    unsigned long pwj = pw[j&31];
    unsigned long xi, zi;

    for( m = 0; m < 2*k; m++ )
    {
        xi = x[m][i5]&pwi;       // copy column i
        zi = z[m][i5]&pwi;
        x[m][i5] |= pwi;         // set column i
        z[m][i5] |= pwi;
        // flip column i if column j not set
        if( !(x[m][j5]&pwj) ) x[m][i5] ^= pwi;
        if( !(z[m][j5]&pwj) ) z[m][i5] ^= pwi;
        x[m][j5] |= pwj;         // set column j
        z[m][j5] |= pwj;
        // flip column j if column i not set
        if( !xi ) x[m][j5] ^= pwj;
        if( !zi ) z[m][j5] ^= pwj;
    }
}

// Return a string containing the stabilizer and
    destabilizer
// for a state q in row-reduced form
string QState::reducedState(void)
{
    QState tmp = (*this);
    tmp.gaussjordan();
    return tmp.state();
}

// Return a string containing the stabilizer and
    destabilizer
// for state q
string QState::state(void)
{
    long i;
    long j;
    long j5;
    unsigned long pwt;

    string result;

    for (i = 0; i < 2*k; i++)
    {
        if (i == k)
        {
            result += "\ncgstate   ";
            for (j = 0; j < n+1; j++)
                result += "-";
        }
        if ( i > 0 )
        {
            if (r[i]==2) result += "\ncgstate "
                + to_string(i) + " -";
            else result += "\ncgstate " +
                to_string(i) + " +";
        }
        else
        {
            if (r[i]==2) result += "\ncgstate "
                + to_string(i) + " -";
            else result += "\ncgstate " +
                to_string(i) + " +";
        }
        for (j = 0; j < n; j++)
        {
```

```
            j5 = j>>5;
            pwt = pw[j&31];
            if ((!(x[i][j5]&pwt)) && (!(z[i][j5
                ]&pwt))) result += "I";
            if ((x[i][j5]&pwt) && (!(z[i][j5]&
                pwt))) result += "X";
            if ((x[i][j5]&pwt) && (z[i][j5]&pwt)
                ) result += "Y";
            if ((!(x[i][j5]&pwt)) && (z[i][j5]&
                pwt)) result += "Z";
        }
    }

    return result;
}

// Measure qubit b
// Return         0 if outcome would always be 0
//                1 if outcome would always be 1
//                2 if outcome was random and 0
    was chosen
//                3 if outcome was random and 1
    was chosen
// sup: 1 if determinate measurement results
    should be
//      suppressed, 0 otherwise
int QState::measure(long b, int sup)
{
    int ran = 0;
    long i;
    long p;      // pivot row in stabilizer
    long m;      // pivot row in destabilizer
    long b5;
    unsigned long pwt;

    b5 = b>>5;
    pwt = pw[b&31];
    for (p = 0; p < k; p++)              // loop
        over stabilizer generators
    {
        if (x[p+k][b5]&pwt) ran = 1;
            // if a Zbar does NOT commute with
            Z_b (the
        if (ran) break;
                                    // operator
            being measured), then outcome is
            random
    }

    // If outcome is indeterminate
    if (ran)
    {
        rowcopy(p, p + k);
                    // Set Xbar_p := Zbar_p
        rowset(p + k, b + k);
            // Set Zbar_p := Z_b
        r[p + k] = 2*(rand()%2);
                    // moment of quantum
            randomness
        for (i = 0; i < 2*k; i++)
                    // Now update the Xbar's
            and Zbar's that don't commute with
            if ((i!=p) && (x[i][b5]&pwt))
                        // Z_b
                rowmult(i, p);
        if (r[p + k]) return 3;
        else return 2;
    }

    // If outcome is determinate
    if ((!ran) && (!sup))
    {
        for (m = 0; m < k; m++)
                    // Before we were checking
            if stabilizer generators commute
            if (x[m][b5]&pwt) break;
                        // with Z_b; now we're
                checking destabilizer
                generators
        rowcopy(2*k, m + k);
        for (i = m+1; i < k; i++)
            if (x[i][b5]&pwt)
                rowmult(2*k, i + k);
        if (r[2*k]) return 1;
        else return 0;
        /*for (i = m+1; i < n; i++)
          if (x[i][b5]&pw)
          {
          rowmult(m + n, i + n);
          rowmult(i, m);
          }
          return (int)r[m + n];*/
        }

        return 0;
}


// Gauss-Jordan elimination to put the
    stabilizer generators into
// a unique form.
// (Return value = number of such generators =
    log_2 of number of nonzero basis states)
long QState::gaussjordan(void)
{
    long j, m;
    long l, l5;
    unsigned long pwt;
    long row, g;

    g = gaussian();

    // jordan reduction

    // find the first row from the bottom with
        an X
    // (if it doesn't exist then row = n - 1)
    for( m = 2*k - 1; m > k - 1; m-- )
    {
        for( l = 0; l < n; l++ )
        {
            l5 = l >> 5;
            pwt = pw[l&31];
            if( x[m][l5]&pwt ) break;
        }
        if( l < n ) break;
    }

    // printf("first search for X: row = %ld,
        col = %ld\n",m, l);
    row = m;

    // for all of the Z rows
    for( m = 2*k-1; m > row; m-- )
    {
        // find the first Z from the left
        // (one should always exist)
        for( l = 0; l < n; l++ )
        {
            l5 = l >> 5;
            pwt = pw[l&31];
            if( z[m][l5]&pwt ) break;
        }

        // printf("found z at row = %ld col = %
            ld\n",m, l);

        // for all generators above with a Z
        // in the l position, reduce
        for( j = m - 1; j > k - 1; j-- )
        {
            if( z[j][l5]&pwt )
            {
                // printf("z reducing row = %ld\
                    n",j);
                rowmult(j,m);
                rowmult(j-n,m-n);
            }
        }
    }

    // for all of the X rows
    for( m = row; m > k - 1; m-- )
    {
        // find the first X from the left
        // (one should always exist)
        for( l = 0; l < n; l++ )
        {
            l5 = l >> 5;
            pwt = pw[l&31];
            if( x[m][l5]&pwt ) break;
        }

        // printf("found x at row = %ld col = %
            ld\n",m, l);

        // for all generators above with an X
        // in the l position, reduce
        for( j = m - 1; j > k - 1; j-- )
        {
            if( x[j][l5]&pwt )
            {
                // printf("z reducing row = %ld\
                    n",j);
```

```
                rowmult(j,m);
                rowmult(j-n,m-n);
            }
        }
    }

    return g;
}

// Do Gaussian elimination to put the stabilizer
    generators in the following form:
// At the top, a minimal set of generators
    containing X's and Y's, in "quasi-upper-
    triangular" form.
// (Return value = number of such generators =
    log_2 of number of nonzero basis states)
// At the bottom, generators containing Z's only
    in quasi-upper-triangular form.
long QState::gaussian(void)
{

    long i = k;
    long m;
    long m2;
    long j;
    long j5;
    long g;     // Return value
    unsigned long pwt;

    for (j = 0; j < n; j++)
    {
        j5 = j>>5;
        pwt = pw[j&31];
        for (m = i; m < 2*k; m++)       // Find
            a generator containing X in jth
            column
            if (x[m][j5]&pwt) break;
        if (m < 2*k)
        {
            rowswap(i, m);
            rowswap(i-k, m-k);
            for (m2 = i+1; m2 < 2*k; m2++)
                if (x[m2][j5]&pwt)
                {
                    rowmult(m2, i);
                        // Gaussian elimination
                        step
                    rowmult(i-k, m2-k);
                }
            i++;
        }
    }
    g = i - k;

    for (j = 0; j < n; j++)
    {
        j5 = j>>5;
        pwt = pw[j&31];
        for (m = i; m < 2*k; m++)       // Find
            a generator containing Z in jth
            column
            if (z[m][j5]&pwt) break;
        if (m < 2*k)
        {
            rowswap(i, m);
            rowswap(i-k, m-k);
            for (m2 = i+1; m2 < 2*k; m2++)
                if (z[m2][j5]&pwt)
                {
                    rowmult(m2, i);
                    rowmult(i-k, m2-k);
                }
            i++;
        }
    }

    return g;
}

// append a new qubit to q in state z+
void QState::add(void)
{
    if( n > 0 )
    {
        addcol();
        addrow();
        // initialize the new generators
        x[k-1][(n-1)>>5] = pw[(n-1)&31];
        z[2*k-1][(n-1)>>5] = pw[(n-1)&31];
        r[k-1] = 0;
        r[2*k-1] = 0;
    }
    else sInit(1);
}

// remove the bth qubit of q
void QState::remove(long b)
{

    long j, m;
    long b5 = b>>5;
    unsigned long pwt = pw[b&31];

    measure(b,1); // measure the bth qubit,
        neglecting outcome

    // find the first stabilizer generator with
        a Z in the bth term.
    // guaranteed to find one, because we
        measured Z_b.
    for( j = k; j < 2*k; j++ )
        if( z[j][b5]&pwt ) break;

    // multiply all generators with weight in
        the bth term
    // by the generator we just found.
    for( m = k; m < 2*k; m++ )
        if( z[m][b5]&pwt && m != j )
            rowmult(m,j);

    if( n > 1 )
    {
        removerow(j-k);
        removecol(b);
    }
    else sfree();
}

// return a string containing the basis state
    corresponding to
// applying the scratch space generator to the
    zero ket.
string QState::basisstate(void)
{
    long j;
    long j5;
    unsigned long pwt;
    int e = r[2*k];

    string result;

    for (j = 0; j < n; j++)
    {
        j5 = j>>5;
        pwt = pw[j&31];
        if ((x[2*k][j5]&pwt) && (z[2*k][j5]&pwt)
            )
// Pauli operator is "Y"
            e = (e+1)%4;
    }
    if (e==0) result += " +|";
    if (e==1) result += "+i|";
    if (e==2) result += " -|";
    if (e==3) result += "-i|";

    for (j = 0; j < n; j++)
    {
        j5 = j>>5;
        pwt = pw[j&31];
        if (x[2*k][j5]&pwt) result += "1";
        else result+= "0";
    }
    result += ">";
    return result;
}

// Finds a Pauli operator P such that the basis
    state P|0...0> occurs with nonzero
    amplitude in q, and
// writes P to the scratch space of q.  For this
    to work, Gaussian elimination must already
    have been
// performed on q.  g is the return value from
    gaussian(q).
void QState::seed(long g)
{

    long i;
    long j;
    long j5;
    unsigned long pwt;
    int f;
    long min;
```

```
    // the last row of q is scratch space
    r[2*k] = 0;
    for (j = 0; j < over32; j++)
    {
        x[2*k][j] = 0;              // Wipe the
            scratch space clean
        z[2*k][j] = 0;
    }
    for (i = 2*k - 1; i >= k + g; i--)
    {
        f = r[i];
        for (j = n - 1; j >= 0; j--)
        {
            j5 = j>>5;
            pwt = pw[j&31];
            if (z[i][j5]&pwt)
            {
                min = j;
                if (x[2*k][j5]&pwt) f = (f+2)%4;
            }
        }
        if (f==2)
        {
            j5 = min>>5;
            pwt = pw[min&31];
            x[2*k][j5] ^= pwt;             // Make
                the seed consistent with the
                ith equation
        }
    }
}

// returns a string containing the ket
    representation of q
string QState::ket(void)
{
    QState tmp = (*this);
    return tmp.destructiveKet();
}

// returns a string containing the ket
    representation of q
// can invalidate the commutation relations
    between
// stabilizer and destabilizer
string QState::destructiveKet(void)
{
    long g;              // log_2 of number of
        nonzero basis states
    unsigned long t;
    unsigned long t2;
    long i;

    string result;

    if( n == 0 )
    {
        return "ket ";
    }
    g = gaussian();
    if (g > 31)
    {
        result = "ket too large";
        return result;
    }
    seed(g);
    result += "ket ";
    result += basisstate();
    for (t = 0; t < pw[g]-1; t++)
    {
        t2 = t ^ (t+1);
        for (i = 0; i < g; i++)
            if (t2 & pw[i])
                rowmult(2*k, k + i);
        result += basisstate();
    }

    return result;
}

QState::QState()
{
    int i;
    // initialize the state to empty
    n = k = 0;
    x = z = NULL;
    r = NULL;
    over32 = 0;
    pw[0] = 1;
    for (i = 1; i < 32; i++)
        pw[i] = 2*pw[i-1];
}
```

C.2.20 QState.hh

```
// QState.hh
// Derived from Scott Aaronson's CHP simulator
// Andrew Cross <awcross@mit.edu>
// $Id: QState.hh 152 2004-08-20 18:34:08Z setso
    $
// Quantum state (or subspace) given by a set
// of stabilizer generators in the binary
// representation.

