Investigating Quantum Speedups through Numerical Simulations

Thesis by
Shannon Wang

In Partial Fulfillment of the Requirements for the
degree of
Bachelor of Science in Physics

Caltech

CALIFORNIA INSTITUTE OF TECHNOLOGY
Pasadena, California

2017
Defended May 23, 2017

© 2017

Shannon Wang
ORCID: 0000-0003-0585-6556

ACKNOWLEDGEMENTS

ABSTRACT

It has been recently noted in a paper by Brandão *et al.* that the structure of a linear program in a classical semidefinite programming algorithm lends itself to quantization, such that the classical algorithm may experience a quantum speedup if the step of solving a linear program is replaced with the preparation of a Gibbs state of classical Hamiltonian on a quantum computer, where the Hamiltonian is given by a linear combination of the semidefinite program's constraint matrices. The quantum speedup would be exponential if the complexity of the Gibbs sampler used to execute the update step is polynomial in system size. The Gibbs samplers with explicitly defined runtimes are exponential in system size; however, while the quantum Metropolis sampling algorithm by Temme *et al.* does not have a runtime bounded explicitly in system size, the algorithm heuristically runs in polylogarithmic time. Since the inverse spectral gap of the quantum Metropolis map varies inversely with the running time of the algorithm, we simulate the behavior of the quantum Metropolis map's spectral gap as a function of system size and row sparsity. We also examine how different definitions of fixed row sparsity affect the spectral gap's behavior when the system size is increased linearly. While more numerical evidence is needed to draw a definitive conclusion, the current results appear to indicate that for system sizes ranging from three to ten qubits, if fixed row sparsity is defined as a fixed polynomial function of the system size, then the quantum Metropolis spectral gap behaves as a polynomial function of system size.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

NOMENCLATURE

Affine function. A function composed of a linear function followed by its translation..

Completely positive map. A mapping describing quantum evolution by mapping from a set of density matrices onto itself..

Frustration free system. A system whose Hamiltonian can be written as the sum of terms whose ground states match that of the Hamiltonian..

Hilbert space. The configuration space of a quantum system..

Phase space. A multidimensional space that contains all of the particle's possible states, which are characterized by position and momentum..

Positive semidefinite matrix. A Hermitian matrix with nonzero eigenvalues..

Slater's condition. A sufficient condition that a primal problem must satisfy if strong duality is to hold. The condition holds if the primal problem is strictly feasible; the weak form of the condition allows a relaxation of the strictly feasible rule if the function in question is affine..

Sparsity pattern. An explicitly defined pattern of nonzero elements in a matrix..

Spectral gap. The largest difference between the eigenvalues of the map..

Spectral norm. The square root of the maximum eigenvalue of a matrix..

Stochastic map. A completely positive map that also preserves the trace..

Strong duality. A condition under which the duality gap is zero..

Wavefunction. A complex-valued function that describes the quantum state of a system..

C h a p t e r   1

INTRODUCTION

The concept of quantum computers was born in 1981, when Feynman presented the difficulties of simulating quantum systems with classical approximations. In his seminal lecture on simulating physics with computers, he posed a question to physicists: is it possible to build a computer in which the elements required to simulate a physical system is proportional to the space-time volume of the system? Such a computer wouldn't be classical – quantum mechanics is inherently probabilistic, but simulating probabilities for large physical systems is an intractable problem for classical computers. If a classical computer is asked to calculate all of the possible configurations of R particles for N points in space, then the computer must be able to hold N^R configurations – something that cannot be done by a computer of order N. For realistic physical systems, the number of particles is on the same order of the number of points in space – in such cases, the computer must hold N^N elements. Feynman concluded that if a polynomial increase in the size of a physical system results in an exponential increase of required computing elements, then a classical computer can't efficiently simulate quantum physics by computing the evolution of a wavefunction [1]. Could a classical computer imitate the behavior of quantum mechanics by directly generating the probabilities encountered in nature? The probability of finding a classical particle at a certain point in the phase space is characterized by the phase-space probability distribution, which is non-negative and yields one when integrated over the phase space. The counterpart to the classical phase-space probability distribution in the quantum domain is the Wigner quasiprobability distribution. The Wigner function gives the probability of locating a particle at x when it is integrated over momentum, and it gives the momentum distribution when it is integrated over position [2]. However, the results of the Wigner quasiprobability distribution cannot be interpreted as probabilities, because the Wigner function admits negative values – how would negative probabilities be interpreted [3]?

The question then became: is there a way to circumvent the negative probabilities? Can the Wigner function's probabilities serve as quantum mechanical probabilities? Feynman proved with a two-photon correlation experiment how probabilities calculated by classical means do not agree with the probabilities given by quantum

mechanical formulas, demonstrating that classical systems cannot accurately imitate quantum mechanics [1]. Thus a quantum computer becomes very desirable – if a quantum system can directly replicate the circumstances required to generate the probabilities relevant to the problem at hand, then the probabilities will always be non-negative, and the problem of the exponential explosion of the Hilbert space with the system size can be circumvented. Feynman hypothesized that there must exist a class of quantum operators that when locally coupled with one another can simulate the Hamiltonian of any discrete quantum mechanical system with a finite number of degrees of freedom. In 1996, Lloyd verified Feynman's hypothesis – he demonstrated that the local Hamiltonian evolution of a quantum many-body system can be efficiently simulated on quantum computers. Since local Hamiltonians can be written as the sum of operators that operate on local Hilbert spaces, the time evolution operator can be broken up into local time evolution operators, which can then be simulated by discretizing time into small slices. The total number of operations required to simulate a quantum system within a given error is then proportional to the system size [4]. This was heartening, because most of the classical attempts at solving the many-body Schrödinger equation, such as perturbation theory, density functional theory, and the Hartree-Fock method, only yield approximate solutions limited to describing weak interaction systems. Yet many interesting problems in quantum chemistry, condensed matter physics, and high energy physics lie in strong interaction systems. Fortunately, spin systems of interest, such as the Ising and Heisenberg models, as well as both strong and weak interaction systems are local systems; indeed, any system that obeys the laws of special and general relativity is local [4]. As the advent of quantum computing nears, the hope that these problems will no longer be computationally intractable continues to grow. Lloyd posited towards the end of his paper that while a quantum computer operating on only three to four qubits is too small to solve classically intractable problems, it is still sufficient to verify quantum computing hypotheses. In the absence of quantum computers, we can only classically simulate a quantum algorithm's running time on a realistic quantum computer, and thus we are still limited by the exponentially growing Hilbert space – thus, we limit our analysis to systems ranging from three to eleven qubits. Nevertheless, a prediction of a quantum algorithm's behavior on a small-scale quantum computer should be indicative of how the algorithm will behave for larger systems.

1.1 Applications to Traditional Computing

Up until now, we have limited our discussion of the practical applications of quantum computers to simulating quantum physics. However, the influx of quantum algorithms – namely, algorithms that run on quantum computers – has piqued interest in applying quantum computing techniques to problems in classical computer science. One class of algorithms of particular interest to our research is the quantum Gibbs sampler – a class of quantum Markov chain Monte Carlo algorithm that allows random samples to be generated from arbitrary distributions without calculating the probability density function [5]. Gibbs sampling is primarily used in physics to find a statistical ensemble's Gibbs state, or the system's equilibrium distribution that remains stationary under further evolution. Since it is a statistical technique, it can be utilized in any statistical context, and thus plays a role in solving convex optimization problems. Convex optimization is a category of problems that can be solved by minimizing a convex function over a convex set – practical applications of convex optimization range from operations research to communications to control systems to finance. We limit ourselves to discussing the application of Gibbs sampling to semidefinite programming, a subfield of convex optimization that minimizes a linear function over the cone defined by a linear combination of positive semidefinite matrices with coefficients that add up to one.

A semidefinite programming algorithm that may be quantized through quantum Gibbs sampling is Arora's and Kale's primal-dual approach that utilizes the matrix multiplicative weights update method [6]. The matrix multiplicative weights update step calculates a matrix exponential and uses it to compute a density matrix, which is then passed as the input of an auxiliary algorithm called **ORACLE**. The auxiliary algorithm **ORACLE** checks whether the candidate primal solution of the semidefinite program violates the primal constraints and outputs a vector **y**, which will be used in the update step as well as in the attempt to find a dual solution. Brandão *et al.* proved in a recent paper that the output of **ORACLE** can be interpreted as the Gibbs state of a Hamiltonian that can be expressed as a linear combination of the constraint matrices, thus introducing the connection between quantum Gibbs sampling and a quantum version of a semidefinite program solver. [7]. Known quantum Gibbs samplers can be separated into two classes: the first class has an explicitly defined exponential running time in the system size, and the second class has a running time that has not been defined or bounded, and thus offers hope for an exponential speed-up. Only the quantum Metropolis algorithm developed by Temme *et al.* offers a method for numerically estimating its running time on a

quantum computer [8]. The quantum Metropolis stochastic map can be simulated classically by performing a random walk on the Hamiltonian's eigenstates, discarding rejections, and accounting for all possible consecutive moves. The spectral gap of the resulting map is shown to be inversely proportional to the running time of the quantum algorithm; in our case, the map is represented as $2^N x 2^N$ matrix, where N is the number of qubits in the system, and the spectral gap is the difference between the largest and the second largest eigenvalues. Thus, a program hoping to model the running time of the quantum Metropolis sampling algorithm as a function of system size or sparsity must compute the spectral gap for random Hamiltonians. While the quantum Metropolis algorithm has been shown to increase linearly in the system size, our study focuses on Hamiltonians generated randomly from normal distributions. While there is no guarantee that the algorithm will yield exponential speedups for arbitrary local Hamiltonians – indeed, the recent work of Brandão and Svore only demonstrate that a quadratic speedup can be achieved for quantum semidefinite programming with existing quantum Gibbs samplers [7] – the linear combinations of the constraint matrices used in semidefinite programming are best represented as randomly generated Hamiltonians.