#include <cstdlib>
#include <string>
#include <vector>

#include "utilities.hh"

using namespace std;

#ifndef __QState__
#define __QState__

class QState
{
private:
    long n;      // # of qubits
    long k;      // # of generators
    unsigned long **x;  // (2k+1)*n matrix; x
        bits
    unsigned long **z;  // z bits
    int *r;                        // phase: 0 for
        +1, 1 for i, 2 for -1, ...
    unsigned long pw[32];          // pw[i] = 2^i
    long over32;                   // floor(n/8)
        + 1
    // state creation
    void sInit(long n);
    void sfree(void);
    // basic row operations
    void rowcopy(long i, long m);
    void rowswap(long i, long m);
    void rowset(long i, long b);
    void rowmult(long i, long m);
    bool rowcommutes(long i, long j);
    void removerow(long i);
    void addrow(void);
    // basic column operations
    void removecol(long j);
    void addcol(void);
    void swapcol(long i, long j);
    // other operations
    int clifford(long i, long m);
    string basisstate(void);
    long gaussjordan(void);
    long gaussian(void);
    void seed(long g);
    int weight(unsigned long w);
    void stringToRow(string s, long row);
    string destructiveKet(void);
public:
    QState();
    QState(long n) { sInit(n); }
    ~QState() { sfree(); }
    QState(const QState& q);
    QState& operator=(const QState& q);
    bool operator==(const QState& rhs);

    inline void reseed(unsigned int seed) {
        srand(seed); }
    void cnot(long b, long c);
    void h(long b);
    void s(long b);
    inline void gx(long b) { h(b); s(b); s(b); h
        (b); }
    inline void gy(long b) { gz(b); gx(b); }
    inline void gz(long b) { s(b); s(b); }
    int measure(long b, int sup);
    string reducedState(void);
    string state(void);
    string ket(void);
    string subset_ket(vector<int> qlist);
    string raw(void);
    void add(void);
    void remove(long b);
    long size(void);
    bool subset(vector<int> qlist, vector<string
        > sgens);
```

```
};
#endif
```

C.2.21 arq.cc

```
// arq.cxx
// Andrew Cross <awcross@mit.edu>
// Architecture Research Quantum Simulator
// $Id: arq.cc 149 2004-08-17 13:44:11Z awcross
    $
//
// Copyright (c) 2004 Scott Aaronson, Andrew
    Cross, Tzvetan Metodiev, Darshan Thaker
// Contact <awcross@mit.edu>
//
// arq is free software; you can redistribute it
    and/or modify
// it under the terms of the GNU General Public
    License as published by
// the Free Software Foundation; either version
    2 of the License, or
// (at your option) any later version.
//
// arq is distributed in the hope that it will
    be useful,
// but WITHOUT ANY WARRANTY; without even the
    implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR
    PURPOSE. See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU
    General Public License
// along with arq; if not, write to the Free
    Software
// Foundation, Inc., 59 Temple Place, Suite
    330, Boston, MA  02111-1307  USA
//

const char *cvsdate = "SVN $Id: arq.cc
    149 2004-08-17 13:44:11Z awcross $";
const char *version = "0.12 alpha";

#include <iostream>
#include <sys/time.h>
#include <time.h>

#include <cstdlib>
#include <fstream>

#include "include/utilities.hh" // for
    parseCmdLine()
#include "include/Machine.hh"
#include "include/CGMachine.hh"
#include "include/PCGMachine.hh"
#include "include/ITMachine.hh"
#include "include/NoisyCGMachine.hh"
#include "include/NoisyPCGMachine.hh"
#include "include/NoisyITMachine.hh"
#include "include/commandLine.hh"
#include "include/interactive.hh"
#include "include/loglevels.hh"

using namespace std;
using std::ofstream;
int log_level;

int main(int argc, char *argv[])
{
        // Time-keeping variables
        clock_t tick_count = clock();
        clock_t tick_count2;
        double thetime;

        // Reseed the random number generator
            from the clock
        time_t seed0 = time(0);
        srand(seed0);

        int nsamples = 1;
        double start, end;
        string machineName;
        Machine *m;
        struct commandLine cmdLine;

        int nFail;
        int nPass;
        if( !parseCmdLine(cmdLine, argc, argv)
            || !cmdLineDependencies(cmdLine))
        {
                cout << "** Quantum Computer
                    Architecture Simulator" <<
                    endl;
                cout << "** " << cvsdate << endl
                    ;
                cout << "** " << version << endl
                    ;
#ifdef TURBO
                cout << "** Compiled with -
                    DTURBO" << endl;
#endif // TURBO

                cout << "\nATTENTION: arqsample
                    can be used to collect
                    statistics!\n";
                cout << endl;
                cout << "Use: arq param=value,
                    param=value, ..." << endl;
                cout << "        arq --interactive"
                     << endl;
                cout << endl;
                cout << "parameter
                                value (
                    default)" << endl;
                cout << "------------------------"
                     << endl;
                cout << "machine (-m)
                                cc,cg,pcg,cit,
                    ncg,npcg,ncit (ncg)" <<
                    endl;
                cout << "source (-s)
                                filename (
                    required)" << endl;
                cout << "layout (-l)
                                filename (\"\")
                    " << endl;
                cout << "parameters (-p)
                            filename (required)
                    " << endl;
                cout << "samples (-n)
                                integer (1)" <<
                    endl;
                cout << "verbosity (-v)
                            integer (1, use
                     127 with arqview)" << endl
                    ;
                cout << "datafile (-f)
                            filename (\"
                    default.data\")" << endl;
                return 1;
        }

        // Start the interactive mode if it was
            requested
        if( cmdLine.interactive )
        {
                InteractivePrompt i;
                return i.start();
        }

        nsamples = cmdLine.samples;
        log_level = cmdLine.verbosity;

        // Open the Data File for writing
        ofstream dataFile( cmdLine.datafile.
            c_str(), ios::out);

        // The Machine's constructor should
        // dump the parameter file in the format
            :
        // parameter name value
        if (log_level & STARTUP) cout << "
            machine " << machineToString(
            cmdLine.machine) << endl;
        if( cmdLine.machine == CC ) m = new
            Machine(cmdLine.parameters);
        else if( cmdLine.machine == CG ) m = new
             CGMachine(cmdLine.parameters);
        else if( cmdLine.machine == PCG ) m =
            new PCGMachine(cmdLine.parameters);
        else if( cmdLine.machine == CIT ) m =
            new ITMachine(cmdLine.parameters);
        else if( cmdLine.machine == NCG ) m =
            new NoisyCGMachine(cmdLine.
            parameters);
        else if( cmdLine.machine == NPCG ) m =
            new NoisyPCGMachine(cmdLine.
```

```
	parameters);
else if( cmdLine.machine == NCIT ) m =
	new NoisyITMachine(cmdLine.
	parameters);
else
{
	cerr << "Internal Error! No
		machine! "
		<< "Check the command line
			parser." << endl;
	return 1;
}
if (log_level & STARTUP) {
	cout << "parameterfile " <<
		cmdLine.parameters << endl;
	cout << "sourcefile " << cmdLine
		.source << endl;
}
// This is defined for all machines - it
	's ok to call
// even if the particular machine doesn'
	t have a layout
m->layout(cmdLine.layout);

// Load the source file and do all of
	the relevant
// preprocessing. This will completely
	initialize
// the Machine.
try { m->load(cmdLine.source); }
catch(Error::Syntax_error e)
{
	cerr << "Syntax Error! Quitting"
		<< endl;
	return 1;
}
// This is the main sampling loop. We
	will execute the source
// "nsamples" times and write the
	simulation results to
// standard out.
for( long i = 0; i < nsamples; i++ )
{
	tick_count2 = clock();
	bool fail = false;

	if( log_level & STARTUP ) cout
		<< "trial " << (i+1) << "
		" << nsamples << endl;

	// Save the volatile state
		information of the
	// machine before executing the
		source on the machine.
	m->save();

	// Execute the source
	try { m->run(); }
	catch(Error::Exec_error e)
	{
		cerr << "Execution Error
			! Quitting" << endl
			;
		return 1;
	}
	catch(Error::Crash_error e)
	{
		if( log_level & ERRORS )
			cout << "halt "
			<< e.msg << endl;
		fail = true;
	}

	// Restore the volatile elements
		of the machine state
	m->restore();

	if( log_level & STARTUP ) cout
		<< "result " << (fail?"F":
		"P") << endl;
	dataFile << (fail?"F":"P") << "\
		t";
	dataFile << double(clock() -
		tick_count2) /
		CLOCKS_PER_SEC << endl;
}
thetime = double(clock() - tick_count)
	/ CLOCKS_PER_SEC;
cout << "wallseconds " << thetime <<
	endl;
cout << "sampleseconds " << double(clock
	() - tick_count2) / CLOCKS_PER_SEC
		<< endl;
	return 0;
}
```

C.2.22 commandLine.cc

```
// commandLine.cxx
// Command line parser, default setter,
	dependency checker
// $Id: arq.cxx 6 2004-07-16 03:48:34Z awcross $

#include "include/utilities.hh"
#include "include/commandLine.hh"

using namespace std;

string machineToString(mType m)
{
	switch(m)
	{
		case CC: return string("
			classical");
		case CG: return string("clifford
			");
		case PCG: return string("planar"
			);
		case CIT: return string("iontrap
			");
		case NCG: return string("noisy
			clifford");
		case NPCG: return string("noisy
			planar");
		case NCIT: return string("noisy
			iontrap");
		default: return string("internal
			 error");
	}
}

// Set default values for the command line
// If the user just runs ARQ this is what will
// happen (probably nothing).
commandLine::commandLine()
{
	// set defaults
	machine = NCG;
	source = "";
	layout = "";
	parameters = "";
	samples = 1;
	verbosity = 1;
	datafile = "default.data";
	interactive = false;
}

// Check that we have enough information to run
	the simulation.
bool cmdLineDependencies(struct commandLine& cmd
	)
{
	if( !cmd.interactive )
	{
		// Source file required
		if( cmd.source.size() == 0 )
			return false;
		// Layout file required for
			planar simulations
		if( (cmd.machine == PCG || cmd.
			machine == CIT ||
			cmd.machine == NPCG || cmd.
				machine == NCIT) &&
			cmd.layout.size() == 0 )
				return false;
		// Require a parameter file for
			now, but we can FIX
		// this later by writing a
			default parameter file.
		if( cmd.parameters.size() == 0 )
			return false;
	}
	return true;
}

// Read the command line arguments into the
	commandLine structure,
// returning false if the command line doesn't
	look right.
bool parseCmdLine(struct commandLine& cmd, int
	argc, char *argv[])
```

```
{
        int loc = 1;
        if( argc == 2 )
        {
                string line(argv[1]);
                if( line == "--interactive" )
                    cmd.interactive = true;
                else cmd.interactive = false;
        }
        while( loc < argc && !cmd.interactive )
        {
                string line(argv[loc]);
                string s = Token(line,"=");
                if(( s == "machine" ) || ( s ==
                    "-m"))
                {
                        if( line == "cc" ) cmd.
                            machine = CC;
                        else if( line == "cg" )
                            cmd.machine = CG;
                        else if( line == "pcg" )
                            cmd.machine = PCG;
                        else if( line == "cit" )
                            cmd.machine = CIT;
                        else if( line == "ncg" )
                            cmd.machine = NCG;
                        else if( line == "npcg"
                            ) cmd.machine =
                            NPCG;
                        else if( line == "ncit"
                            ) cmd.machine =
                            NCIT;
                        else return false;
                } else if(( s == "source" )
                    || ( s == "-s")) cmd.
                    source = line;
                else if( (s == "layout") || ( s
                    == "-l" )) cmd.layout =
                    line;
                else if( (s == "parameters")
                    || ( s == "-p" )) cmd.
                    parameters = line;
                else if( (s == "datafile") || (
                    s == "-f" )) cmd.datafile
                    = line;
                else if( (s == "verbosity") || (
                    s == "-v"))
                {
                        if( !isInteger(line) )
                            return false;
                        cmd.verbosity = str2int(
                            line);
                }
                else if((s == "samples" ) || (s
                    == "-n"))
                {
                        if( !isInteger(line) )
                            return false;
                        cmd.samples = str2int(
                            line);
                }
                else return false;
                loc++;
        }
        return true;
}
```

C.2.23 commandLine.hh

```
// commandLine.hh
// Command line parser, default setter,
    dependency checker
// $Id: arq.cxx 6 2004-07-16 03:48:34Z awcross $

#include <string>
#include "utilities.hh"

using namespace std;

#ifndef _COMMANDLINE_
#define _COMMANDLINE_

// Valid machine types that we can simulate
typedef enum { CC,CG, PCG, CIT, NCG, NPCG, NCIT
    } mType;

// Structure to hold the parameters that ARQ
    needs to
// run a simulation.
struct commandLine
{
        mType machine;
        string source;
        string layout;
        string parameters;
        int samples;
        string datafile;
        int verbosity;
        bool interactive;
        commandLine();
};

// Check that we have enough information to run
    the simulation.
bool cmdLineDependencies(struct commandLine& cmd
    );

// Read the command line arguments into the
    commandLine structure,
// returning false if the command line doesn't
    look right.
bool parseCmdLine(struct commandLine& cmd, int
    argc, char *argv[]);

string machineToString(mType m);

#endif
```

C.2.24 globals.hh

```
/* globals.hh
   Contains all the variables that have to be
        accesse by multiple
   files.
   ddthaker@ucdavis.edu
   7/19/04
*/

extern int log_level;
```

C.2.25 htable.c

```
/*
  Hash table for the perfect hash functions.