The focus of this thesis is to observe and discuss the effects that preparing the Gibbs states on a quantum computer has on the performance of classical semidefinite programming with the matrix multiplicative weights update method. Since the update step is prepared by Gibbs sampling, the running time of the quantum Gibbs sampler dominates the quantum semidefinite programming algorithm's running time, and thus an exponential speedup in the preparation of Gibbs states on a quantum computer will result in an exponential speedup in semidefinite programming. The numerical simulations of the running time of a quantum Gibbs sampler – in this case, the quantum Metropolis algorithm by Temme *et al.* – for different system sizes and different sparsities will shed light on the question of whether there exists certain circumstances under which the Gibbs states of randomly generated Hamiltonians are prepared exponentially faster on a quantum computer.

References

[1]R. P. Feynman, "Simulating physics with computers", International Journal of Theoretical Physics **21**, 467–488 (1982).

[2]E. P. Wigner, "On the quantum correction for thermodynamic equilibrium", Physical Review **40**, 749–759 (1932).

[3]J. E. Moyal, "Quantum mechanics as a statistical theory", Mathematical Proceedings of the Cambridge Philosophical Society **45**, 99–124 (1949).

[4]S. Lloyd, "Universal quantum simulators", Science **273**, 1073–1078 (1996).

[5]G. Casella and E. I. George, "Explaining the gibbs sampler", The American Statistician **46**, 167–174 (1992).

[6]S. Arora and S. Kale, "A combinatorial, primal-dual approach to semidefinite programs", Journal of the ACM **46**, 12.1–12.35 (2016).

[7]F. Brandão and K. Svore, "Quantum speed-ups for semidefinite programming", (2016), arXiv:1609.05537 [quant-ph].

[8]K. Temme, T. J. Osborne, K. G. Vollbrecht, D. Poulin, and F. Verstraete, "Quantum metropolis sampling", Nature **471**, 87–90 (2011).

C h a p t e r   2

CLASSICAL SEMIDEFINITE PROGRAMMING

Before the quantization of the semidefinite programming algorithm can be discussed, we must clarify what Arora's and Kale's combinatorial, primal-dual semidefinite programming entails. We use Arora's and Kale's definition of a primal-dual algorithm – that is, an algorithm that gives both primal and dual solutions, and bounds the duality gap, or the difference between the two solutions, with weak duality. Weak duality states that the duality gap is always greater than or equal to zero, which indicates that the primal solution is always greater than or equal to the dual solution [1]. For a general semidefinite program with n^2 variables and m constraints, the primal problem is generally formulated as the following [2].

$$
\begin{aligned}
\max \quad & \mathrm{tr}(\mathbf{CX}) \\
\forall j \in [m]: \quad & \mathrm{tr}(\mathbf{A}_j\mathbf{X}) \le b_j \\
& \mathbf{X} \ge \mathbf{0}
\end{aligned}
$$

The corresponding dual problem is as follows [2].

$$
\begin{aligned}
\min \quad & \mathbf{by} \\
\sum_{j=1}^{m} \; & \mathbf{A}_j y_j \ge \mathbf{C} \\
& \mathbf{y} \ge \mathbf{0}
\end{aligned}
$$

We can convert a semidefinite program given in the primal form to the dual form and vice versa. We observe that the n^2 variables are presented as a *nxn* positive semidefinite matrix $\mathbf{X}$. Because there are m constraints, there are m *nxn* Hermitian matrices $\mathbf{A}_j$ for $j \in [m]$. The matrix $\mathbf{C}$ is a *nxn* positive-semidefinite matrix. For the dual program, the dual variables are held in the vector $\mathbf{y} =< y_1, y_2, ..., y_m >$ and the constraints are real numbers packed into the vector $\mathbf{b} =< b_1, b_2, ..., b_m >$. A positive semidefinite matrix $\mathbf{X}$ that satisfies all of the primal constraints outlined by the primal problem is a primal feasible solution; likewise, a nonnegative vector $\mathbf{y}$ that satisfies all of the dual constraints outlined by the dual problem is a dual feasible solution.

2.1 Arora-Kale Algorithm for Semidefinite Programming

The Arora-Kale algorithm for semidefinite programming seeks a primal feasible solution to a primal-dual semidefinite program; simultaneously, it attempts to incrementally build a dual solution with the help of an auxiliary function called the **ORACLE**. It takes as its initial inputs the following parameters: α, which serves as a guess for the optimum value of the solution; an error parameter represented by an arbitrarily small constant $\epsilon > 0$; a width parameter $\rho \geq 0$ for the **ORACLE**; and a scaling constraint R [2]. By taking $\mathbf{A}_1 = \mathbf{I}$ and $b_1 = R$, Arora and Kale guarantee that $\mathrm{tr}(\mathbf{X}) \leq R$, which gives a constraint that bounds the feasible region of the semidefinite program [2]. Furthermore, setting $\mathbf{A}_1 = \mathbf{I}$ allows us to find values $y_1, y_2, \ldots y_m > 0$ that satisfy $\mathbf{A}_j y_j \geq \mathbf{C}$. The weak form of Slater's condition then holds, ensuring that strong duality holds in our semidefinite program. Thus the duality gap is zero, and the primal problem and the dual problem share the same optimum solution [3]. Nevertheless, in order to account for error, we introduce ϵ as the gap between the primal feasible solution and the dual feasible solution. The algorithm is searching for a positive semidefinite matrix as a primal feasible solution that is $\geq \alpha$ – if **ORACLE** never fails, then the algorithm successfully yields a dual feasible solution that is $\leq \alpha + \epsilon$ [2].

The parameter α bounds the range that **ORACLE** searches – it appears as a constraint on the vector $\mathbf{y}$ outputted by the **ORACLE**. The width parameter ρ is defined as the smallest nonnegative value that bounds the **ORACLE**'s output $\mathbf{y}$ such that $||\mathbf{A}_j y_j - \mathbf{C}|| \leq \rho$. The significance of the width parameter is twofold: ρ serves as both a measure of progress and a scaling factor for the loss matrices $\mathbf{M}$ that guarantee the spectral norm of $\mathbf{M}$ is bound by one. A large width would indicate that the **ORACLE** has not been effective in helping the algorithm make progress. The loss matrix $\mathbf{M}$ is required to perform the matrix multiplicative weights update step, as are the other parameters ϵ and R. The last two parameters are gathered with ρ and α into a new parameter ϵ, which plays a role in the matrix multiplicative weights algorithm.

The ORACLE

Before we discuss the matrix multiplicative weights algorithm, which not only serves as the backbone of this class of semidefinite programming algorithms, but also the foundation upon which the quantum semidefinite programming algorithm is based, we must first discuss the concept of the **ORACLE**. Arora and Kale classify the **ORACLE**s by the type of problem encountered and offer definitions for the

ORACLE based on whether the problem is a maximization semidefinite program or a minimization semidefinite program. We reproduce their definitions here:

Definition 2.1 (**ORACLE** for Maximization Semidefinite Programs) For maximization semidefinite programs, the auxiliary algorithm **ORACLE** must be constructed so that it takes as its input a positive semidefinite matrix $\mathbf{X} \geq \mathbf{0}$, or the candidate primal solution, and tries to output a vector $\mathbf{y}$ that meets the following constraints:

$$\mathbf{b} \cdot \mathbf{y} \leq \alpha$$

$$\sum_{j=1}^{m} \text{tr}(\mathbf{A}_j \mathbf{X}) y_j \geq \text{tr}(\mathbf{C}\mathbf{X})$$

$$\mathbf{y} \geq \mathbf{0}$$

If no such vector $\mathbf{y}$ exists, then the **ORACLE** outputs **FAIL** and returns $\mathbf{X}$. Arora and Kale prove that if the **ORACLE** fails, then an appropriately scaled $\mathbf{X}$ is a primal feasible solution $\geq \alpha$. But if a $\mathbf{y}$ is found, then the current candidate primal solution $\mathbf{X}$ fails as the primal feasible solution, and the algorithm must proceed with its iterations. If the **ORACLE** never fails for T iterations, where T is determined by the matrix size and the four parameters defined earlier, then the algorithm outputs a dual feasible solution.

The definition of the **ORACLE** for minimization semidefinite programs is similarly constructed and reproduced below:

Definition 2.2 (**ORACLE** for Minimization Semidefinite Programs) For maximization semidefinite programs, the auxiliary algorithm **ORACLE** must be constructed so that it takes as its input a positive semidefinite matrix $\mathbf{X} \geq \mathbf{0}$, or the candidate primal solution, and tries to output a vector $\mathbf{y}$ that meets the following constraints:

$$\mathbf{b} \cdot \mathbf{y} \geq \alpha$$

$$\sum_{j=1}^{m} \text{tr}(\mathbf{A}_j \mathbf{X}) y_j \leq \text{tr}(\mathbf{C}\mathbf{X})$$

$$\mathbf{y} \geq \mathbf{0}$$

Likewise, should the **ORACLE** fail to find such a vector $\mathbf{y}$ in the region bounded by the constraints, then it will fail and a primal feasible solution $\mathbf{X}$ will be returned. If the **ORACLE** never fails, then a linear combination of all of the vectors $\mathbf{y}$ generated throughout the iterations will be used to construct the dual feasible solution. Rather

than reproduce the proof of the algorithm and the required lemmas below, we will point interested readers in the direction of the original paper [2]. But we will delve into a discussion of the mechanisms used in their paper, especially the mechanisms that are crucial to developing the quantum version of this algorithm.