  Created May 2004.                    Darshan
      Thaker    thaker@cs.ucdavis.edu

$Id: htable.c 6 2004-07-16 03:48:34Z awcross $

*/


#ifndef STANDARD
#include "include/standard.h"
#endif
#ifndef HTABLE
#include "include/htable.h"
#endif

/*
   This function creates a hash table for qbits
        and returns a pointer to
   the table. Null if not successful.

   Parameter passed is num, which is the number
        of entries expected to be
   stored in the table. Table size if power of
          2 greater than or equal
   to num.
*/
q_htable *q_htable_create (int num)
{
        int e, table_size, entry_size;
        q_htable *htable;

        /* Size of table has to be in power of
             2. So we calulate the
           power of two greater than or equal to
              num of entries */
        for (e=1; e<num; e<<=1);
```

```
        entry_size = (e-1) * sizeof (q_hentry);
        table_size = sizeof (q_htable);
        table_size += entry_size;
        htable = (q_htable *) calloc(1,
             table_size);

        if (htable == NULL) {
                printf("Can't get enough space
                     for q_htable\n");
                return NULL;
        }

        htable->size = e;

        return htable;
}



q_hentry *q_htable_find (q_htable *htable, ub1 *
     key, ub4 klen)
{
        int index;
        q_hentry *hentry;

        index = phash(key,klen);

        /* Look in chain if more than one key
             hashes to the same index */
        for (hentry = htable->table[index];
             hentry; hentry = hentry->next) {
                     if (memcmp(hentry->key, key
                        , klen) == 0)
                           break;
        }

        return hentry;
}




q_hentry* q_htable_add (q_htable *htable, ub1 *
     key, ub4 klen)
{
        int index;
        char a[2] = "XX";
        q_hentry *hentry, *tmp;

        index = phash(key,klen);
        hentry = htable->table[index];

        tmp = (q_hentry *)malloc(sizeof (
             q_hentry));
        memset(tmp, 0 ,sizeof(q_hentry));
        if (tmp == NULL) {
                printf("Can't allocate mem for
                      new entry\n");
                return NULL;
        }
        strncpy(tmp->key, key, klen);
        tmp->next = htable->table[index];
        htable->table[index] = tmp;
        return tmp;
}


void q_htable_print (q_htable *t)
{
        int i;
        q_hentry *hentry;

        for (i=0; i< t->size; i++) {
                printf("Entry: %d\n",i);
                for (hentry=t->table[i]; hentry
                      ; hentry=hentry->next)
                           printf(" %s ",hentry->
                               key);
                printf("\n");
        }
}


p_htable *p_htable_create (int num)
{
        int e, table_size, entry_size;
        p_htable *htable;
```

```
        /* Size of table has to be in power of
             2. So we calulate the
           power of two greater than or equal to
                  num of entries */
        for (e=1; e<num; e<<=1);

        entry_size = (e-1) * sizeof (p_hentry);
        table_size = sizeof (p_htable);
        table_size += entry_size;
        htable = (p_htable *) calloc(1,
             table_size);

        if (htable == NULL) {
                printf("Can't get enough space
                      for p_htable\n");
                return NULL;
        }

        htable->size = e;

        return htable;
}


p_hentry *p_htable_find (p_htable *htable, ub4
     key)
{
        int index;
        p_hentry *hentry;

        index = phash_i(key);
        index += 2;

        /* Look in chain if more than one key
             hashes to the same index */
        for (hentry = htable->table[index];
             hentry; hentry = hentry->next) {
                     //if (memcmp(hentry->key,
                            key, klen) == 0)
                     if(hentry->key == key)
                           break;
        }

        return hentry;
}




p_hentry *p_htable_add (p_htable *htable, ub4
     key)
{
        int index;
        char a[2] = "XX";
        p_hentry *hentry, *tmp;

        /* First do a lookup */
        index = phash_i(key);
        index +=2;
        hentry = htable->table[index];

        //printf("Index %d, key %s\n",index, key
            );
        for (hentry = htable->table[index];
             hentry; hentry = hentry->next) {
                  if((ub4)hentry->key == key) {
                          printf("Duplicate key \n
                              ");
                          return hentry;
                  }
        }

        /* No duplicates found so insert.
           hentry should be NULL */

        if (hentry == NULL) {
        /*XXX need a better malloc here -- this
             is resource intensive.*/
                tmp = (p_hentry *)malloc(sizeof
                      (p_hentry));
                memset(tmp, 0 ,sizeof(p_hentry))
                      ;
                if (tmp == NULL) {
                        printf("Can't allocate
                             mem for new entry\n
                             ");
                        return NULL;
                }

                tmp->key = key;
```

```
            tmp->next = htable->table[index
                ];
            htable->table[index] = tmp;
            return tmp;
    }

}


void p_htable_print (p_htable *t)
{
    int i;
    p_hentry *hentry;

    for (i=0; i< t->size; i++) {
        printf("Entry: %d\n",i);
        for (hentry=t->table[i]; hentry
            ; hentry=hentry->next)
                printf(" %d -> ",hentry
                    ->key);
        printf("\n");
    }
}
```

C.2.26 htable.h

```
/*
structs and functions related to hash tables.

Created - May 2004.   Darshan Thaker
    thaker@cs.ucdavis.edu
$Id: htable.h 6 2004-07-16 08:48:34Z awcross $

The Hash table for both qbits and physical
    locations have table entries
that are of type "q_hentry" and "p_hentry"
    respectively.

A qbit entry contains a pointer to its physical
    location. This pointer
is of type p_hentry. This allows us to move a
    qbit by simply
changing pointers.
*/

#include "standard.h"

struct qbit_hentry
{
    /*ub1 *key;       Key is the qbit name */
    char key[10];
    ub4  len;       /* Length of the key */
    ub4  hash;      /* generated hash */
    struct phy_hentry *pos;
    struct qbit_hentry *next;  /* Next
        hentry. Unused when the hash
        function
                            is a perfect
                            hash */
};


struct qbit_htable
{
    struct qbit_hentry *table[1];    /* Hash
        table */
    ub4     size;                    /* size
        of the table */
    struct qbit_hentry *free;        /* Used
        entries */
};

struct phy_hentry
{
    ub4  key;       /* Key is the qbit name
        */
    ub4  len;       /* Length of the key */
    ub4  hash;      /* generated hash */
    int  occupied;  /* anything here??
                    occupied = 1 if qbit
                        present */
    struct qbit_hentry *bit;
    struct phy_hentry *next;  /* Next hentry
        . Unused when the hash function
                            is a perfect
                            hash */
};

struct phy_htable
{
    struct phy_hentry *table[1];     /* Hash
        table */
    ub4     size;                /* size of the
        table */
    ub4     free;                /* Used entries
        */
};

typedef struct qbit_hentry q_hentry;
typedef struct phy_hentry  p_hentry;
typedef struct qbit_htable q_htable;
typedef struct phy_htable  p_htable;

q_htable *q_htable_create();
q_hentry *q_htable_find();
q_hentry *q_htable_add();
p_htable *p_htable_create();
void q_htable_print();
```

C.2.27 interactive.cc

```
// interactive.cxx
// ARQ's interactive mode
// Essentially a standalone interface to QState.
    cxx
// $Id: $

#include "include/interactive.hh"

using namespace std;

bool InteractivePrompt::hasId(string id)
{
   map<string, QState* >::iterator l;
   l = states.find(id);
   if( l != states.end() ) return true;
   return false;
}

InteractivePrompt::InteractivePrompt()
{
}

void InteractivePrompt::help(string op)
{
    if( op.size() == 0 )
    {
        cout << "Help topics: exit add rm new ls
            ket sgt rrt eoe gsf" << endl;
        cout << "                cnot cz h s x y z"
            << endl;
    }
    if( op == "quit" )
    {
        cout << "Use: quit|exit" << endl << endl
            ;
        cout << "The commands quit and exit
            return you to the shell." << endl;
    }
    else if( op == "add" )
    {
        cout << "Use: add <state_name>" << endl
            << endl;
        cout << "This appends a new qubit to the
            quantum state named" << endl;
        cout << "state_name. The new qubit is
            initialized to |0>." << endl;
    }
    else if( op == "rm" )
    {
        cout << "Use: rm <state_name> <
            qubit_location>" << endl << endl;
        cout << "This deletes the qubit in
            location qubit_location" << endl;
        cout << "from the quantum state named
            state_name. If that qubit is" <<
            endl;
        cout << "entangled with the reset of the
            system, then it is measured" <<
            endl;
        cout << "prior to removal. The outcome
            might not be deterministic." <<
            endl;
    }
```

```
    else if( op == "new" )
    {
        cout << "Use: new <state_name>" << endl
            << endl;
        cout << "Create a new quantum state
            object named state_name." << endl;
    }
    else if( op == "ls" )
    {
        cout << "Use: ls" << endl << endl;
        cout << "List all of the names of the
            quantum state objects." << endl;
    }
    else if( op == "ket" )
    {
    }
    else if( op == "sgt" )
    {
    }
    else if( op == "rrt" )
    {
    }
    else if( op == "eoc" )
    {
    }
    else if( op == "gsf" )
    {
    }
    else if( op == "cnot" )
    {
    }
    else if( op == "cz" )
    {
    }
    else if( op == "h" )
    {
    }
    else if( op == "s" )
    {
    }
    else if( op == "x" || op == "y" || op == "z"
        )
    {
    }
}

int InteractivePrompt::start(void)
{
    vector<Op> cmd;
    vector<Op>::iterator ci;

    quit = false;

    while( !quit )
    {
        cout << "arq> ";
        cin >> cmd;

        for( ci = cmd.begin(); ci != cmd.end();
            ci++ )
        {
            if( ci->opcode == "help" )
            {
                vector<string>::iterator i;
                if( ci->args.empty() )
                    help("");
                else
                    for( i = ci->args.begin(); i
                        != ci->args.end(); i
                        ++ ) help(*i);
            }
            else if( ci->opcode == "new" )
            {
                if( ci->args.size() > 0 )
                {
                    cout << ">> Creating a new
                        stabilizer state named
                        " << ci->args[0] <<
                        endl;
                    if( hasId(ci->args[0]) )
                        cout << ">> This name is
                            already in use,
                            too bad." << endl;
                    else
                        states[ci->args[0]] =
                            new QState();
                }
            }
            else if( ci->opcode == "ls" )
            {
                map<string,QState*>::iterator mi
                    ;
                cout << ">> Listing quantum
                    state labels" << endl;
                for( mi = states.begin(); mi !=
                    states.end(); mi++ )
                    cout << mi->first << " with
                        " << mi->second->size()
                        << " qubits" << endl;
            }
            else if( ci->opcode == "add" )
            {
                if( ci->args.size() > 0 && hasId
                    (ci->args[0]) )
                {
                    states[ci->args[0]]->add();
                    cout << ">> Added a new
                        qubit to \"" << ci->
                        args[0]
                        << "\" in state |0>"
                            << endl;
                }
            }
            else if( ci->opcode == "del" )
            {
                if( ci->args.size() > 1 )
                    if( hasId(ci->args[0]) &&
                        isInteger(ci->args[1]))
                    {
                        long loc = str2int(ci->
                            args[1]);
                        if( loc < 1 || loc >
                            states[ci->args
                            [0]]->size() )
                        {
                            cout << ">> The
                                index isn't in
                                range" << endl;
                        }
                        else
                        {
                            states[ci->args
                                [0]]->remove(
                                loc-1);
                            cout << ">> Deleted
                                qubit at
                                location " <<
                                loc
                                << " in \"" <<
                                    ci->args
                                    [0] << "\"
                                    " << endl;
                        }
                    }
            }
            else if( ci->opcode == "exit" )
            {
                quit = true;
                cout << ">> Quitting" << endl;
                break;
            }
            else if( ci->opcode == "ket" )
            {
                if( ci->args.size() > 0 && hasId
                    (ci->args[0]))
                {
                    cout << ">> Displaying the
                        state \"" << ci->args
                        [0]
                        << "\" as a ket" <<
                            endl;
                    cout << states[ci->args
                        [0]]->ket() << endl;
                }
            }
            else if( ci->opcode == "sgt" )
            {
                if( ci->args.size() > 0 && hasId
                    (ci->args[0]))
                {
                    cout << ">> Displaying the
                        state \"" << ci->args
                        [0]
                        << "\" as a tableau of
                            generators" <<
                            endl;
                    cout << states[ci->args
                        [0]]->state() << endl;
                }
            }
            else if( ci->opcode == "rrt" )
            {
                if( ci->args.size() > 0 && hasId
                    (ci->args[0]))
```