Matrix Multiplicative Weights Algorithm

The backbone of the Arora-Kale primal-dual algorithm for solving semidefinite programs is the matrix multiplicative weights algorithm, which takes as its input the number of iterations T and a parameter ϵ, and uses the density matrix $\mathbf{P}^{(t)}$ created from the previous weight matrix $\mathbf{W}^{(t)}$, along with a loss matrix generated by an external source $\mathbf{M}^{(t)}$, to output a new weight matrix $\mathbf{W}^{(t+1)}$. This final matrix is responsible for updating the algorithm. The superscript (t) denotes the current iteration. The algorithm in its entirety is reproduced below from [2]:

Algorithm 1 (Matrix Multiplicative Weights Algorithm)

Let $\eta \leq 1$. Let T be the number of iterations. For $t = 1, ..., T$:

1. *Find the density matrix $\boldsymbol{P}^{(t)} = \frac{W^{(t)}}{tr(W^{(t)})}$.*

2. *Find the loss matrix $M^{(t)}$ from the external source.*

3. *Compute the weight matrix for the next iteration:*
   $\boldsymbol{W}^{(t+1)} = \exp\left(-\eta(\sum_{\tau=1}^{t-1} \boldsymbol{M}^{(\tau)})\right).$

The matrix multiplicative weights algorithm is the matrix version of the multiplicative weights update method described in [4], which allows iterative updates to be made to the weights assigned to players associated with elements in a distribution. In this framework, the loss matrix $\mathbf{M}^{(t)}$ represents the penalty paid in the t-th iteration for the yielded output. The goal of the algorithm is to minimize the total loss so that it isn't much greater than the minimum loss $\lambda_n(\sum_{t=1}^{T} \mathbf{M}^{(t)})$, where λ_n is the smallest eigenvalue of $\sum_{t=1}^{T} \mathbf{M}^{(t)}$ [2]. While the primal-dual semidefinite program solver makes use of loss matrices, it disassociates the loss matrix from the concept of penalties – the main algorithm uses the matrix multiplicative weights framework because the following theorem and its corollary, which we reproduce from [2], yield an inequality that imposes the desired constraints upon the update step:

Theorem 2.1: For any collection of loss matrices $\{M^{(1)}, M^{(2)}, ...M^{(T)}\}$, the matrix multiplicative weights algorithm generates a collection of density matrices $\{P^{(1)}, P^{(2)}, ...P^{(T)}\}$ that satisfies the following inequality:

$$\text{tr}\left(\sum_{t=1}^{T} \mathbf{M}^{(t)}\mathbf{P}^{(t)}\right) \leq \lambda_n(\sum_{t=1}^{T} \mathbf{M}^{(t)}) + \eta \sum_{t=1}^{T} \text{tr}\left((\mathbf{M}^{(t)})^2\mathbf{P}^{(t)}\right) + \frac{\log n}{\eta} \qquad (2.1)$$

Corollary 2.2: For any collection of loss matrices $\{M^{(1)}, M^{(2)}, ...M^{(T)}\}$, the matrix multiplicative weights algorithm generates a collection of density matrices $\{P^{(1)}, P^{(2)}, ...P^{(T)}\}$ that satisfies the following inequality:

$$\text{tr}\left(\sum_{t=1}^{T} \mathbf{M}^{(t)}\mathbf{P}^{(t)}\right) \leq \lambda_n(\sum_{t=1}^{T} \mathbf{M}^{(t)}) + \eta T + \frac{\log n}{\eta} \qquad (2.2)$$

We will not reproduce the proofs, but we will discuss the significance of the derived conclusions to the final algorithm. In the Arora-Kale primal-dual algorithm for solving semidefinite programs, we have $\eta = \frac{\epsilon}{2\rho R}$ and $T = \lceil\frac{4\rho^2 R^2 \log(n)}{\epsilon^2}\rceil$. The main algorithm uses the slack matrix $M^{(t)} = \sum_{j=1}^{m} \mathbf{A}_j y_j - \mathbf{C}$ as the loss matrix during the matrix multiplicative weights update step. The slack matrix does not represent loss; mathematically speaking, it encodes the structure of the polytope $\mathcal{D}_\alpha = \mathbf{y} : \mathbf{y} \geq 0, \mathbf{b} \cdot \mathbf{y} \leq \alpha$ for maximization problems, and of the polytope $\mathcal{D}_\alpha = \mathbf{y} : \mathbf{y} \geq 0, \mathbf{b} \cdot \mathbf{y} \geq \alpha$ [5]. The slack matrix can also be intuitively interpreted as a matrix that represents the dual constraints [2]. The goal of the main algorithm is to produce such a dual feasible solution $\mathbf{y}$ that the slack matrix is positive semidefinite – this is a costly task, so rather than pursue this goal directly, the algorithm uses the **ORACLE** to seek a slack matrix that has a nonnegative matrix inner product with $\mathbf{X}^{(t)} = R\mathbf{P}^{(t)}$, the candidate primal solution that is updated iteratively along with the density matrices. This reduces the semidefinite dual problem to a linear program with two nontrivial constraints, which is an easier problem to solve [2].

The reason that finding $\mathbf{M}^{(t)}$ such that $\text{tr}(\mathbf{M}^{(t)}\mathbf{P}^{(t)}) \geq 0$ for all t is a feasible alternative to finding $\mathbf{y}$ such that the slack matrix is positive semidefinite rests in the inequality yielded by Corollary (2.2). Corollary (2.2) guarantees that the smallest eigenvalue of the average loss matrix $\frac{1}{T}\sum_{t=1}^{T}\mathbf{M}^{(t)}$ is not a large negative number – the corollary bounds it so that it is at least $-\eta - \frac{\log(n)}{\eta T}$. This implies that after a few iterations, the average slack matrix becomes almost positive semidefinite – thus, the algorithm will yield an almost feasible dual solution [2].

The Primal-Dual Solver

We have discussed the matrix multiplicative weights update method and the **ORACLE** in depth. These two auxiliary algorithms are the key mechanisms of the main algorithm: the matrix multiplicative weights update method urges the slack matrix in the positive semidefinite direction, which speeds up the search for a dual feasible solution, and the **ORACLE** checks iteratively which solution – primal or dual – is close to feasibility. The **ORACLE** may not necessarily output the dual feasible solution even when such a solution exists – in cases when the candidate primal solution has become almost primal feasible, but the dual feasible solution has not been reached yet, the **ORACLE** may choose to fail and simply output the primal feasible solution [2].

The original version of the Arora-Kale semidefinite programming algorithm was published in proceedings back in 2007 [6], spurring a surge in semidefinite programming algorithms that use the multiplicative weights method. There have been slight changes made in the 2016 publication of the final paper, but besides minor modifications to the error parameter and the loss matrix, the algorithm remains largely the same. The primal-dual algorithm for solving semidefinite programs is reproduced below in its entirety from Arora *et al*'s 2016 publication [2]:

Algorithm 2 (Arora-Kale Semidefinite Programming Algorithm (2016))

Set $X^{(1)} = \frac{R}{n}I$. Let T be the number of iterations. We have $\eta = \frac{\epsilon}{2\rho R}$ and $T = \lceil \frac{4\rho^2 R^2 \log(n)}{\epsilon^2} \rceil$. For $t = 1, ..., T$:

1. *Run the matrix multiplicative weights algorithm and find the density matrix $P^{(t)}$.*

2. *Set $X^{(t)} = RP^{(t)}$.*

3. *Input $X^{(t)}$ into **ORACLE** and run the auxiliary algorithm.*

4. *If **ORACLE** fails, then abort and return $X^{(t)}$. Otherwise, store the results obtained as vector $y^{(t)}$.*

5. *Compute the loss matrix $M^{(t)} = \sum_{j=1}^m A_j y_j - C$ and feed it into the matrix multiplicative weights algorithm for the next iteration.*

6. *If **ORACLE** doesn't fail for T iterations, then output the dual feasible solution $\bar{y} = \frac{1}{T} \sum_{t=1}^T y^{(t)} + \frac{\epsilon}{R}e_1$, where $e_1 =< 1, 0, ...0 >^T \in \mathcal{R}^m$.*

Brandão *et al.* uses the 2007 version of the Arora-Kale algorithm for their quantum semidefinite programming algorithm, where the parameters are defined in a slightly different way. The error parameter is defined as δ, such that the dual feasible solution is at most $(1 + \delta)\alpha$, and what is defined as η in the 2016 publication is separated into two parameters ϵ and ϵ' in the 2007 proceedings paper. We reproduce the 2007 version of the matrix multiplication weights algorithm from [6]:

Algorithm 3 (Matrix Multiplicative Weights Algorithm)

Let $\epsilon < \frac{1}{2}$ and let $\epsilon' = -\log(1 - \epsilon)$. Let T be the number of iterations. For $t = 1, ..., T$:

1. *Compute the weight matrix: $W^{(t)} = \exp\left(-\epsilon'(\sum_{\tau=1}^{t-1} M^{(\tau)})\right)$.*

2. *Find the density matrix $P^{(t)} = \frac{W^{(t)}}{tr(W^{(t)})}$.*

3. *Find the loss matrix $M^{(t)}$ from the external source.*

The main algorithm witnesses one more change in its transformation from the 2007 version to the 2016 version: in its previous incarnation, the loss matrix was defined as $M^{(t)} = \left(\sum_{j=1}^{m} \mathbf{A}_j y_j - \mathbf{C} + \rho \mathbf{I} \right) / 2\rho$. The full algorithm is reproduced below from [6]:

Algorithm 4 (Arora-Kale Semidefinite Programming Algorithm (2007))

Set $\boldsymbol{X}^{(1)} = \frac{R}{n} \boldsymbol{I}$. *Let T be the number of iterations. We have* $\epsilon = \frac{\delta \alpha}{2\rho R}$ *and* $\epsilon' = -\log(1 - \epsilon)$. *In addition, we let* $T = \frac{\rho^2 R^2 \log(n)}{\delta^2 \alpha^2}$. *For* $t = 1, ..., T$:

1. *Input* $\boldsymbol{X}^{(t)}$ *into* **ORACLE** *and run the auxiliary algorithm.*

2. *If* **ORACLE** *fails, then abort and return* $\boldsymbol{X}^{(t)}$. *Otherwise, store the results obtained as vector* $\boldsymbol{y}^{(t)}$.

3. *Compute the loss matrix* $M^{(t)} = \sum_{j=1}^{m} \boldsymbol{A}_j y_j - \boldsymbol{C}$ *and feed it into the matrix multiplicative weights algorithm for the next iteration.*

4. *Run the matrix multiplicative weights algorithm and find the density matrix* $\boldsymbol{P}^{(t)}$.

5. *Set* $\boldsymbol{X}^{(t)} = R\boldsymbol{P}^{(t)}$.

6. *If* **ORACLE** *doesn't fail for T iterations, then output the dual feasible solution* $\bar{\boldsymbol{y}} = \frac{1}{T} \sum_{t=1}^{T} \boldsymbol{y}^{(t)} + \frac{\delta \alpha}{R} \boldsymbol{e}_1$, *where* $\boldsymbol{e}_1 = <1, 0, ...0 >^T \in \mathcal{R}^m$.

The worst case running time for the algorithm above is $\tilde{O}\left(\frac{\rho^2 R^2 mns}{\delta^2} \right)$, where m is the number of input matrices $\mathbf{A}_j$, n is the dimension of the input matrices, and s is the row sparsity of the input matrices [6]. Alternate semidefinite programming algorithms using the multiplicative weights method published in the wake of the 2007 publication may have faster running times, but we have chosen to focus on the quantum version of the Arora-Kale algorithm. Nevertheless, other classical algorithms for solving semidefinite programs that uses the matrix multiplicative weight method may also be viable candidates for quantization. The runtimes described in Brandão's and Svore's paper will not necessarily hold, but as long as the matrix multiplicative weight method is used, then the algorithm can be given a quantum speedup by using

a quantum computer to prepare the Gibbs states of the Hamiltonians constructed by the linear combinations of the program's input matrices [7]. Thus the numerical results of our runtime simulations for the quantum Metropolis algorithm will have implications for classical semidefinite programming algorithms that are not necessarily primal-dual but use the matrix multiplicative weight algorithm, such as the ones outlined by Peng *et al.* [8]and Allen-Zhu *et al.* [9].

References

[1]E. de Klerk, *Aspects of semidefinite programming: interior point algorithms and selected applications* (Kluwer Academic Publishers, Dordrecht, The Netherlands, 2002).

[2]S. Arora and S. Kale, "A combinatorial, primal-dual approach to semidefinite programs", Journal of the ACM **46**, 12.1–12.35 (2016).

[3]S. Boyd and L. Vandenberghe, *Convex optimization* (Cambridge University Press, Cambridge, UK, 2004).

[4]S. Arora, E. Hazan, and S. Kale, "The multiplicative weights update method: a meta-algorithm and applications", Theory of Computing **8**, 121–164 (2012).

[5]R. Z. Robinson, "The positive semidefinite rank of matrices and polytopes", PhD thesis (University of Washington, 2014).

[6]S. Arora and S. Kale, "A combinatorial, primal-dual approach to semidefinite programs", Proceedings of the 39th ACM Symposium on Theory of Computing (STOC'07), 227–236 (2007).

[7]F. Brandão and K. Svore, "Quantum speed-ups for semidefinite programming", (2016), arXiv:`1609.05537` `[quant-ph]`.

[8]R. Peng, K. Tangwongsan, and P. Zhang, "Faster and simpler width-independent parallel algorithms for positive semidefinite programming", (2016), arXiv:`1201.5135` `[cs.DS]`.

[9]Z. Allen-Zhu, Y. T. Lee, and L. Orecchia, "Using optimization to obtain a width-independent, parallel, simpler, and faster positive sdp solver", (2016), arXiv:`1507.02259` `[cs.DS]`.

QUANTUM SEMIDEFINITE PROGRAMMING

As in the classical version of the algorithm, the quantum semidefinite programming algorithm proposed by Brandão *et al.* takes in $m + 1$ $n \times n$ constraint matrices, finds the optimal values of primal and dual problems presented in the previous chapter, and outputs the primal feasible solution and/or the dual feasible solution. However, while the classical algorithm takes in the input matrices through the loss matrix $M^{(t)}$, the quantum algorithm uses a different method to access the input matrices.

In [1], there are two methods of accessing the input matrices. The first method is to define an oracle that takes as its input the index j denoting the constraint matrix A_j, the index $k \in [n]$ denoting the row of A_j, and $l \in [s]$ denoting the number of nonzero elements in row k, where $l \leq s$, and outputs a bit string representation of the $l - th$ non-zero element in the $k - th$ row in the $j - th$ constraint matrix A_j, which would yield the following map:

$$|j, k, l, z\rangle \rightarrow |j, k, l, z \oplus (A_j)_{k f_{jk}(l)}\rangle, \tag{3.1}$$

where $f_{jk}(l)$ is a function that outputs the column index corresponding to the position of the $l - th$ nonzero element in the $k - th$ row of A_j.

The second method is more complicated and rests on the spectral decomposition of A_i. We define two sets of oracles: the first set takes in the input matrices A_i, prepares their eigenstates, and uses oracles to access their eigenvalues and the numbers b_i. The decomposition of the input matrix A_i is: $A_i = \sum_{l=1}^{r_i} \kappa_l^i |\eta_l^i\rangle \langle \eta_l^i|$. The value r_i will be defined shortly. The second set of oracles only contains one oracle, which takes as its input (i, l), where i takes on the role of row index, and $r_i := \lambda \text{tr}(A_i \rho) + \mu b_i$ for real numbers λ and μ, and quantum state ρ; the oracle outputs an approximation of A_j's eigenstate $|\eta_l^i\rangle$ and the corresponding eigenvalue κ_l^i up to an error ν. The value r_i is defined such that it is the i-th eigenvalue of the Hamiltonian defined as $H(\rho, \lambda, \mu) := \sum_{i=1}^{m} r_i |i\rangle \langle i|$ [1].

The output is the same for both methods of input. Rather than output the dual feasible solution **y** or the primal feasible solution **X**, the algorithm outputs a simpler

result in the interest of efficiency – it yields an estimated optimal value, an estimated $||y||_1$ and/or $tr(X)$, and samples from $y/||y||_1$ and/or from $\rho := X/tr(X)$.

The essence of the algorithm is that the time-consuming linear programs and matrix exponentials in the classical algorithm [2, 3] are replaced with Gibbs states prepared on a quantum computer, a replacement that offers a quantum speedup. The Gibbs samplers used to prepare the Gibbs states may range from [4–8], although Brandão *et al.* express hope that the quantum Metropolis sampling algorithm by Temme *et al.* [7] may work best heuristically. Brandão *et al.* noted that the density matrix generated with the matrix exponential of the loss matrix using the matrix multiplicative weights method from [3] can be replaced with a Gibbs state of a Hamiltonian composed from a linear combination of the input matrices [1]. They also demonstrate how the output of the classical oracle in [2, 3] can be approximated with a Gibbs state using a modified version of Jayne's principle of maximum entropy [9, 10], which we reproduce from [10] with modifications from [1]:

Lemma 3.1 (Lemma 4.6 from [10]) Let $\mathcal{M}(\mathbb{C}^n)$ be the set of Hermitian matrices over $\mathbb{C}^n$, and $\mathcal{D}(\mathbb{C}^n)$ be the set of density matrices over $\mathbb{C}^n$. We define $\mathcal{T} \subseteq \mathcal{M}(\mathbb{C}^n)$ as compact set of matrices and define $\Delta\mathcal{T} := \sup_{A \in \mathcal{T}} ||A||$. Define $\pi \in \mathcal{D}(\mathbb{C}^n)$. Finally, define $\gamma = \lceil \frac{8}{\kappa^2} \log(m)\Delta(\mathcal{T})^2 \rceil$. We define one more input parameter κ. Then for every $\kappa > 0$, there exists a set of matrices $X_1, ..., X_\gamma \in \mathcal{T}$ such that:

$$\tilde{\pi} := \frac{\exp\left(-\frac{\kappa}{4\Delta(\mathcal{T})^2} \sum_{i=1}^{\gamma} X_i\right)}{\operatorname{tr}\left(\exp\left(-\frac{\kappa}{4\Delta(\mathcal{T})^2} \sum_{i=1}^{\gamma} X_i\right)\right)} \tag{3.2}$$

is bounded by $[\pi - \tilde{\pi}]_{\mathcal{T}} \leq \kappa$. It should be obvious by now that κ is related to the error parameter ν in our case.

Brandão *et al.* then adapt this lemma so that the output of the **ORACLE**(ρ) becomes:

$$q_{\rho,\lambda,\nu}(i) := \frac{\exp(\lambda\operatorname{tr}(A_i\rho) + \mu \sum_i b_i)}{\sum_i \exp(\lambda\operatorname{tr}(A_i\rho) + \mu \sum_i b_i)}. \tag{3.3}$$

The constants λ and ν are reparameterized into k so that for an integer k we have [1]:

$$\lambda = -\frac{\kappa}{4R^2}k, \tag{3.4}$$

$$\nu = -\frac{\kappa}{4R^2}(\gamma - k), \tag{3.5}$$

and

$$\gamma = \lceil \frac{8}{\kappa^2} \log(m)R^2 \rceil. \tag{3.6}$$

As in the classical algorithm, R is the upper bound on the trace of the primal optimal solution [1]. We can now write $q_{\rho,\lambda,\nu}$ as $q_{\rho,k}$. We then give the definition that Brandão *et al.* use for a Gibbs sampler.

Definition 3.2 (Gibbs Sampler) A quantum circuit that takes as its input ν, a Hamiltonian H, and an oracle to access the entries of H [11], and outputs a state ρ that obeys the constraint $||\rho - e^H/tr(e^H)||_1 \leq \nu$.

We see then that the output of the **ORACLE** used in [2, 3] can be approximated with Gibbs sampling. We won't reproduce the algorithm in [1] for instantiation of **ORACLE**(ρ) through sampling, but it should be noted that the algorithm samples from distributions $\bar{q}_{\rho,k}$ that are close in variational distance to $q_{\rho,k}$ rather than from $q_{\rho,k}$ itself.