```
        {
            cout << ">> Displaying the
                state \"" << ci->args
                [0]
                << "\" as a row reduced
                    tableau" << endl;
            cout << states[ci->args
                [0]]->reducedState()
                << endl;
        }
    }
    else if( ci->opcode == "eoe" )
    {
        cout << ">> Entropy of
            entanglement" << endl;
        cout << ">> @____________ fruit
            by the foot" << endl;
    }
    else if( ci->opcode == "gsf" )
    {
        cout << ">> Displaying graph
            state form of stabilizer
            generators" << endl;
        cout << ">> <@((< fish" << endl;
    }
    else if( ci->opcode == "h" )
    {
        if( ci->args.size() > 1 )
        {
            if( hasId(ci->args[0]) &&
                isInteger(ci->args[1]))
            {
                long loc = str2int(ci->
                    args[1]);
                if( loc < 1 || loc >
                    states[ci->args
                    [0]]->size() )
                {
                    cout << ">> The
                        index isn't in
                        range" << endl;
                }
                else
                {
                    states[ci->args
                        [0]]->h(loc-1);
                    cout << ">> Applying
                         Hadamard to
                        qubit "
                        << loc << " of
                            state \""
                            << ci->
                            args[0]
                        << "\"" << endl
                            ;
                }
            }
        }
    }
    else if( ci->opcode == "cnot" )
    {
        cout << ">> Applying controlled-
            NOT to qubit " << endl;
    }
    else if( ci->opcode == "cz" )
    {
        cout << ">> Apply controlled-Z
            to qubit " << endl;
    }
    else if( ci->opcode == "s" )
    {
        if( ci->args.size() > 1 )
            if( hasId(ci->args[0]) &&
                isInteger(ci->args[1]))
            {
                long loc = str2int(ci->
                    args[1]);
                if( loc < 1 || loc >
                    states[ci->args
                    [0]]->size() )
                {
                    cout << ">> The
                        index isn't in
                        range" << endl;
                }
                else
                {
                    states[ci->args
                        [0]]->s(loc-1);
                    cout << ">> Applying
                         pi/4 gate to
                        qubit "
                        << loc << " of
                            state \""
                            << ci->
                            args[0]
                        << "\"" << endl
                            ;
                }
            }
    }
    else if( ci->opcode == "x" )
    {
        if( ci->args.size() > 1 )
            if( hasId(ci->args[0]) &&
                isInteger(ci->args[1]))
            {
                long loc = str2int(ci->
                    args[1]);
                if( loc < 1 || loc >
                    states[ci->args
                    [0]]->size() )
                {
                    cout << ">> The
                        index isn't in
                        range" << endl;
                }
                else
                {
                    states[ci->args
                        [0]]->gx(loc-1)
                        ;
                    cout << ">> Applying
                         Pauli X gate
                        to qubit "
                        << loc << " of
                            state \""
                            << ci->
                            args[0]
                        << "\"" << endl
                            ;
                }
            }
    }
    else if( ci->opcode == "y" )
    {
        if( ci->args.size() > 1 )
            if( hasId(ci->args[0]) &&
                isInteger(ci->args[1]))
            {
                long loc = str2int(ci->
                    args[1]);
                if( loc < 1 || loc >
                    states[ci->args
                    [0]]->size() )
                {
                    cout << ">> The
                        index isn't in
                        range" << endl;
                }
                else
                {
                    states[ci->args
                        [0]]->gy(loc-1)
                        ;
                    cout << ">> Applying
                         Pauli Y gate
                        to qubit "
                        << loc << " of
                            state \""
                            << ci->
                            args[0]
                        << "\"" << endl
                            ;
                }
            }
    }
    else if( ci->opcode == "z" )
    {
        if( ci->args.size() > 1 )
            if( hasId(ci->args[0]) &&
                isInteger(ci->args[1]))
            {
                long loc = str2int(ci->
                    args[1]);
                if( loc < 1 || loc >
                    states[ci->args
                    [0]]->size() )
                {
                    cout << ">> The
                        index isn't in
                        range" << endl;
                }
                else
```

```
                        {
                            states[ci->args
                                [0]]->gz(loc-1)
                                ;
                            cout << ">> Applying
                                Pauli Z gate
                                to qubit "
                                << loc << " of
                                    state \""
                                    << ci->
                                    args[0]
                                << "\"" << endl
                                ;
                        }
                    }
                }
            }
        }

    return 0;
}
```

C.2.28 interactive.hh

```
// interactive.hh
// interactive mode
// $Id: $

#ifndef __INTERACTIVE__
#define __INTERACTIVE__

#include <algorithm>
#include <string>
#include <map>

#include "utilities.hh"
#include "QState.hh"
#include "Op.hh"

using namespace std;

class InteractivePrompt
{
private:
    map<string, QState* > states;
    bool quit;

    bool hasId(string id);
    void help(string op);

public:
    InteractivePrompt();
    int start(void);
};

#endif
```

C.2.29 loglevels.hh

```
// loglevels.h
// Darshan Thaker and Andrew Cross

#define STARTUP 1
#define MESSAGE 2
#define CLOCK   4
#define GATES   8
#define MOVE    16
#define ERRORS  32
#define INIT    64
#define DEBUG   128

/* This file describes how the verbosity
    parameter of arq works.

Verbosity:    0 1 1 1 1 1 1 1

Bit 0 -> Startup information.
Bit 1 -> Message commands.
Bit 2 -> Clock
Bit 3 -> Gate Instructions (including
    measurement)
Bit 4 -> Move Instructions
Bit 5 -> Inst. Errors
Bit 6 -> Initialize instructions.
Bit 7 -> Debug info.

Each bit functions as a toggle. If the bit value
        is 0, the
function is turned off, and if it is 1, that log
        function is
turned on.

For example,
Verbosity = 7    : Startup, Message and Clock
Verbosity = 120  : Only Instructions related logs
    .
Verbosity = 128  : Only Debug
Verbosity = 255  : Everything.

Default value (verbosity = 57) */
```

C.2.30 lookupa.c

```
/*
lookupa.c, by Bob Jenkins, December 1996.  Same
      as lookup2.c
Use this code however you wish.  Public Domain.
      No warranty.
Source is http://burtleburtle.net/bob/c/lookupa.
      c
$Id: lookupa.c 6 2004-07-16 03:48:34Z awcross $
*/
#ifndef STANDARD
#include "include/standard.h"
#endif
#ifndef LOOKUPA
#include "include/lookupa.h"
#endif

/*
mix -- mix 3 32-bit values reversibly.
For every delta with one or two bit set, and the
      deltas of all three
  high bits or all three low bits, whether the
      original value of a,b,c
  is almost all zero or is uniformly distributed
      ,
* If mix() is run forward or backward, at least
      32 bits in a,b,c
  have at least 1/4 probability of changing.
* If mix() is run forward, every bit of c will
      change between 1/3 and
  2/3 of the time.  (Well, 22/100 and 78/100 for
      some 2-bit deltas.)
mix() was built out of 36 single-cycle latency
      instructions in a
  structure that could supported 2x parallelism
      , like so:
      a -= b;
      a -= c; x = (c>>13);
      b -= c; a ^= x;
      b -= a; x = (a<<8);
      c -= a; b ^= x;
      c -= b; x = (b>>13);
      ...
  Unfortunately, superscalar Pentiums and Sparcs
      can't take advantage
  of that parallelism.  They've also turned some
      of those single-cycle
  latency instructions into multi-cycle latency
      instructions.  Still,
  this is the fastest good hash I could find.
      There were about 2^^68
  to choose from.  I only looked at a billion or
      so.
*/
#define mix(a,b,c) \
{ \
  a -= b; a -= c; a ^= (c>>13); \
  b -= c; b -= a; b ^= (a<<8); \
  c -= a; c -= b; c ^= (b>>13); \
  a -= b; a -= c; a ^= (c>>12);  \
  b -= c; b -= a; b ^= (a<<16); \
  c -= a; c -= b; c ^= (b>>5); \
  a -= b; a -= c; a ^= (c>>3);  \
  b -= c; b -= a; b ^= (a<<10); \
  c -= a; c -= b; c ^= (b>>15); \
}

/*
lookup() -- hash a variable-length key into a
      32-bit value
  k       : the key (the unaligned variable-length
      array of bytes)
```

```
  len    : the length of the key, counting by
       bytes
  level : can be any 4-byte value
Returns a 32-bit value.  Every bit of the key
      affects every bit of
the return value.  Every 1-bit and 2-bit delta
      achieves avalanche.
About 6len+35 instructions.

The best hash table sizes are powers of 2.
      There is no need to do
mod a prime (mod is sooo slow!).  If you need
      less than 32 bits,
use a bitmask.  For example, if you need only
      10 bits, do
  h = (h & hashmask(10));
In which case, the hash table should have
      hashsize(10) elements.

If you are hashing n strings (ub1 **)k, do it
      like this:
  for (i=0, h=0; i<n; ++i) h = lookup( k[i], len
      [i], h);

By Bob Jenkins, 1996.  bob_jenkins@burtleburtle.
      net.  You may use this
code any way you wish, private, educational, or
      commercial.

See http://burtleburtle.net/bob/hash/evahash.
      html
Use for hash table lookup, or anything where one
      collision in 2^32 is
acceptable.  Do NOT use for cryptographic
      purposes.
*/

ub4 lookup( k, length, level)
register ub1 *k;          /* the key */
register ub4  length;     /* the length of the key
      */
register ub4  level;      /* the previous hash, or
      an arbitrary value */
{
   register ub4 a,b,c,len;

   /* Set up the internal state */
   len = length;
   a = b = 0x9e3779b9;  /* the golden ratio; an
        arbitrary value */
   c = level;           /* the previous hash
        value */

   /* handle most of the key */
   while (len >= 12)
   {
      a += (k[0] +((ub4)k[1]<<8) +((ub4)k
           [2]<<16) +((ub4)k[3]<<24));
      b += (k[4] +((ub4)k[5]<<8) +((ub4)k
           [6]<<16) +((ub4)k[7]<<24));
      c += (k[8] +((ub4)k[9]<<8) +((ub4)k
           [10]<<16)+((ub4)k[11]<<24));
      mix(a,b,c);
      k += 12; len -= 12;
   }

   /* handle the last 11 bytes */
   c += length;
   switch(len)              /* all the case
        statements fall through */
   {
   case 11: c+=((ub4)k[10]<<24);
   case 10: c+=((ub4)k[9]<<16);
   case 9 : c+=((ub4)k[8]<<8);
      /* the first byte of c is reserved for the
           length */
   case 8 : b+=((ub4)k[7]<<24);
   case 7 : b+=((ub4)k[6]<<16);
   case 6 : b+=((ub4)k[5]<<8);
   case 5 : b+=k[4];
   case 4 : a+=((ub4)k[3]<<24);
   case 3 : a+=((ub4)k[2]<<16);
   case 2 : a+=((ub4)k[1]<<8);
   case 1 : a+=k[0];
     /* case 0: nothing left to add */
   }
   mix(a,b,c);
   /* report the result */
   return c;
}
```

```
/*
mixc -- mixc 8 4-bit values as quickly and
      thoroughly as possible.
Repeating mix() three times achieves avalanche.
Repeating mix() four times eliminates all
      funnels and all
  characteristics stronger than 2^{-11}.
*/
#define mixc(a,b,c,d,e,f,g,h) \
{ \
   a^=b<<11; d+=a; b+=c; \
   b^=c>>2;  e+=b; c+=d; \
   c^=d<<8;  f+=c; d+=e; \
   d^=e>>16; g+=d; e+=f; \
   e^=f<<10; h+=e; f+=g; \
   f^=g>>4;  a+=f; g+=h; \
   g^=h<<8;  b+=g; h+=a; \
   h^=a>>9;  c+=h; a+=b; \
}

/*
checksum() -- hash a variable-length key into a
      256-bit value
  k      : the key (the unaligned variable-length
      array of bytes)
  len    : the length of the key, counting by
      bytes
  state : an array of CHECKSTATE 4-byte values
      (256 bits)
The state is the checksum.  Every bit of the key
      affects every bit of
the state.  There are no funnels.  About
      112+6.875len instructions.

If you are hashing n strings (ub1 **)k, do it
      like this:
  for (i=0; i<8; ++i) state[i] = 0x9e3779b9;
  for (i=0, h=0; i<n; ++i) checksum( k[i], len[i
      ], state);

(c) Bob Jenkins, 1996.  bob_jenkins@burtleburtle
      .net.  You may use this
code any way you wish, private, educational, or
      commercial, as long
as this whole comment accompanies it.

See http://burtleburtle.net/bob/hash/evahash.
      html
Use to detect changes between revisions of
      documents, assuming nobody
is trying to cause collisions.  Do NOT use for
      cryptography.
*/
void  checksum( k, len, state)
register ub1 *k;
register ub4  len;
register ub4 *state;
{
   register ub4 a,b,c,d,e,f,g,h,length;

   /* Use the length and level; add in the
        golden ratio. */
   length = len;
   a=state[0]; b=state[1]; c=state[2]; d=state
        [3];
   e=state[4]; f=state[5]; g=state[6]; h=state
        [7];

   /* handle most of the key */
   while (len >= 32)
   {
      a += (k[0] +(k[1]<<8) +(k[2]<<16) +(k
           [3]<<24));
      b += (k[4] +(k[5]<<8) +(k[6]<<16) +(k
           [7]<<24));
      c += (k[8] +(k[9]<<8) +(k[10]<<16)+(k
           [11]<<24));
      d += (k[12]+(k[13]<<8)+(k[14]<<16)+(k
           [15]<<24));
      e += (k[16]+(k[17]<<8)+(k[18]<<16)+(k
           [19]<<24));
      f += (k[20]+(k[21]<<8)+(k[22]<<16)+(k
           [23]<<24));
      g += (k[24]+(k[25]<<8)+(k[26]<<16)+(k
           [27]<<24));
      h += (k[28]+(k[29]<<8)+(k[30]<<16)+(k
           [31]<<24));
      mixc(a,b,c,d,e,f,g,h);
      mixc(a,b,c,d,e,f,g,h);
      mixc(a,b,c,d,e,f,g,h);
      mixc(a,b,c,d,e,f,g,h);
      k += 32; len -= 32;
```

```
    }

    /* handle the last 31 bytes */
    h += length;
    switch(len)
    {
    case 31: h+=(k[30]<<24);
    case 30: h+=(k[29]<<16);
    case 29: h+=(k[28]<<8);
    case 28: g+=(k[27]<<24);
    case 27: g+=(k[26]<<16);
    case 26: g+=(k[25]<<8);
    case 25: g+=k[24];
    case 24: f+=(k[23]<<24);
    case 23: f+=(k[22]<<16);
    case 22: f+=(k[21]<<8);
    case 21: f+=k[20];
    case 20: e+=(k[19]<<24);
    case 19: e+=(k[18]<<16);
    case 18: e+=(k[17]<<8);
    case 17: e+=k[16];
    case 16: d+=(k[15]<<24);
    case 15: d+=(k[14]<<16);
    case 14: d+=(k[13]<<8);
    case 13: d+=k[12];
    case 12: c+=(k[11]<<24);
    case 11: c+=(k[10]<<16);
    case 10: c+=(k[9]<<8);
    case 9 : c+=k[8];
    case 8 : b+=(k[7]<<24);
    case 7 : b+=(k[6]<<16);
    case 6 : b+=(k[5]<<8);
    case 5 : b+=k[4];
    case 4 : a+=(k[3]<<24);
    case 3 : a+=(k[2]<<16);
    case 2 : a+=(k[1]<<8);
    case 1 : a+=k[0];
    }
    mixc(a,b,c,d,e,f,g,h);
    mixc(a,b,c,d,e,f,g,h);
    mixc(a,b,c,d,e,f,g,h);
    mixc(a,b,c,d,e,f,g,h);

    /* report the result */
    state[0]=a; state[1]=b; state[2]=c; state[3]=
        d;
    state[4]=e; state[5]=f; state[6]=g; state[7]=
        h;
}
```

C.2.31 lookupa.h

```
/*
By Bob Jenkins, September 1996.
lookupa.h, a hash function for table lookup,
      same function as lookup.c.
Use this code in any way you wish.  Public
      Domain.  It has no warranty.
Source is http://burtleburtle.net/bob/c/lookupa.
      h
$Id: lookupa.h 6 2004-07-16 03:48:34Z awcross $
*/

#ifndef STANDARD
#include "standard.h"
#endif

#ifndef LOOKUPA
#define LOOKUPA

#define CHECKSTATE 8
#define hashsize(n) ((ub4)1<<(n))
#define hashmask(n) (hashsize(n)-1)

ub4  lookup(/*_ ub1 *k, ub4 length, ub4 level _
      */);
void checksum(/*_ ub1 *k, ub4 length, ub4 *state
      _*/);

#endif /* LOOKUPA */
```

C.2.32 noise.cc

```
// noise.cxx
// $Id: noise.cc 43 2004-07-19 08:59:55Z ike $
// Helper functions for depolarizing noise,
      bitflip noise, and
// phase flip noise. These can be replaced later
      by the noise
// implementation in ./temp/ when more general
      noise is needed.

#include "include/noise.hh"

// Depolarize qubit named q with probability p
Op depolarize(string q, double p)
{
        Op o;
        double r = ((double)rand())/((double)
            RAND_MAX);

        if( r < (1.0-p) )
        { // no error
        }
        else if( r < 1.0 - 2.0 * p/3.0 )
        {
                o.opcode = "x";
                o.args.push_back(q);
        }
        else if( r < 1.0 - p / 3.0 )
        {
                o.opcode = "y";
                o.args.push_back(q);
        }
        else
        {
                o.opcode = "z";
                o.args.push_back(q);
        }
        return o;
}

vector<Op> depolarize(string q1, string q2,
     double p)
{
        vector<Op> o;
        Op op;
        double r = ((double)rand())/((double)
            RAND_MAX);

        if( r < (1.0-p) )
        { // no error
        }
        else if( r < 1.0 - 14.0 * p/15.0 )
        {
                op.opcode = "x"; // IX
                op.args.push_back(q1);
                o.push_back(op);
        }
        else if( r < 1.0 - 13.0 * p/15.0 )
        {
                op.opcode = "y"; // IY
                op.args.push_back(q1);
                o.push_back(op);
        }
        else if( r < 1.0 - 12.0 * p/15.0 )
        {
                op.opcode = "z"; // IZ
                op.args.push_back(q1);
                o.push_back(op);
        }
        else if( r < 1.0 - 11.0 * p/15.0 )
        {
                op.opcode = "x"; // XI
                op.args.push_back(q2);
                o.push_back(op);
        }
        else if( r < 1.0 - 10.0 * p/15.0 )
        {
                op.opcode = "x"; // XX
                op.args.push_back(q1);
                o.push_back(op);
                op.args.pop_back();
                op.args.push_back(q2);
                o.push_back(op);
        }
        else if( r < 1.0 - 9.0 * p/15.0 )
        {
                op.opcode = "y"; // XY
                op.args.push_back(q1);
                o.push_back(op);
                op.opcode = "x";
                op.args.pop_back();
                op.args.push_back(q2);
                o.push_back(op);
        }
        else if( r < 1.0 - 8.0 * p/15.0 )
```

```
        {
                op.opcode = "z"; // XZ
                op.args.push_back(q1);
                o.push_back(op);
                op.opcode = "x";
                op.args.pop_back();
                op.args.push_back(q2);
                o.push_back(op);
        }
        else if( r < 1.0 - 7.0 * p/15.0 )
        {
                op.opcode = "y"; // YI
                op.args.push_back(q2);
                o.push_back(op);
        }
        else if( r < 1.0 - 6.0 * p/15.0 )
        {
                op.opcode = "x"; // YX
                op.args.push_back(q1);
                o.push_back(op);
                op.opcode = "y";
                op.args.pop_back();
                op.args.push_back(q2);
                o.push_back(op);
        }
        else if( r < 1.0 - 5.0 * p/15.0 )
        {
                op.opcode = "y"; // YY
                op.args.push_back(q1);
                o.push_back(op);
                op.args.pop_back();
                op.args.push_back(q2);
                o.push_back(op);
        }
        else if( r < 1.0 - 4.0 * p/15.0 )
        {
                op.opcode = "z"; // YZ
                op.args.push_back(q1);
                o.push_back(op);
                op.opcode = "y";
                op.args.pop_back();
                op.args.push_back(q2);
                o.push_back(op);
        }
        else if( r < 1.0 - 3.0 * p/15.0 )
        {
                op.opcode = "z"; // ZI
                op.args.push_back(q2);
                o.push_back(op);
        }
        else if( r < 1.0 - 2.0 * p/15.0 )
        {
                op.opcode = "x"; // ZX
                op.args.push_back(q1);
                o.push_back(op);
                op.opcode = "z";
                op.args.pop_back();
                op.args.push_back(q2);
                o.push_back(op);
        }
        else if( r < 1.0 - p/15.0 )
        {
                op.opcode = "y"; // ZY
                op.args.push_back(q1);
                o.push_back(op);
                op.opcode = "z";
                op.args.pop_back();
                op.args.push_back(q2);
                o.push_back(op);
        }
        else
        {
                op.opcode = "z"; // ZZ
                op.args.push_back(q1);
                o.push_back(op);
                op.args.pop_back();
                op.args.push_back(q2);
                o.push_back(op);
        }
        return o;
}

// bitflip error

Op bitflip(string q, double p)
{
        Op o;
        double r = ((double)rand())/((double)
            RAND_MAX);

        if( r < (1.0-p) )
        { // no error
```

```
        }
        else
        {
                o.opcode = "x";
                o.args.push_back(q);
        }
        return o;
}

vector<Op> bitflip(string q1, string q2, double
    p)
{
        vector<Op> o;
        Op op;
        double r = ((double)rand())/((double)
            RAND_MAX);

        if( r < (1.0-p) )
        { // no error
        }
        else if( r < 1.0 - 2.0 * p/3.0 )
        {
                op.opcode = "x"; // IX
                op.args.push_back(q1);
                o.push_back(op);
        }
        else if( r < 1.0 - p/3.0 )
        {
                op.opcode = "x"; // XI
                op.args.push_back(q2);
                o.push_back(op);
        }
        else
        {
                op.opcode = "x"; // XX
                op.args.push_back(q1);
                o.push_back(op);
                op.args.pop_back();
                op.args.push_back(q2);
                o.push_back(op);
        }
        return o;
}

// phaseflip error

Op phaseflip(string q, double p)
{
        Op o;
        double r = ((double)rand())/((double)
            RAND_MAX);

        if( r < (1.0-p) )
        { // no error (nop)
        }
        else
        {
                o.opcode = "z";
                o.args.push_back(q);
        }
        return o;
}

vector<Op> phaseflip(string q1, string q2,
    double p)
{
        vector<Op> o;
        Op op;
        double r = ((double)rand())/((double)
            RAND_MAX);

        if( r < (1.0-p) )
        { // no error
        }
        else if( r < 1.0 - 2.0 * p/3.0 )
        {
                op.opcode = "z"; // IZ
                op.args.push_back(q1);
                o.push_back(op);
        }
        else if( r < 1.0 - p/3.0 )
        {
                op.opcode = "z"; // ZI
                op.args.push_back(q2);
                o.push_back(op);
        }
        else
        {
                op.opcode = "z"; // ZZ
                op.args.push_back(q1);
                o.push_back(op);
                op.args.pop_back();
```

```
                op.args.push_back(q2);
                o.push_back(op);
        }
        return o;
}

// bitflip error with no pre-computed
    propagation through gates

Op bitflip_nopropagate(string q, double p)
{
        Op o;
        double r = ((double)rand())/((double)
            RAND_MAX);

        if( r < (1.0-p) )
        { // no error
        }
        else
        {
                o.opcode = "x";
                o.args.push_back(q);
        }
        return o;
}

vector<Op> bitflip_nopropagate(string q1, string
      q2, double p)
{
        vector<Op> o;
        Op op;

        op = bitflip_nopropagate(q1,p);
        o.push_back(op);

        op = bitflip_nopropagate(q2,p);
        o.push_back(op);
        return o;
}
```

C.2.33 noise.hh

```
// noise.hh
// $Id: noise.hh 43 2004-07-19 08:59:55Z ike $
// Helper functions for depolarizing noise,
     bitflip noise, and
// phase flip noise. These can be replaced later
     by the noise
// implementation in ./temp/ when more general
     noise is needed.