We now reproduce the quantum semidefinite programming algorithm in its entirety below [11]:

Algorithm 5 (Brandão-Svore Quantum SDP Algorithm (2016))

Oracles for accessing $\{A_1, ... A_m, C\}$ where $||C||, ||A_i|| \leq 1$. Oracles for accessing $\{b_1, ..., b_m\}$ where $b_i \geq 1$. Parameter R where R is the upper bound on the trace of the primal optimal solution. Parameter α, which a guess for the optimal solution, and the error $\delta > 0$ such that that dual feasible solution is at most $\alpha(1 + \delta)$. A free parameter Ξ that features in the lower bound on the probability of success: $1 - \mathcal{O}(\exp(-\log^{\Xi}(nm)))$. Set $\rho^{(1)} = \frac{I}{n}$, $\epsilon = \frac{\delta}{28R^2}$, $\epsilon' = -ln(1 - \epsilon)$, $M = 80 \log^{1+\Xi}(8R^2nm/\epsilon)/\epsilon^2$, $L = 80 \log^{1+\Xi}(nm)/\epsilon^2$, and $Q = (10R)^6 ln^{2+\Xi}(nm)/\epsilon^4$. Set $\gamma = \lceil \frac{8}{\epsilon^2} \log(m)R^2 \rceil$. Define $G_H(\rho)$ as the maximum number of calls made by the Gibbs sampler to the oracle that accesses the entries of $H(\rho, k)$ for $k \in [\gamma]$. Let T be the number of iterations. Set $T = \frac{500R^3 ln(n)}{\delta^2}$. For $t = 1, ..., T$:

1. *Set $y^{(t)} = (0, ...0)$.*

2. *For $k = 1, ..., \gamma$ and $N = 1, ..., \lceil \frac{\alpha}{\epsilon} \rceil$, use the Gibbs sampler on $H(\rho, k)$ to create M copies of q with up to an error of $\epsilon/4$. Obtain M independent samples $\{i_1, ... i_M\}$ from q. Using $(M + 1)L$ samples from $\rho^{(t)}$, approximate $\{tr(A_{i_1}\rho^{(t)}), ..., tr(A_{i_M}\rho^{(t)}), tr(C\rho^{(t)})\}$ up to accuracy $\epsilon/2$ and store the results as $\{e_{i_1}, ..., e_{i_M}\}, f$. If $1/M \sum_{j=1}^{M} e_{i_j} \geq f/(N\epsilon) - \epsilon$ and $1/M \sum_{j=1}^{M} b_{i_j} \leq \alpha/(N\epsilon) + R\epsilon$, then set $k_t = k$, $N_t = N$, $q_{(t)} = q$, and $y^{(t)} = N\epsilon q^{(t)}$.*

3. *Abort and indicate failure if $y^{(t)} = (0, ..., 0)$.*

4. *Use the Gibbs sampler on $H(\rho^{(t)}, k_t)$ to create $Q + 1$ copies of $q^{(t)}$ with up to an error of $\epsilon/4$.*

5. *Obtain Q independent samples $\{i_1, ... i_Q\}$ from $q^{(t)}$.*

6. *Compute $M^{(t)} = \frac{\epsilon N_t Q^{-1} \sum_{j=1}^{Q} A_{ij} - C + 2\alpha I}{4\alpha}$.*

7. *Define $C_t := \frac{10 \log(m)}{\epsilon^2} \left(\frac{\gamma\alpha}{\epsilon} M + Q \right) G_H(\rho^{(t)}) + \frac{2\gamma\alpha}{\epsilon} ML$ and the Gibbs sampler on $-\epsilon' \left(\sum_{\tau=1}^{t} M^{(\tau)} \right)$ with an error up to $\epsilon/4$ to make C_t copies of $\rho^{(t+1)}$.*

If the algorithm runs without failing, then it will output $||\bar{y}||_1$ and a sample from $\bar{y}/||\bar{y}||_1$ for $\bar{y} = \frac{\delta\alpha}{2R} e_1 + \frac{1}{T} \sum_{t=1}^{T} y^{(t)}$, where $e_1 = (1, 0, ...0)$. We won't replicate the proofs leading to the final runtimes, but we will give a brief description of the

quantum speedups that have been rigorously verified in Brandão *et al.* The quantum semidefinite programming algorithm by Brandão *et al.* has a worst case runtime of $n^{\frac{1}{2}}m^{\frac{1}{2}}s\,\mathrm{poly}(\log(n),\log(m),R,1/\delta)$, where we once again emphasize that n is the dimension of the input matrices and thus the Hilbert space dimension and s is the row sparsity of the input matrices. The quantum algorithm offers a square root speedup in n and m over classical methods; Brandão *et al.* demonstrate that this is the best possible runtime in terms of n and m and give a lower bound for the complexity of the algorithm: $\Omega(n^{\frac{1}{2}}+m^{\frac{1}{2}})$ for a given (s,δ,R). However, there is one case in which the algorithm can witness exponential speedups in n – when the quantum Gibbs states used in the algorithm can be efficiently prepared on a quantum computer [1].

While there is no way to rigorously prove that using the quantum Metropolis sampling scheme [7] to prepare the required Gibbs states will exponentially speed up the quantum semidefinite programming algorithm by Brandão *et al.* for arbitrary row-sparse Hamiltonians, we seek to demonstrate numerically that for three to ten qubits, the quantum semidefinite programming algorithm experiences an exponential speedup in n when the quantum Metropolis algorithm is the chosen Gibbs sampler.

References

[1] F. Brandão and K. Svore, "Quantum speed-ups for semidefinite programming", (2016), arXiv:1609.05537 [quant-ph].

[2] S. Arora and S. Kale, "A combinatorial, primal-dual approach to semidefinite programs", Journal of the ACM **46**, 12.1–12.35 (2016).

[3] S. Arora and S. Kale, "A combinatorial, primal-dual approach to semidefinite programs", Proceedings of the 39th ACM Symposium on Theory of Computing (STOC'07), 227–236 (2007).

[4] A. N. Chowdhury and R. D. Somma, "Quantum algorithms for gibbs sampling and hitting-time estimation", (2016), arXiv:1603.02940 [quant-ph].

[5] M. J. Kastoryano and F. G. S. L. Brandão, "Quantum gibbs samplers: the commuting case", Communications in Mathematical Physics **344**, 915–957 (2016).

[6] D. Poulin and P. Wocjan, "Sampling from the thermal quantum gibbs state and evaluating partition functions with a quantum computer", Physical Review Letters **103**, 220502 (2009).

[7] K. Temme, T. J. Osborne, K. G. Vollbrecht, D. Poulin, and F. Verstraete, "Quantum metropolis sampling", Nature **471**, 87–90 (2011).

[8]M. Yung and A. Aspuru-Guzik, "A quantum–quantum metropolis algorithm", Proceedings of the National Academy of Sciences, 754–759 (2012).

[9]E. T. Jaynes, "Information theory and statistical mechanics ii", Physical Review **108**, 171–190 (1957).

[10]J. R. Lee, P. Raghavendra, and D. Steurer, "Lower bounds on the size of semidefinite programming relaxations", Proceedings of the 47th Annual ACM Symposium on Theory of Computing (STOC '15), 567–576 (2015).

[11]D. W. Berry, A. M. Childs, and R. Kothari, "Hamiltonian simulation with nearly optimal dependence on all parameters", Proceedings of the 56th IEEE Symposium on Foundations of Computer Science, 792–809 (2015).

C h a p t e r 4

QUANTUM METROPOLIS SAMPLING

Metropolis *et al.* devised the Metropolis algorithm in 1953 to calculate the properties of classical many-body systems with the Markov chain Monte Carlo method [1]; while ground-breaking for classical simulations, it was not until recently that attempts to generalize the algorithm to quantum systems have succeeded. Previous quantum Monte Carlo methods failed after encountering fermionic systems and the related sign problem, a catchall phrase describing the complications and difficulty surrounding the evaluation of a highly oscillatory integral [2]. Long after the publication of Lloyd's paper on the viability of quantum computing, there was still a dearth of quantum algorithms that allowed the efficient preparation of a Hamiltonian's thermal or ground state.

The quantum algorithms that did prepare ground states efficiently only worked for frustration free systems, which limited research to a subset of physical systems [3]. In 2000, Terhal and Divincenzo discussed sampling from a physical system's thermal state using methods reminiscent of the classical Metropolis algorithm; however, they neglected to provide a way to circumvent the no-cloning problem, which posed an obstacle to the crucial rejection step in the Metropolis algorithm. Furthermore, computing the eigenvalues of the Hamiltonian with the Fourier transform, a step that makes up half of the algorithm, may require a runtime exponential in system size [4]. In 2005, Aspuru-Guzik *et al.* expanded the phase estimation algorithm developed by Abrams *et al.* [5] into a quantum algorithm that prepares ground states of more general Hamiltonians; the stringent requirement that the variational state overlap greatly with the ground state nevertheless limits the algorithm's range of applicability [6].

In 2009, Temme *et al.* synthesized a quantum Metropolis algorithm that allows the efficient preparation of ground states and Gibbs states of many-body quantum systems, and thus may be widely used in simulations of such systems. While this algorithm does not guarantee the efficient preparation of the ground or thermal states of random Hamiltonians, heuristically the update step can always be chosen so that the algorithm thermalizes in polynomial time if the physical system does so too. Since the quantum Metropolis algorithm samples directly from the given Hamil-

tonian's eigenstates, it dispenses with the sign problem, offering an exponential speedup to quantum algorithms still plagued with the task of dealing with negative statistical weights. Furthermore, Temme *et al.* gives a method of circumventing the no-cloning problem and resolving the issue of the rejection step; the analysis of the algorithm's total runtime includes the runtime of the rejection procedure. The running time is bounded by a function of the quantum Metropolis stochastic map's spectral gap; since the relationship between the spectral gap and the system size is unknown for random Hamiltonians, the quantum Metropolis algorithm does not have a running time that is explicitly defined in system size [7]. Thus the quantum Metropolis algorithm in theory may offer an exponentially faster method of preparing a quantum computer in the ground or thermal state of the physical system at hand.

4.1 The Algorithm

Before the quantum version of the classical Metropolis algorithm is presented, we must introduce the classical Metropolis sampling method. We use the simplest example presented in the original 1953 paper: a lattice of N points where the distances between all of the points are known, so that the potential energy of the system E is easily calculated. Configurations with a probability of $\exp(-E/kT)$ are chosen beforehand and equally weighed. The particles are then moved randomly; after a move, the change of energy ΔE is calculated. If $\Delta E < 0$, then the move has caused the system to shift to a state of lower energy – such a move is permitted and thus retained. But if $\Delta E > 0$, then the probability of such a move is $\exp(-E/kT)$ – an implementation of a probabilistic move can be done by randomly selecting a number from $[0, 1]$ and retaining the move if the number $< \exp(-E/kT)$ [1].