#ifndef __NOISE__
#define __NOISE__

#include <vector>
#include <string>
#include <cstdlib>
#include "Op.hh"

Op depolarize(string q, double p);

vector<Op> depolarize(string q1, string q2,
     double p);

Op bitflip(string q, double p);
vector<Op> bitflip(string q1, string q2, double
     p);

Op bitflip_nopropagate(string q, double p);
vector<Op> bitflip_nopropagate(string q1, string
      q2, double p);

Op phaseflip(string q, double p);
vector<Op> phaseflip(string q1, string q2,
     double p);

#endif
```

C.2.34 phash.c

```
/* Generated file. Please do NOT edit. Refer to
   perfect.c.                       --
        ddthaker@ucdavis.edu
*/

/* table for the mapping for the perfect hash */
#ifndef STANDARD
#include "include/standard.h"
#endif /* STANDARD */
#ifndef PHASH
#include "include/phash.h"
#endif /* PHASH */
#ifndef LOOKUPA
#include "include/lookupa.h"
#endif /* LOOKUPA */

/* small adjustments to _a_ to make values
     distinct */
ub1 tab[] = {
0,235,113,0,0,0,113,113,0,0,0,88,0,124,0,0,
113,183,0,0,0,0,0,220,82,113,0,220,40,0,125,0,
0,116,0,183,22,0,0,125,42,229,183,0,87,0,0,0,
0,0,235,125,183,183,235,116,165,183,113,0,120,
131,253,87,27,183,125,0,235,85,113,235,214,125,
60,0,214,0,124,131,87,124,235,89,183,7,131,131,
11,214,0,135,0,125,0,113,0,0,7,69,214,7,183,146,
237,145,0,183,235,184,0,232,0,87,128,87,132,146,
0,240,131,148,111,0,0,40,148,154,
};

/* The hash function */
ub4 phash(key, len)
char *key;
int   len;
{
  ub4 rsl, val = lookup(key, len, 0x9e3779b9);
  rsl = ((val>>25)^tab[val&0x7f]);
  return rsl;
}

/* Generated file. Please do NOT edit. Refer to
   perfect.c.                       --
        ddthaker@ucdavis.edu
*/

/* table for the mapping for the perfect hash */
#ifndef STANDARD
#include "standard.h"
#endif /* STANDARD */
#ifndef PHASH
#include "phash.h"
#endif /* PHASH */
#ifndef LOOKUPA
#include "lookupa.h"
#endif /* LOOKUPA */

/* small adjustments to _a_ to make values
     distinct */
ub1 tab_i[] = {
0,0,0,0,0,135,27,145,51,0,113,125,7,85,183,124,
11,22,0,40,11,146,32,17,148,132,0,0,0,0,0,0,
0,113,125,7,85,131,22,82,40,0,220,116,27,125,
85,22,87,124,131,40,74,32,0,0,0,87,120,42,26,
113,125,131,40,0,7,124,11,27,183,125,87,113,
124,27,74,32,61,142,69,131,146,135,145,111,60,
148,132,0,113,85,120,32,45,111,60,165,0,0,0,113,
92,142,97,131,40,88,11,146,26,159,152,55,183,8,
73,74,111,142,112,70,137,110,79,102,0,0,
};

/* The hash function */
ub4 phash_i(val)
ub4 val;
{
  ub4 a, b, rsl;



  b = (val & 0x7f);
  a = ((val << 18 ) >> 25);
  rsl = (a^tab[b]);
  return rsl;
}
```

C.2.35 phash.h

```
/* Generated file. Please do NOT edit. Refer to
   perfect.c.                       -- thaker@cs.
        ucdavis.edu
*/

/* Perfect hash definitions */
```

```
#ifndef STANDARD
#include "standard.h"
#endif /* STANDARD */
#ifndef PHASH
#define PHASH

extern ub1 tab[];
#define PHASHLEN 0x80  /* length of hash mapping
      table */
#define PHASHNKEYS 200  /* How many keys were
      hashed */
#define PHASHRANGE 256  /* Range any input might
      map to */
#define PHASHSALT 0x9e3779b9 /* internal,
      initialize normal hash */

ub4 phash();

#endif  /* PHASH */

/* Generated file. Please do NOT edit. Refer to
   perfect.c.                    -- thaker@cs.
        ucdavis.edu
*/

/* Perfect hash definitions */
#ifndef STANDARD
#include "standard.h"
#endif /* STANDARD */
#ifndef PHASH_I
#define PHASH_I

extern ub1 tab_i[];
#define PHASHLEN_I 0x80  /* length of hash
      mapping table */
#define PHASHNKEYS_I 162  /* How many keys were
      hashed */
#define PHASHRANGE_I 256  /* Range any input
      might map to */
#define PHASHSALT_I 0x9e3779b9 /* internal,
      initialize normal hash */

ub4 phash_i();

#endif  /* PHASH */
```

C.2.36 planarops.h

```
/*
   Reads layout and move instructions and moves
        qbits.

   Created May 2004.
        ddthaker@ucdavis.edu
$Id: planarops.h 6 2004-07-16 03:48:34Z awcross
    $
*/

//#define DEBUG

#ifndef STANDARD
#include "standard.h"
#endif

#ifndef HTABLE
#include "htable.h"
#endif

p_htable * create_p_table (int);
q_htable * create_q_table (int);

void init_position(const char *, int, p_htable
      *, q_htable*);
void add_loc(int read, p_htable *ptab);

// movement
void move_step (const char *, char, int,
      p_htable*, q_htable*);
void move (const char *, char, int, p_htable*,
      q_htable*, int);

// operations
q_hentry * find_in_qhtable(const char *,
      q_htable*);
p_hentry * find_in_phtable(int, p_htable*);
int get_loc (const char *, p_htable*, q_htable*)
    ;
char * get_name (int, p_htable*);
q_hentry *get_adj (const char *, char, p_htable
      *, q_htable*);
q_hentry *get_near (const char *, char, p_htable
      *, q_htable*);
q_hentry *get_dist (const char *, char, int,
      p_htable*, q_htable*);
```

C.2.37 planarops.c

```
// File: planarops.c
// $Id: planarops.c 6 2004-07-16 03:48:34Z
      awcross $

#ifndef PLANAROPS
#include "include/planarops.h"
#endif

p_htable * create_p_table (int TAB_SIZE)
{
        p_htable *ptab;
        ptab = (p_htable *) p_htable_create(
            TAB_SIZE);
        if (ptab == NULL) {
                fprintf(stderr,"P-Hash table ptr
                    is NULL\n");
                exit(1);
        }
        return ptab;
}



q_htable * create_q_table (int TAB_SIZE)
{
        q_htable *qtab;
        qtab = (q_htable *) q_htable_create(
            TAB_SIZE);
        if (qtab == NULL) {
                fprintf(stderr,"Q-Hash table ptr
                    is NULL\n");
                exit(1);
        }
        return qtab;
}



void add_loc(int read, p_htable *ptab)
{
        p_hentry *p_item;
        if( (p_item = (p_hentry *)p_htable_add (
            ptab,read)) == NULL)
        {
                fprintf(stderr,"Error inserting
                    . key %d\n",read);
                exit(1);
        }
}



// Return the qubit object with the given name
q_hentry * find_in_qhtable(const char *name,
     q_htable *qtab)
{
        q_hentry *q;
        int len;
        len = (ub4)strlen( (const char *)name);
        if( (q = q_htable_find (qtab,(ub1 *)name
            ,len)) == NULL) {
                fprintf(stderr,"Error Finding
                    key %s\n",(ub1 *)name);
                exit(1);
        }
        return q;
}



// Return the qubit object at the given location
p_hentry * find_in_phtable(int loc, p_htable *
     ptab)
```

```
{
        p_hentry * p;
        if( (p = (p_hentry *)p_htable_find(ptab,
             loc)) == NULL) {
                fprintf(stderr,"Error finding
                     the physical position\n");
                exit(1);
        }
        if (p->occupied == 1) return p;
        else return NULL;
}



/*
Set the initial position of a qbit.

Before this function is called, all physical
     locations
should have been initialized. But this is the
     first
time that a qbit is actually linked with its
physical position.
*/
void init_position(const char *name, int xy,
     p_htable *ptab, q_htable *qtab)
{
        q_hentry *q;
        p_hentry *p;
        int len;
        ub1 *key;

        len = (ub4)strlen( (char *)name);
        key = (ub1 *)name;

        if( (q = q_htable_add (qtab,key,len))
              == NULL) {
                printf("Error inserting. key %s\
                     n",key);
                exit(1);
        }

        if( (p = (p_hentry *)p_htable_find(ptab,
             xy)) == NULL) {
                fprintf(stderr,"Error finding
                     the physical position\n");
                exit(1);
        }
        q->pos = p;
        p->bit = q;
        p->occupied = 1;
}



//
//
// Move a qbit 'steps' positions in 'dir'
     Direction
//
void move_step (const char *name, char dir, int
     steps, p_htable *ptab, q_htable *qtab)
{
        q_hentry *q;
        p_hentry *p;
        int len, cur_xy, new_xy;
        int cur_x, cur_y;
        ub1 *key;

        len = (ub4)strlen( (char *)name);
        key = (ub1 *)name;

        if( (q = q_htable_find (qtab,key,len))
              == NULL) {
                fprintf(stderr,"Error Finding
                     key %s\n",key);
                exit(1);
        }

        cur_xy = q->pos->key;    // get current
             position
        switch(dir) {
                case 'N': new_xy = cur_xy +
                     steps;
                          break;

                case 'S': new_xy = cur_xy -
                     steps;
                          break;

                case 'W': cur_y = cur_xy%1000;
                          cur_x = cur_xy/1000;
                          new_xy = ((cur_x-steps
                               )*1000) + cur_y;;
                          break;

                case 'E': cur_y = cur_xy%1000;
                          cur_x = cur_xy/1000;
                          new_xy = ((cur_x+steps
                               )*1000) + cur_y;;
                          break;

                default : fprintf(stderr,"ERR:
                     Incorrect Direction\n");
                          break;
        }

        p = (p_hentry *)p_htable_find (ptab,
             new_xy);
        if (p->occupied != 1) {
                // Ok to move.

                q->pos->bit = NULL; // remove
                     from old location
                q->pos->occupied = 0;

                // Put in new location
                p->occupied = 1;
                q->pos = p;
                p->bit = q;

                //calculate_err(q);
        } else {
                fprintf(stderr,"FATAL : Moving %
                     s to an already full
                     location\n",name);
                exit(1);
        }

        //printf("Moved %s %c one step\n",name,
             dir);

}



//
// Depending on the DEBUG flag, the function
     moves a qbit
// either 1 step at a time, or jumps to the new
     location
//
void move (const char *name, char dir, int steps
     , p_htable *ptab, q_htable *qtab, int Debug
     )
{
        int i, debug_step;

        if (Debug) {
                // If debug ON then move one
                     position at a time
                debug_step = 1;
                for (i=1; i<=steps; i++) {
                        move_step (name, dir,
                             debug_step, ptab,
                             qtab);
                        fprintf(stderr,"Moved %s
                             %c by 1 step\n",
                             name,dir);
                }
        }
        else {
                // Else jump
                move_step (name, dir, steps,
                     ptab, qtab);
                //fprintf(stderr,"Moved %s %c by
                      %d steps\n",name,dir,
                      steps);
        }

}



//
// GET_LOC
// Return the Current Location of a Qubit in
     P_HTABLE format (i.e. integer).
//
int get_loc (const char *name, p_htable *ptab,
     q_htable *qtab)
{
        q_hentry *q;
        int len = (ub4)strlen( (char *)name);
```

```
        if( (q = q_htable_find (qtab,(ub1*)name,
            len)) == NULL)
        {
                fprintf(stderr,"Error Finding
                    key %s\n",(ub1*)name);
                exit(1);
        }
        return q->pos->key; // return position
}


//
// GET_NAME
// Return the Name of a Qubit at some location
//
char * get_name (int loc , p_htable *ptab)
{
        p_hentry *p;
        q_hentry *q;
        if((p = (p_hentry *)p_htable_find (ptab
            , loc)) == NULL)
        {
                fprintf(stderr,"Error Finding
                    location %d\n",loc);
                return NULL;
        }
        if (p->occupied == 1) return p->bit->key
            ;
        else return "no_qubit";
}



//
// GET_ADJ
// Return the adjacent qubit if it exists and is
// located in the specified direction ("N","S","
    E","W")
//
// Note:
//              returns the qubit itself if
    there is an empty space
//              in the specified direction.
//
q_hentry *get_adj (const char *name, char dir ,
    p_htable *ptab, q_htable *qtab)
{
        q_hentry *q, *q_adj;
        p_hentry *p;

        int len;
        ub4 loc_xy , adj_loc;     // location of
            the qubit
        ub1 *key;

        len = (ub4)strlen( (char *)name);
        key = (ub1 *)name;

        if( (q = q_htable_find (qtab,key,len))
             == NULL) {
                fprintf(stderr,"Error Finding
                    key %s\n",key);
                exit(1);
        }

        loc_xy = q->pos->key;    // get qubit's
            position
        switch(dir)
        {
                case 'N': adj_loc = loc_xy + 1;
                                break;

                case 'S': adj_loc = loc_xy - 1;
                                break;

                case 'W': adj_loc = (((loc_xy
                    /1000)-1)*1000) + (loc_xy
                     % 1000);
                                break;

                case 'E':       adj_loc = (((
                    loc_xy/1000)+1)*1000) + (
                    loc_xy % 1000);
                                break;

                default :       fprintf(stderr,"
                    ERR: Incorrect Direction\n"
                    );
                                break;
        }
```