Temme *et al.* have devised a quantum algorithm that performs the same procedure on a given Hamiltonian's eigenstates. Instead of moving a particle to a new location, a random local unitary operator is selected to act upon the qubits; the acceptance and rejection terms remain the same as in the classical case. But in the quantum case, the difficulties are threefold: the algorithm has to account for computing the Hamiltonian's eigenstates and their corresponding energies, undoing quantum transformations if the move is rejected, and proving that the desired Gibbs state is the stationary state of the Markov chain [7].

The first objective can be easily accomplished if there are no instances where no action by the randomly selected local unitary operator needs to be undone. Since

finding the phase of a unitary operator is equivalent to calculating the operator's energy eigenvalue, the quantum phase estimation algorithm can be used to prepare a random eigenstate $|\psi_i\rangle$ as the initial guess for the Hamiltonian's ground state and calculate its energy, which is stored in an empty register, yielding $|\psi_i\rangle |E_i\rangle$. An implementation of the phase estimation algorithm can be found in [8]. A random local unitary operation C is then applied to the eigenstate, yielding $C |\psi_i\rangle = \sum_k x_k^i |\psi_k\rangle$. The quantum phase estimation algorithm is performed again to calculate the energy of the new configuration, which is stored in the second register, yielding $\sum_k x_k^i |\psi_k\rangle |E_i\rangle |E_k\rangle$.

If there were no rejection procedure, then we could proceed to measure the second register to find the energy and collapse the wavefunction. If this were a classical algorithm, then we could store a copy of the original wavefunction so that we could simply return to it should the update be rejected; however, the no-cloning theorem rules out this possibility for quantum algorithms. Thus the quantum Metropolis circuit must include another gate – a gate that doesn't perform a full energy measurement, but rather only slightly disturbs the state. The only information that is yielded by such a measurement is whether to accept or reject the update. A final register is introduced in this step, as the gate transforms the state into $\sum_k x_k^i \sqrt{f_k^i} |\psi_k\rangle |E_i\rangle |E_k\rangle |1\rangle + \sum_k x_k^i \sqrt{1 - f_k^i} |\psi_k\rangle |E_i\rangle |E_k\rangle |0\rangle$, where the amplitudes $x_k^i \sqrt{f_k^i}$ are the quantum analog of the classical Metropolis transition probabilities. The function f_k^i is $f_k^i = \min(1, \exp(-\beta(E_k - E_i)))$.

The quantum Metropolis circuit up to the rejection procedure can be summed up as [7]:

$$
\begin{aligned}
|\psi_i\rangle |E_i\rangle |0\rangle |0\rangle &\rightarrow C |\psi_i\rangle |E_i\rangle |0\rangle |0\rangle \\
&\rightarrow \sum_k x_k^i |\psi_k\rangle |E_i\rangle |0\rangle |0\rangle \\
&\rightarrow \sum_k x_k^i |\psi_k\rangle |E_i\rangle |E_k\rangle |0\rangle \\
&\rightarrow \sum_k x_k^i \sqrt{f_k^i} |\psi_k\rangle |E_i\rangle |E_k\rangle |1\rangle + \sum_k x_k^i \sqrt{1 - f_k^i} |\psi_k\rangle |E_i\rangle |E_k\rangle |0\rangle
\end{aligned}
$$

A measurement of the last qubit will reveal the acceptance decision – if the outcome is $|0\rangle$, then the third register containing the energy E_k should be measured and the inverse quantum phase estimation gate should be applied. The new eigenstate

will then be fed into the algorithm again at the start of a new iteration. This measurement is denoted as Q. But if the outcome is $|1\rangle$, then the move has been rejected and the algorithm must return to the original state [7]. In the case that the update is rejected, the algorithm can return to the original state using a procedure similar to the witness-preserving amplification scheme for QMA described in [9]. The circuits used to implement the quantum Metropolis algorithm are described in detail below in Fig. (4.1), which is reproduced from [7]:
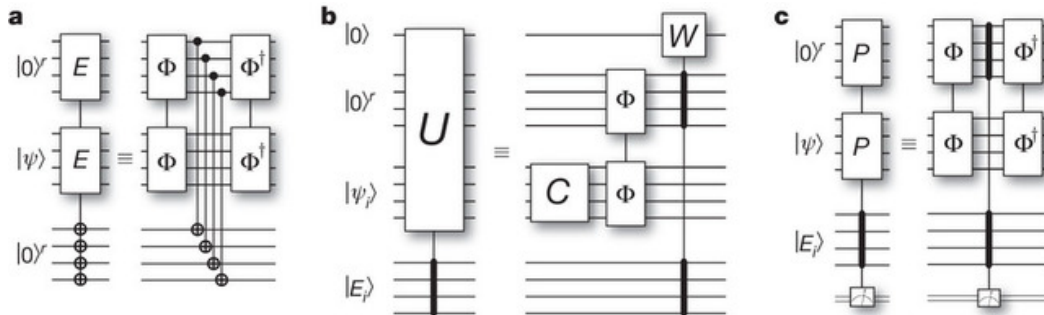
Figure 4.1: **Circuits for Each Step of the Quantum Metropolis Algorithm.** **a**, This subfigure depicts the initial application of the quantum phase estimation algorithm. The input is the initial guess for the ground state and two r-qubit registers. The quantum phase estimation circuit, represented by Φ in the diagram, acts on the state and the second register. The resulting energy is copied from the second register to the first register by a sequence of r CNOT gates. The inverse quantum phase estimation circuit, represented by $\Phi^\dagger$, is then applied to the second register and the state. **b**, The quantum Metropolis circuit takes as its input the state, one r-qubit register initialized to $|0\rangle^r$, and a single qubit register initialized to $|0\rangle$. The circuit can be separated into the local unitary operator C, which acts upon the state, and two quantum phase estimation gates, which act upon both the state and the r-qubit register. The quantum Metropolis gate acts upon the single-qubit register $|0\rangle$. **c**, The operations required for returning to the original state if the update is rejected are shown in this subfigure. The quantum phase estimation gate is applied to both the state and the new r-qubit register; once the energy is compared to the original energy E_i, the quantum phase estimation can be undone with inverse quantum phase estimation gates. Reprinted by permission from Macmillan Publishers Ltd: *Nature* 471:87, copyright 2011.

The rejection procedure – more specifically, the process for returning to the original state – is best described as an alternating sequence of two measurements. The first measurement determines whether the update is accepted or rejected. The second measurement determines whether the algorithm has returned to the original state $|\psi_i\rangle$. If both measurements yield negative results, the sequence is repeated until the second measurement gives a positive outcome. The probability of obtaining

two negatives is constant, so repeated applications of this sequence gives a success probability that exponentially approaches one [7]. The circuit for the rejection procedure can be found below in Fig. (4.2), which is reproduced from [7]:
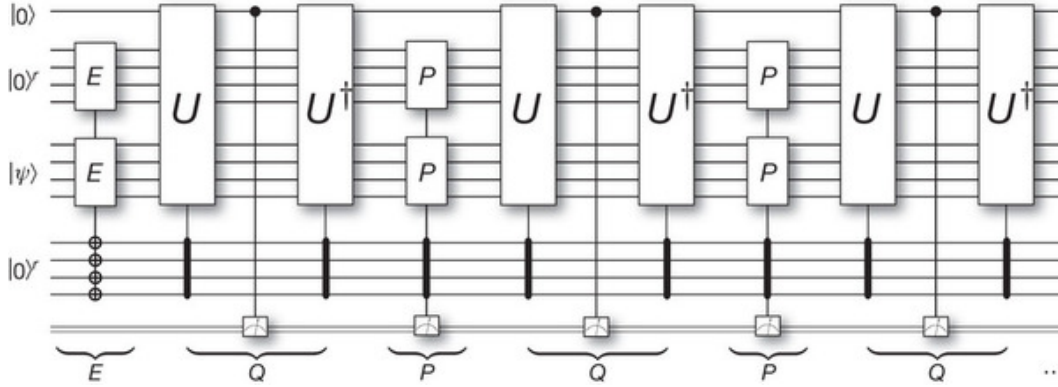
Figure 4.2: **A single application of the quantum Metropolis map.** The circuit for a single application of the quantum Metropolis stochastic map is presented above. The two gates E act on the state $|\psi\rangle$ and the r-qubit register prepare the eigenstate as the input of the algorithm. The first U gate proposes an update; the first Q measurement determines whether it is accepted or rejected. Should the update be rejected, then the P measurement will be performed. The alternating measurements of Q and P will continue until a positive measurement outcome of P_1 is obtained. Reprinted by permission from Macmillan Publishers Ltd: *Nature* 471:87, copyright 2011.

The possible results of the measurement, represented as the Hermitian projectors Q_0, Q_1, P_0, and P_1 are defined as below:

$$Q_0 = U^\dagger(\mathbb{I} \otimes \mathbb{I} \otimes \mathbb{I} \otimes |0\rangle\langle0|)U$$

$$Q_1 = U^\dagger(\mathbb{I} \otimes \mathbb{I} \otimes \mathbb{I} \otimes |1\rangle\langle1|)U$$

$$P_0 = \sum_i \sum_{E_\alpha \neq E_i} |\psi_\alpha\rangle\langle\psi_\alpha| \otimes |E_i\rangle\langle E_i| \otimes \mathbb{I} \otimes \mathbb{I}$$

$$P_1 = \sum_i \sum_{E_\alpha = E_i} |\psi_\alpha\rangle\langle\psi_\alpha| \otimes |E_i\rangle\langle E_i| \otimes \mathbb{I} \otimes \mathbb{I}$$

It is obvious from the definition of P_1 that if P_1 is obtained, then the measurement P, as positioned in part c of Fig. (4.1), has compared the energy of the state to its original energy and found them to be the same.