```
        if((p = (p_hentry *)p_htable_find (ptab
            , adj_loc)) == NULL) {
                // Adjacent Location doesn't
                    exist
                //fprintf(stderr,"Error Finding
                    adj. Location %d\n",adj_loc
                    );
                return NULL;
        }
        else if (p->occupied == 1) {
                q_adj = p->bit;
                return q_adj;
        }
        return NULL;
}


//
// GET_NEAR
// Return the nearest qubit in a straight line
    if it exits
// and pointing to the specified direction ('N
    ','S','E','W')
//
// Note:
//      It doesn't fly over electrodes, so it
    only checks
//      each direction until it runs out of
    empty spaces.
//
q_hentry *get_near (const char *name, char dir,
    p_htable* ptab, q_htable *qtab)
{
        q_hentry *q, *q_near;
        p_hentry *p;

        int len;
        ub1 *key;
        int loc_xy , next_loc;

        len = (ub4)strlen( (char *)name);
        key = (ub1 *)name;

        if( (q = q_htable_find (qtab,key,len))
             == NULL) {
                fprintf(stderr,"Error Finding
                    key %s\n",key);
                exit(1);
        }

        loc_xy = q->pos->key;    // get qubit's
            position
        next_loc = loc_xy;

        while(1)
        {
                // Figure out the next location
                    to check.
                switch(dir)
                {
                        case 'N': next_loc =
                            next_loc + 1;
                                break;
                        case 'S': next_loc =
                            next_loc - 1;
                                break;
                        case 'E': next_loc = (((
                            next_loc/1000)+1)
                            *1000) + (next_loc
                            %1000);
                                break;
                        case 'W': next_loc = (((
                            next_loc/1000)-1)
                            *1000) + (next_loc
                            %1000);
                                break;
                        default:
                                fprintf(stderr,"
                                    ERR:
                                    Incorrect
                                    Direction\n
                                    ");
                                break;
                }
                if((p = (p_hentry *)
                    p_htable_find (ptab,
                    next_loc)) == NULL) {
                        // next location doesn't
                            exist
                        return NULL;
                }
                else if (p->occupied == 1) {
```

```
                        q_near = p->bit;
                        return q_near;
                }
        }

        // Should Never Get Here !!!
        return NULL;

} // END GET_NEAR



//
// GET_DIST:
// Retrurn the qubit a distance d from the name-
     qubit.
// Return NULL if nothing there or we hit an
     electrode.
//
q_hentry *get_dist (const char *name, char dir,
     int d, p_htable* ptab, q_htable *qtab)
{
        q_hentry *q, *dist_q;
        p_hentry *p;

        int len;
        ub1 *key;
        int loc_xy, new_loc;

        len = (ub4)strlen( (char *)name);
        key = (ub1 *)name;

        if( (q = q_htable_find (qtab,key,len))
             == NULL) {
                fprintf(stderr,"Error Finding
                     key %s\n",key);
                exit(1);
        }

        loc_xy = q->pos->key;
        switch(dir)
        {
                case 'N': new_loc = loc_xy + d;
                          break;
                case 'S': new_loc = loc_xy - d;
                          break;
                case 'E': new_loc = (((loc_xy
                     /1000)+d)*1000) + (loc_xy
                     %1000);
                          break;
                case 'W': new_loc = (((loc_xy
                     /1000)-d)*1000) + (loc_xy
                     %1000);
                          break;
                default:
                        fprintf(stderr,"ERR:
                             Incorrect Direction
                             \n");
                        break;
        }

        // See if the new location exists.
        if((p = (p_hentry *)p_htable_find (ptab
             , new_loc)) == NULL) return NULL;

        else if (p->occupied == 1)
        {
                dist_q = p->bit;
                return dist_q;
        }

        return NULL;
}
```

C.2.38 standard.h

```
/*
Standard definitions and types, Bob Jenkins
*/
#ifndef STANDARD
# define STANDARD
# ifndef STDIO
#  include <stdio.h>
#  define STDIO
# endif
# ifndef STDDEF
#  include <stddef.h>
#  define STDDEF
# endif
typedef  unsigned long long  ub8;
#define UB8MAXVAL 0xffffffffffffffffLL
#define UB8BITS 64
typedef    signed long long  sb8;
#define SB8MAXVAL 0x7fffffffffffffffLL
typedef  unsigned long  int  ub4;   /* unsigned
      4-byte quantities */
#define UB4MAXVAL 0xffffffff
typedef    signed long  int  sb4;
#define UB4BITS 32
#define SB4MAXVAL 0x7fffffff
typedef  unsigned short int  ub2;
#define UB2MAXVAL 0xffff
#define UB2BITS 16
typedef    signed short int  sb2;
#define SB2MAXVAL 0x7fff
typedef  unsigned       char ub1;
#define UB1MAXVAL 0xff
#define UB1BITS 8
typedef    signed       char sb1;   /* signed 1-
     byte quantities */
#define SB1MAXVAL 0x7f
typedef                 int  word;  /* fastest
     type available */

#define bis(target,mask)  ((target) |=  (mask))
#define bic(target,mask)  ((target) &= ~(mask))
#define bit(target,mask)  ((target) &   (mask))
#ifndef min
# define min(a,b) (((a)<(b)) ? (a) : (b))
#endif /* min */
#ifndef max
# define max(a,b) (((a)<(b)) ? (b) : (a))
#endif /* max */
#ifndef align
# define align(a) (((ub4)a+(sizeof(void *)-1))
     &(~(sizeof(void *)-1)))
#endif /* align */
#ifndef abs
# define abs(a)   (((a)>0) ? (a) : -(a))
#endif
#define TRUE  1
#define FALSE 0
#define SUCCESS 0  /* 1 on VAX */

#endif /* STANDARD */
```

C.2.39 utilities.cc

```
// utilities.cxx
// $Id: utilities.cc 16 2004-07-16 19:15:46Z
     awcross $
// Andrew Cross <awcross@mit.edu>
// useful utility functions

#include "include/utilities.hh"

using namespace std;

void Trim(std::string& str, const std::string&
     ChrsToTrim, int TrimDir)
{
    size_t startIndex = str.find_first_not_of(
         ChrsToTrim);
    if (startIndex == std::string::npos){str.
         erase(); return;}
    if (TrimDir < 2) str = str.substr(startIndex
         , str.size()-startIndex);
    if (TrimDir!=1) str = str.substr(0, str.
         find_last_not_of(ChrsToTrim) + 1);
}

inline void TrimRight(std::string& str, const
     std::string& ChrsToTrim)
{
    Trim(str, ChrsToTrim, 2);
}

inline void TrimLeft(std::string& str, const std
     ::string& ChrsToTrim)
{
    Trim(str, ChrsToTrim, 1);
}

std::string Token(std::string& str, const std::
     string& Chrs)
{
```

```
    string token;
    string::size_type pos = str.find_first_of(
        Chrs,0);
    if( pos == string::npos )
    {
        token = str;
        str = "";
        return token;
    }
    token = str.substr(0,pos);
    token.erase(pos,1);
    str.erase(0,pos+1);
    Trim(token);
    Trim(str);
    return token;
}

// Get the integer number representing the
    string name
long int str2int(std::string name)
{
    long int number = 0;
    int tmp;
    double j = 0.0;
    for (int i=(name.length()-1); i >= 0; i--)
    {
        tmp = static_cast<int>(name[i]) - 48;
        number = number + (tmp * static_cast<
            long int>(pow(10.0,j)));
        j++;
    }
    return number;
}

// Check if s represents a non-negative integer
bool isInteger(std::string s)
{
    string nums = "0123456789";
    for( int i=0; i < s.size(); i++ )
        if( nums.find(s[i],0) == string::npos )
            return false;
    return true;
}


//
string date2str(void)
{
        time_t lt = time(NULL);
        struct tm *timeptr;
        string tmp1,tmp2,date;

        timeptr = localtime(&lt);
        tmp1 = asctime(timeptr);

        // Put current Date in tmp2 but
        // with : in the middle.
        for (int i = 0; i <= 4; i++)
        {
                string t = Token(tmp1," ");
                tmp2.append(t);
        }
        for (int i = 0; i < 3; i++)
        {
                string t = Token(tmp2,":");
                date.append(t);
        }
        return date;
}
```

C.2.40 utilities.hh

```
// utilities.hh
// Andrew Cross <awcross@mit.edu>
// useful utility functions
// $Id: utilities.hh 6 2004-07-16 03:48:34Z
    awcross $

#include <string>
#include <iostream>
#include <sstream>
#include <math.h>
#include <time.h>

#ifndef __utilities__
#define __utilities__

void Trim(std::string& str, const std::string &
    ChrsToTrim = " \t\n\r", int TrimDir = 0);

inline void TrimRight(std::string& str, const
    std::string & ChrsToTrim = " \t\n\r");

inline void TrimLeft(std::string& str, const std
    ::string & ChrsToTrim = " \t\n\r");

std::string Token(std::string& str, const std::
    string & Chrs = " ");

template< class type>
inline std::string to_string( const type & value
    )
{
        std::ostringstream sout;
        sout << value;
        return sout.str();
}

long int str2int(std::string name);
bool isInteger(std::string s);
std::string date2str(void);

#endif
```

Bibliography

[ABO96] D. Aharonov and M. Ben-Or. Polynomial simulations of decohered quantum computers. *Proceedings of 37th Annual FOCS*, 1996.

[ABO99] D. Aharonov and M. Ben-Or. Fault-tolerant quantum computation with constant error rate. *Submitted to SIAM journal of computation*, 1999.

[AG04] S. Aaronson and D. Gottesman. Improved simulation of stabilizer circuits. *To appear in Phys. Rev. A*, 2004.

[Art91] M. Artin. *Algebra*. Prentice Hall, 1991.

[BCS03] S. Bettelli, T. Calarco, and L. Serafini. Toward an architecture for quantum programming. *Eur. Phys. Jour. D*, **25**(2), 181–200, 2003.

[BDS+03] M. D. Barrett, B. DeMarco, T. Schaetz, D. Leibfried, J. Britton, J. Chiaverini, W. M. Itano, B. Jelenkovic, J. D. Jost, C. Langer, T. Rosenband, and D. J. Wineland. Sympathetic cooling of $^{9}Be^{+}$ and $^{24}Mg^{+}$ for quantum logic. *Phys. Rev. A*, **68**(4), 42302, 2003.

[Bla04] R. Blatt. Trapped-ion quantum information processing. Talk presented at the MIT Quantum Information Processing Seminar, 2004. November 29th.

[CDB+95] C.Monroe, D.M.Meekhof, B.E.King, W.M.Itano, and D.J. Wineland. Demonstration of a fundamental quantum logic gate. *Physical Review Letters*, **75**(25), 4714–4717, 1995.

[COI+03] D. Copsey, M. Oskin, F. Impens, T. Metodiev, A. Cross, F. Chong, I. Chuang, and J. Kubiatowicz. Toward a scalable, silicon-based quantum computing architecture. *Journal of Selected Topics in Quantum Electronics*, **9**(6), 1552–1569, 2003.

[CS96] A. R. Calderbank and P. W. Shor. Good quantum error-correcting codes exist. *Phys. Rev. A*, **54**(2), 1098–1106, 1996.

[CSB+04] M. Barrett J. Chiaverini, T. Schaetz, J. Britton, W. M. Itano, J. D. Jost, E. Knill, C. Langer, D. Leibfried, R. Ozeri, and D. J. Wineland. Deterministic quantum teleportation of atomic qubits. *Nature*, **429**, 2004.

[CZ95] J. I. Cirac and P. Zoller. Quantum computations with cold trapped ions. *Phys. Rev. Let.*, **74**(20), 1995.

[DAKJ+01] D.Kielpinski, A.Ben-Kish, J.Britton, V.Meyer, M.A.Rowe, C.A.Sackett, W.M.Itano, C.Monroe, and D.J.Wineland. Recent results in trapped-ion quantum computing at nist. *Experimental Implementation of Quantum Computation (Sydney, 2001)*, 2001.

[DDV+03] D.Leibfried, B. DeMarco, V.Meyer, D.Lucas, M.Barrett, J.Britton, W.M.Itano, B.Jelenkovic, C.Langer, T. Rosenband, and D.J.Wineland. Experimental demonstration of a robust, high-fidelity geometric two ion-qubit phase gate. *Nature*, **422**, 412–415, 2003.

[Div00] D. Divincenzo. The physical implementation of quantum computation. *Fortschr. Phys.*, **48**, 771–783, 2000.

[DM03] J. Dehaene and B. De Moor. The clifford group, stabilizer states, and linear and quadratic operations over $gf(2)$. *Phys. Rev. A*, **68**, 42318, 2003.

[DMW+98] D.J.Wineland, C. Monroe, W.M.Itano, D. Liebfried, B.E. King, and D.M.Meekhof. Experimental issues in coherent quantum-state manipulation of trapped atomic ions. *Journal of Research of the National Institute of Standards and Technology*, **103**(3), 259–328, 1998.

[GBP97] M. Grassl, T. Beth, and T. Pellizzari. Codes for the quantum erasure channel. *Phys. Rev. A*, **56**(1), 33–38, 1997.

[gcc04] Gnu compiler collection. Free Software Foundation, 2004. http://gcc.gnu.org.

[Got97] D. Gottesman. *Stabilizer codes and quantum error correction.* PhD dissertation, Caltech, 1997. arXive e-print quant-ph/9705052.

[Got98a] D. Gottesman. The heisenberg representation of quantum computers. *Unpublished*, 1998.

[Got98b] D. Gottesman. A theory of fault-tolerant quantum computation. *Phys. Rev. A*, **57**(1), 127–137, 1998.

[Got00] D. Gottesman. Fault-tolerant quantum computation with local gates. *J. Mod. Optics*, **47**, 333–345, 2000.

[Got02] D. Gottesman. Beyond the divincenzo criteria: Requirements and desiderata for fault-tolerance. Talk presented at NANO2002 Workshop II: IPAM Workshop on quantum computing, 2002.

[Gro97] L. K. Grover. Quantum mechanics helps in searching for a needle in a haystack. *Phys. Rev. Let.*, **79**, 325, 1997.

[Hal02] S. Hallgren. Polynomial-time quantum algorithms for pell's equation and the principal ideal problem. In *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing*, pp. 653–658, 2002.

[HRC02] A. Harrow, B. Recht, and I. Chuang. Efficient discrete approximations of quantum gates. *J. Math. Phys.*, **43**, 4445, 2002.

[HRTS03] S. Hallgren, A. Russell, and A. Ta-Shma. The hidden subgroup problem and quantum compuation using group representations. *SIAM J. Comput.*, **32**(4), 916–934, 2003.

[Hua85] S. Huang. Height-balanced trees of order (β, γ, δ). *ACM Transactions on Database Systems*, **10**(2), 261–284, June 1985.

[Imp03] F. Impens. *Fine-Grained Fault-Tolerance.* SM thesis, Massachusetts Institute of Technology, 2003.

[KKM+00] D. Kielpinski, B. King, C. Myatt, C. Sackett, Q. Turchette, W. Itano, C. Monroe, and D. Wineland. Sympathetic cooling of trapped ions for quantum logic. *Phys. Rev. A*, **61**(3), 32310, 2000.

[KL96] E. Knill and R. Laflamme. Assumptions for fault-tolerant quantum computing. report LAUR-96-2718, LANL, 1996.

[KLZ98] E. Knill, R. Laflamme, and W. Zurek. Resilient quantum computation: Error models and thresholds. *Science*, **279**(5349), 1998.

[KMW02] D. Kielpinski, C. Monroe, and D. J. Wineland. Architecture for a large-scale ion-trap quantum computer. *Nature*, **417**, 709–711, 2002.

[Kni04] E. Knill. Quantum computing with very noisy devices. *Unpublished*, 2004.

[KWM+98] B. King, C. Wood, C. Myatt, Q. Turchette, D. Leibfried, W. Itano, C. Monroe, and D. Wineland. Cooling the collective motion of trapped ions to initialize a quantum register. *Physical Review Letters*, **81**, 1525–1528, 1998.

[Lin68] A. Lindenmayer. Mathematical models for cellular interaction in development. *Journal of Theoretical Biology*, **18**, 280–315, 1968.

[LMBK+04] D. Leibfried, M.D.Barrett, A. Ben-Kish, J. Britton, J. Chiaverini, B. DeMarco, W.M.Itano, B. Jelenkovic, J.D.Jost, C.Langer, D.Lucas, V.Meter, T.Rosenband, M.A.Rowe, T. Schaetz, and D.J.Wineland. Building blocks for a scalable quantum information processor based on trapped ions. In *Laser spectroscopy: proceedings of the XVI International Conference, Palm Cove, Queensland, Australia, 13-18 July, 2003*, 2004.

[MCT+04] T. Metodiev, A. Cross, D. Thaker, K. Brown, D. Copsey, F. Chong, and I. Chuang. Preliminary results on simulating a scalable fault-tolerant ion-trap system for quantum computation. *Presented at the 3rd workshop on non-silicon computing*, 2004.

[MTC+05a] T. Metodiev, D. Thaker, A. Cross, F. Chong, and I. Chuang. A general purpose architectural layout for arbitrary quantum computations. *Submitted to the 2005 SPIE Defense and Security Symposium*, 2005.

[MTC+05b] T. Metodiev, D. Thaker, A. Cross, F. Chong, and I. Chuang. A quantum fpga architecture: building scalable, fault-tolerant quantum computing with near-term technologies. *Submitted to the 32nd Annual International Symposium on Computer Architecture*, 2005.

[MWD+04] M.J. Madsen, W.K.Hensinger, D.Stick, J.A.Rabchuk, and C.Monroe. Planar ion trap geometry for microfabrication. *Applied Physics B-Lasers and Optics*, **78**(5), 639–651, 2004.

[NC97] M. A. Nielsen and I. L. Chuang. Programmable quantum gate arrays. *Physical Review Letters*, **79**(2), 321–324, 1997.

[NC00] M. Nielsen and I. Chuang. *Quantum computation and quantum information.* Cambridge University Press, Cambridge, England, 2000.

[NDR+99] H. C. Nagerl, D.Leibfried, H. Rohde, G. Thalhammer, J.Eschner, F. Schmidt-Kaler, and R. Blatt. Laser addressing of individual ions in a linear ion trap. *Physical Review A*, **60**(1), 1999.

[PH98] D. Patterson and J. Hennessy. *Computer organization and design.* Morgan Kaufmann, 1998.

[Pre01] J. Preskill. Fault-tolerant quantum computation. In H. Lo, S. Popescu, and T. Spiller, editors, *Introduction to quantum computation and information.* World Scientific Publishing Company, 2001. arXive e-print quant-ph/9712048.

[pyt04] Python. Python Software Foundation, 2004. http://www.python.org.

[RBKD+02] M. A. Rowe, A. Ben-Kish, B. DeMarco, D. Liebfried, V. Meyer, J. Beall, J. Britton, J. Hughes, W.M.Itano, B. Jelenkovic, C.Langer, T. Rosenband, and D.J.Wineland. Transport of quantum states and separation of ions in a dual rf ion trap. *Quantum Information and Computation*, **2**(4), 2002.

[Rei04] B. Reichardt. Improved ancilla preparation scheme increases fault-tolerant threshold. *Unpublished*, 2004.

[RHR+04] M. Riebe, H. Haffner, C.F. Roos, W. Hansel, J. Benhelm, G.P.T. Lancaster, T.W. Korber, C. Becher, F. Schmidt-Kaler, D.F.V. James, and R. Blatt. Deterministic quantum teleportation with atoms. *Nature*, **429**(6993), 734–737, 2004.

[RLR+04] C. Roos, G. Lancaster, M. Riebe, H. Haffner, W. Hansel, S. Gulde, C. Becher, J. Eschner, F. Schmidt-Kaler, and R. Blatt. Bell states of atoms with ultralong lifetimes and their tomographic state analysis. *Physical Review Letters*, **92**(22), 220402, 2004.

[SB02] M. Sasura and V. Buzek. Cold trapped ions as quantum information processors. *Journal of Modern Optics*, **49**(10), 2002.

[SCA+04] K. Svore, A. Cross, A. Aho, I. Chuang, and I. Markov. Toward a software architecture for quantum computing design tools. In *Proc. Quant. Prog. Lang.*, pp. 145–162, 2004.

[Sho94] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science, Proceedings*, pp. 124–134. IEEE, IEEE Press, 1994.

[Sho95] P. W. Shor. Scheme for reducing decoherence in quantum computer memory. *Physical Review A*, **52**(4), 2493–2496, 1995.

[Sho96] P. Shor. Fault-tolerant quantum computation. In *37th Annual Symposium on Foundations of Computer Science, Proceedings*, pp. 56–65, 1996. arXive e-print quant-ph/9605011.

[SKHG+03] F. Schmidt-Kaler, H. Haffner, S. Gulde, M. Riebe, G.P.T. Lancaster, T. Deuschle, C. Becher, W. Hansel, J. Eschner, C.F. Roos, and R. Blatt. How to realize a universal quantum gate with trapped ions. *Applied Physics B-Lasers and Optics*, **77**(8), 789–796, 2003.

[SKHR+03] F. Schmidt-Kaler, H. Haffner, M. Riebe, S. Gulde, G.P.T. Lancaster, T. Deuschle, C. Becher, C.F. Roos, J. Eschner, and R. Blatt. Realization of the cirac-zoller controlled-not quantum gate. *Nature*, **422**(6930), 408–411, 2003.

[SM99] A. Sorensen and K. Molmer. Quantum computation with ions in thermal motion. *Physical Review Letters*, **82**(9), 1971–1974, 1999.

[SM00] A. Sorensen and K. Molmer. Entanglement and quantum computation with ions in thermal motion. *Physical Review A*, **62**(2), 2000.

[STD04] K. Svore, B. Terhal, and D. DiVincenzo. Local fault-tolerant quantum computation. *Unpublished*, 2004.

[Ste96] A. Steane. Multiple particle interference and quantum error correction. In *Proceedings of the Royal Society of London Series A-Mathematical Physical and Engineering Sciences*, Volume 452, pp. 2551–2577, 1996. arXive e-print quant-ph/9601029.

[Ste97] A. Steane. Active stabilization, quantum computation and quantum state synthesis. *Physical Review Letters*, **78**, 2252–2255, 1997.

[Ste98] A. Steane. Space, time, parallelism, and noise requirements for reliable quantum computing. *Fortschritte der Physik*, **46**, 443–458, 1998.

[Ste99] A. Steane. Efficient fault-tolerant quantum computing. *Nature*, **399**(6732), 124–126, 1999.

[Ste02] A. Steane. Quantum computer architecture for fast entropy extraction. *Quantum Information and Computation*, **2**(4), 2002.

[Ste03a] A. Steane. Realistic fast quantum gates with hot trapped ions. *Phys. Rev. A*, **67**, 62318, 2003.

[Ste03b] A. M. Steane. Overhead and noise threshold of fault-tolerant quantum error correction. *Unpublished*, 2003.

[Ste04] A. Steane. How to build a 300 bit, 1 gop quantum computer. *Unpublished*, 2004.

[TB04] B. Terhal and G. Burkard. Fault-tolerant quantum computation for local non-markovian noise. *Unpublished*, 2004.

[VHP04] S. Virmani, S. Huelga, and M. Plenio. Classical simulatability, entanglement breaking, and quantum computation thresholds. *Unpublished*, 2004.

[Vid03] G. Vidal. Efficient classical simulation of slightly entangled quantum computations. *Physical Review Letters*, **91**(14), 147902, 2003.

[von56] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies*, pp. 328–378. Princeton University Press, 1956.

[WC63] S. Winograd and J. D. Cowan. *Reliable computation in the presence of noise.* MIT Press, 1963.

[WDM+04] W.K.Hensinger, D.Stick, M.J.Madsen, M.Acton, D.Hucul, R.Kohn, K.Schwab, J.A.Schwab, J.A.Rabchuk, and C.Monroe. Scalable semiconductor ion traps and trap geometries for complex shuttling operations. Poster presented at the Workshop on Trapped-Ion Quantum Computing at U. Michigan, 2004. http://monroelab2.physics.lsa.umich.edu/TIQCworkshop/.

[WH00] D. Wineland and T. Heinrichs. Ion trap approaches to quantum information processing and quantum computing. *A Quantum Information Science and Technology Roadmap*, 2000.

[Wig03] S. Wiggins. *Introduction to Applied Nonlinear Dynamical Systems and Chaos.* Springer-Verlag, 2003.

[ZLC00] X. Zhou, D. Leung, and I. Chuang. Methodology for quantum logic gate construction. *Physical Review A*, **62**(5), 2000.

[Zur81] W. Zurek. Pointer basis of quantum apparatus: Into what mixture does the wave packet collapse? *Physical Review D*, **24**(6), 1516–1525, 1981.

[Zur82] W. Zurek. Environment-induced superselection rules. *Physical Review D*, **26**(8), 1862–1880, 1982.