Before we reproduce the algorithm in its entirety, we will first discuss how the third obstacle – proving that the desired Gibbs state is the stationary state of the Markov chain – was overcome and then give the algorithm's runtime. Temme *et al.*

came up with the concept of quantum detailed balance, which holds that if $\{\psi_i\}$ is a set of complete basis states on a physical Hilbert space and $\{p_i\}$ is its probability distribution, then if a completely positive map $\mathcal{E}$ satisfies the following condition, $\sigma = \sum_i p_i |\psi_i\rangle \langle\psi_i|$ is the map's fixed point. The condition is as follows [7]:

$$\sqrt{p_n p_m} \langle\psi_i| \mathcal{E}(|\psi_n\rangle \langle\psi_m|) |\psi_j\rangle = \sqrt{p_i p_j} \langle\psi_m| \mathcal{E}(|\psi_j\rangle \langle\psi_i|) |\psi_n\rangle \tag{4.1}$$

This condition only guarantees that the Gibbs state is a possible fixed point of $\mathcal{E}$ – the uniqueness of the fixed point and the rate of convergence are dependent on the choice of updates $\{C\}$ [7]. It is known from [10] that if the set of possible updates is chosen to be the universal gate set, then the completely positive map $\mathcal{E}$ is ergodic, and the fixed point is unique. In our implementation, we choose the set of Pauli operators $\{I, X, Y, Z\}$ as our set of possible updates $\{C\}$.

The runtime of the algorithm is given by the mixing time, or the number of times that the completely positive map $\mathcal{E}$ must be applied to reach the Gibbs state. Before we reproduce the mixing time m_{mix}, we first discuss the mixing error ϵ^{mix} used to bound the mixing time [7]:

$$||\mathcal{E}^{m_{mix}}[\rho_0] - \sigma^*||_1 \leq \epsilon^{mix} \tag{4.2}$$

In other words, the mixing error is defined as the upper bound of the trace norm distance of the completely positive map $\mathcal{E}$. In the equation above, ρ_0 is the initial state and σ^* is the steady state. Since the trace norm distance is also bounded by the following equation [11]:

$$||\mathcal{E}^m[\rho] - \sigma^*||_1 \leq C_{exp}(1 - \Delta)^m, \tag{4.3}$$

where C_{exp} is a constant that scales exponentially with the system size and Δ is the spectral gap, or the difference between the largest and the second largest eigenvalues of $\mathcal{E}$, it is obvious that the mixing time is bounded as below [7]:

$$m_{mix} \geq O\left(\frac{\ln(1/\epsilon^{mix})}{\Delta}\right) \tag{4.4}$$

While this algorithm has a clearly bounded mixing time in the spectral gap of its stochastic map, the relationship between the mixing time and the system size is not

explicitly defined. Thus there is no way to observe how the mixing time behaves as the system size increases. However, it is known that the classical Metropolis algorithm converges quickly if the physical system thermalizes – in fact, it may converge even more quickly, because the Metropolis algorithm is allowed to perform unphysical updates in its simulation of thermalization [7]. The resemblance between the classical and the quantum versions of the algorithm gives hope that the quantum Metropolis algorithm will behave similarly.

We now reproduce the entire algorithm from [7], assuming that no errors occur during quantum phase estimation:

Algorithm 6 (Quantum Metropolis Algorithm by Temme *et al.*)

1. *Initialize four quantum registers. The first will hold the quantum states. The second and third registers need to hold r-qubits and will respectively encode the energy of the original state and be used during quantum phase estimation. The last register only needs to hold a single qubit and is used to accept or reject the Metropolis update.*

2. *Reinitialize the last three ancillas and implement the circuit shown in part a of Fig. (4.1).*

3. *Define {C} such that the probability of randomly selecting C is the same as selecting $C^\dagger$. Implement the circuit shown in part b of Fig. (4.1). For energies E_i and E_k, the unitary $W(E_i, E_k)$ is defined as:*

$$W(E_i, E_k) = \begin{pmatrix} \sqrt{1 - f_k^i} & \sqrt{f_k^i} \\ \sqrt{f_k^i} & -\sqrt{1 - f_k^i} \end{pmatrix} \tag{4.5}$$

4. *Measure the last ancilla qubit. If the result is $|1\rangle$ or Q_1, then the update is accepted with probability $\propto |x_k^i \sqrt{f_k^i}|^2$, and the second ancilla qubit should be measured to obtain the energy required for a new iteration of the quantum Metropolis algorithm. If the result is $|0\rangle$ or Q_0, then the update has been rejected. Apply $U^\dagger$ and proceed.*

5. *Apply the circuit in Fig. (4.2) until P_1 is obtained. Then the algorithm has returned to the initial state and the rejection procedure has been completed.*

4.2   Other Gibbs Samplers

A brief discussion of other Gibbs samplers will follow to demonstrate why the quantum Metropolis algorithm is shown preference in this paper. As Brandão *et al.* discussed in their 2016 paper on Gibbs samplers, the quantum Gibbs samplers developed in the past twenty years can be sorted into two types: samplers with explicitly defined runtimes that are exponential in system size, and samplers that should be efficient but lack a bounded convergence time [12]. The quantum Metropolis algorithm belongs to the latter category. We will briefly discuss the Gibbs samplers that fall into the first category, and then expand on the other Gibbs sampler of the second type that offers a slight improvement upon the quantum Metropolis sampling algorithm by Temme *et al.*

The quantum Gibbs sampler developed by Poulin *et al.* falls squarely into the first category [13]. We will point readers interested in the mechanisms of this sampler in the direction of their 2009 paper and forgo describing the workings of their algorithm in favor of simply giving the runtime: $O\left(\sqrt{\frac{D}{\mathcal{Z}(\beta)}}\right)$, where D is the dimension of the Hilbert space and $\mathcal{Z}(\beta)$ is the partition function at of the quantum system at inverse temperature β. The expression inside the big O notation is equivalent to D^{α}, where the scaling exponent is a function of the Helmholtz free energy density and approaches $1/2$ at low temperatures [13]. Since the complexity of this algorithm scales with the Hilbert space dimension, the runtime is still exponential in the system size.

A recent Gibbs sampler developed in 2016 by Chowdhury *et al.* shares a similar complexity with the previous sampler, but enjoys a quadratic speedup in β [14]. The runtime of this algorithm is: $O\left(\sqrt{\frac{D\beta}{\mathcal{Z}(\beta)}}\right)$. It can be deduced from an analysis of the previous algorithm that the runtime of this Gibbs sampler remains exponential in system size.

The quantum-quantum Metropolis algorithm developed by Yung *et al.* is the other Gibbs sampler that is efficient but does not have a clearly bounded convergence time that we will discuss in this paper [15]. Nevertheless, Yung *et al.* demonstrate in their paper that this new sampler enjoys a quadratic speedup of $O(\frac{1}{\sqrt{\Delta}})$, where Δ is the spectral gap of the completely positive map. This speedup comes from the use of a different basis – the quantum-quantum Metropolis algorithm uses pairs of eigenstates that are connected by time-reversal operations as its basis states, thus allowing it to avoid the inefficiency of working around the no-cloning theorem [15]. However, this requirement also makes a classical simulation of the algorithm's

runtime more difficult to implement, which is why we conduct our investigation of quantum semidefinite programming with the quantum Metropolis algorithm. If the outcomes of the quantum Metropolis algorithm are positive, then it is likely that this algorithm is just as, if not more, efficient than the quantum Metropolis sampling scheme.

References

[1] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculation by fast computing machines", The Journal of Chemical Physics **21**, 1087–1092 (1953).

[2] M. Suzuki, *Quantum monte carlo methods in equilibrium and nonequilibrium systems* (Springer Verlag, Berlin Heidelberg, 1987).

[3] F. Verstraete, M. M. Wolf, and J. I. Cirac, "Quantum computation, quantum state engineering, and quantum phase transitions driven by dissipation", Nature Physics **5**, 633–636 (2009).

[4] B. M. Terhal and D. P. DiVincenzo, "Problem of equilibration and the computation of correlation functions on a quantum computer", Physical Review A **61**, 022301 (2000).

[5] D. S. Abrams and S. Lloyd, "Quantum algorithm providing exponential speed increase for finding eigenvalues and eigenvectors", Physical Review Letters **83**, 5162–5165 (1999).

[6] A. Aspuru-Guzik, A. D. Dutoi, P. J. Love, and M. Head-Gordon, "Simulated quantum computation of molecular energies", Science **309**, 1704–1707 (2005).

[7] K. Temme, T. J. Osborne, K. G. Vollbrecht, D. Poulin, and F. Verstraete, "Quantum metropolis sampling", Nature **471**, 87–90 (2011).

[8] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca, "Quantum algorithms revisited", Proceedings of the Royal Society of London A **454**, 339–354 (1998).

[9] C. Marriott and J. Watrous, "Quantum arthur–merlin games", Computational Complexity **14**, 122–152 (2005).

[10] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter, "Elementary gates for quantum computation", Physical Review A **52**, 3457 (1995).

[11] K. Temme, M. J. Kastoryano, M. B. Ruskai, M. M. Wolf, and F. Verstraete, "The χ^2 divergence and mixing times of quantum markov processes", Journal of Mathematical Physics **51**, 122201 (2010).

[12] M. J. Kastoryano and F. G. S. L. Brandão, "Quantum gibbs samplers: the commuting case", Communications in Mathematical Physics **344**, 915–957 (2016).

[13]D. Poulin and P. Wocjan, "Sampling from the thermal quantum gibbs state and evaluating partition functions with a quantum computer", Physical Review Letters **103**, 220502 (2009).

[14]A. N. Chowdhury and R. D. Somma, "Quantum algorithms for gibbs sampling and hitting-time estimation", (2016), arXiv:1603.02940 [quant-ph].

[15]M. Yung and A. Aspuru-Guzik, "A quantum–quantum metropolis algorithm", Proceedings of the National Academy of Sciences, 754–759 (2012).

C h a p t e r   5

SPARSITY AND HAMILTONIAN SIMULATION

The original definition of matrix sparsity is $1 - d$, where the density d is defined as the ratio of the number of nonzero elements in the matrix to the total number of elements. But in a quantum Hamiltonian, especially in a quantum Hamiltonian that can be separated into independent terms and simulated efficiently on a quantum computer, the traditional definition of sparsity holds little physical significance – this is because if a quantum spin-$\frac{1}{2}$ Hamiltonian that acts on n qubits is written as a matrix, then the matrix has dimensions N^2, where $N = 2^n$. Thus the traditional definition of sparsity, where the total number of nonzero elements is defined as $(1 - d)N^2$, makes sparsity a polynomial function of the total Hilbert space size, rather than a function of the system size. This does not make sense physically, and so we discard this definition.

Row sparsity fills the gap left behind. The original concept of row sparsity arose in the Sparse Hamiltonian lemma proposed by Aharonov and Ta-Shma in [1], where they seek to give a condition for the efficient simulation of a Hamiltonian on a quantum computer, but it must be noted that the Hamiltonians described in Lloyd's work are sparse [2]. Indeed, Lloyd requires that the Hamiltonian is composed of a sum of smaller Hamiltonians that act on subsystems; the requirements are not as stringent in the work by Aharonov and Ta-Shma, for they no longer demand that the Hamiltonian take on a tensor product structure. We reproduce the definitions of row sparse and row computable, as well as the Sparse Hamiltonian lemma below [1].

Definition 5.1 (Row Sparse) A Hamiltonian acting on n qubits is said to be row sparse if the number of nonzero elements in each row is bounded polynomially by a function of n.

Definition 5.2 (D Sparse) A Hamiltonian acting on n qubits is said to be D sparse if the number of nonzero elements in each row is bounded by D, where D is a constant that satisfies the constraint $D \leq poly(n)$.

Definition 5.3 (Row Computable) A Hamiltonian is said to be row computable if there is an efficient algorithm that given a row index generates a list containing the values and positions (column index) of the nonzero elements in the given row.

Lemma 5.4 (Sparse Hamiltonian Lemma) A Hamiltonian H acting on n qubits can be efficiently simulated on a quantum computer if H is row sparse, row computable, and $||H|| \leq poly(n)$.

We won't reproduce the entire proof for Lemma 5.4, but we will sketch out its reasoning. The essence of the proof is the decomposition lemma, which we will reproduce below [1]:

Lemma 5.5 (Decomposition Lemma) If H is a D-sparse, row computable Hamiltonian acting on n qubits, then it can be decomposed into $H = \sum_{m=1}^{(D+1)^2 n^6} H_m$, where H_m is a 2×2 combinatorially block diagonal, row sparse, and row computable Hamiltonian that acts on n qubits.

In other words, if a Hamiltonian satisfies the conditions enumerated by the sparse Hamiltonian lemma, then it can be written as a sum of smaller Hamiltonians H_m whose norm is bounded by a polynomial function of the system size. The Hamiltonians H_m are 2×2 combinatorially block diagonal, row sparse, and row computable, and thus can be simulated; the entire Hamiltonian H can then be simulated with Trotter's formula, which approximates e^{-itH} with products of e^{-itH_m} [3].

The methods proposed by Lloyd in 1996 and Aharonov and Ta-Shma in 2003 are the foundations on which more efficient ways of Hamiltonian simulation have been crafted. The method used to simulate the quantum Hamiltonians in the quantum semidefinite programming algorithm by Brandão *et al.* enjoys optimal performance in the all of the relevant parameters – not only does the algorithm by Berry *et al.* have a complexity that is logarithmic in inverse error, it also has a complexity that scales almost linearly with the product of the evolution time, the row sparsity, and the magnitude of the Hamiltonian's largest element [4]. In this case, complexity is defined as both the number of queries the algorithm makes to the oracle that yields the positions and values of the Hamiltonian's nonzero elements, and the number of 2-qubit gates used to simulate the Hamiltonian. Unlike previous methods, this algorithm doesn't decompose the Hamiltonian into the sum of easily simulated terms; rather, it approximates Hamiltonian evolution with steps from a Szegedy quantum walk. Berry *et al.* show that the Hamiltonian can be prepared in a procedure called controlled state preparation, where every eigenstate of the Hamiltonian is mapped onto a linear combination of two eigenstates $|\mu_{\pm}\rangle$ of a quantum walk step U, where U is defined such that its eigenvalues $\mu_{\pm}$ is related to the eigenvalues of the Hamiltonian λ in the following way:

$$\mu_{\pm} = \pm e^{\pm i \arcsin(\lambda/XD)}, \tag{5.1}$$

where X is greater or equal to the magnitude of the Hamiltonian's largest entry, and D is the row sparsity. It can be shown that for small λ/XD, if a linear combination of quantum walk steps is prepared with coefficients generated by Bessel functions, the quantum walk steps give a phase factor proportional to the phase factor generated by Hamiltonian evolution [4].

During the controlled state preparation, the algorithm makes a call to an oracle that takes as its input j the row index and l for the $l - th$ nonzero element in row j and calculates the column index of the $l - th$ nonzero element in the $j - th$ row of a Hamiltonian. It then makes a call to another oracle that given a row index and a column index outputs a nonzero value for the input position.

We will reproduce the complexities of the algorithm as well as the accompanying theorems below, but point readers interested in their proofs to the original paper [4].

Theorem 5.5 For a Hamiltonian acting on n qubits that has a row sparsity of D, a simulation time of t, and an error of ϵ, the Hamiltonian can be simulated with a number of queries that scales as $O\left(\tau \frac{\log(\tau\epsilon)}{\log\log(\tau\epsilon)}\right)$ and a number of 2-qubit gates that scales as $O\left(\tau[n + \log^{5/2}(\tau/\epsilon)]\frac{\log(\tau\epsilon)}{\log\log(\tau\epsilon)}\right)$, where τ is the product of the row sparsity, the magnitude of the largest element in the Hamiltonian, and the simulation time.

The theorem above gives two upper bounds for the number of queries and 2-qubit gates used to simulate the Hamiltonian. The theorem below gives a lower bound for the number of queries [4].

Theorem 5.6 For a given error ϵ, a simulation time $t > 0$, a row sparsity D greater or equal to 2, and a fixed magnitude for its largest magnitude, there exists a Hamiltonian that can be simulated with a number of queries that scales as $\Omega\left(\tau + \frac{\log(\tau\epsilon)}{\log\log(\tau\epsilon)}\right)$.

Berry *et al.* also demonstrate that there can be tradeoffs between the different parameters τ and ϵ by slightly modifying their algorithm so that the following theorem stands [4]:

Theorem 5.7 For any row sparsity D, simulation time t, error ϵ, system size n, and $\alpha \in (0, 1]$, the Hamiltonian can be simulated with a number of queries that scales as $O\left(\tau^{1+\alpha/2} + \tau^{1-\alpha/2}\log(1/\epsilon)\right)$.

The theorems above demonstrate that as long as our random row-sparse Hamiltonians are the results of a black box that efficiently generates the values and positions of

the nonzero elements given the row index and the number of nonzero elements in the indicated row, they can be efficiently simulated with the algorithm outlined in [4]. It is also worth noting that in the algorithm sketched out by Berry *et al.*, the Hamiltonian is always defined by its size n and its row sparsity D, where both n and D are constants. It is possible to classify Hamiltonians in a manner independent of size, i.e. based on the interactions between its spins. The question becomes whether we can classify classes of random Hamiltonians based on their sparsities in a similar fashion. While the convention of defining a Hamiltonian with constants n and D seems to hint that it is physically viable to classify random Hamiltonians based on a constant row sparsity D, the stipulation that $D \leq poly(n)$ suggests that it may be wiser to classify these random Hamiltonians with sparsities that are polynomial functions of the system size. We shall see how the different interpretations of "fixed row sparsity" affect the spectral gap behavior of the random Hamiltonians' quantum Metropolis maps in the following section.

References

[1] D. Aharonov and A. Ta-Shma, "Adiabatic quantum state generation and statistical zero knowledge", Proceedings of the 35th Annual ACM symposium on Theory of Computing (STOC'03), 20–29 (2003).

[2] S. Lloyd, "Universal quantum simulators", Science **273**, 1073–1078 (1996).

[3] H. Trotter, "On the product of semi-groups of operators", Proceedings of the American Mathematical Society **10**, 545–551 (1959).

[4] D. W. Berry, A. M. Childs, and R. Kothari, "Hamiltonian simulation with nearly optimal dependence on all parameters", Proceedings of the 56th IEEE Symposium on Foundations of Computer Science, 792–809 (2015).

Chapter 6

NUMERICAL SIMULATIONS

We enumerated several quantum Gibbs samplers in a previous chapter and discussed in particular how the quantum Metropolis sampling algorithm doesn't have a explicitly defined running time, but may offer an exponential speedup to classical semidefinite programming. There is no theoretical guarantee that the quantum Metropolis algorithm will always thermalize in polynomial time for an arbitrary Hamiltonian – such an ability would signify that the quantum Metropolis algorithm is capable of solving QMA-complete problems. Nevertheless, heuristically it is always possible to guide the updates of the classical Metropolis algorithm in such a fashion that it thermalizes efficiently if the physical system itself thermalizes efficiently; in the absence of reasons of why this should prove different for the quantum Metropolis algorithm, it is not far-fetched to assume that the quantum Metropolis sampling algorithm may offer substantial – even exponential – speedups to quantum Gibbs sampling [1]. Although there is no quantum computer to directly test this heuristic method, we are able to use numerical methods to investigate the behavior of the quantum Metropolis algorithm's running time for arbitrary physical systems. We know that the runtime, or the mixing time, of the quantum Metropolis algorithm scales as the following:

$$m_{mix} \geq O\left(\frac{\ln(1/\epsilon^{mix})}{\Delta}\right) \tag{6.1}$$

where Δ is the spectral gap of the quantum Metropolis stochastic map, or the difference between the map's two largest eigenvalues, and ϵ^{mix} is the mixing error. It can be observed that the runtime scales as the inverse gap of the quantum Metropolis stochastic map; thus, demonstrating that the gap scales polynomially in the physical system's size, or the number of qubits involved in the physical system, is sufficient to show that the runtime scales as an inverse polynomial in system size. Such a finding would indicate that the quantum Metropolis algorithm allows an exponential speedup in the preparation of the Gibbs state for the physical system in question.

Temme *et al.* simulated the runtime of the algorithm for the XX-chain in a transverse field at $T = 0$ and showed that the inverse gap of the quantum Metropolis map scales

as $O(1/N)$, where N is the number of spins in the system. Their results are shown below in Fig. (6.1):
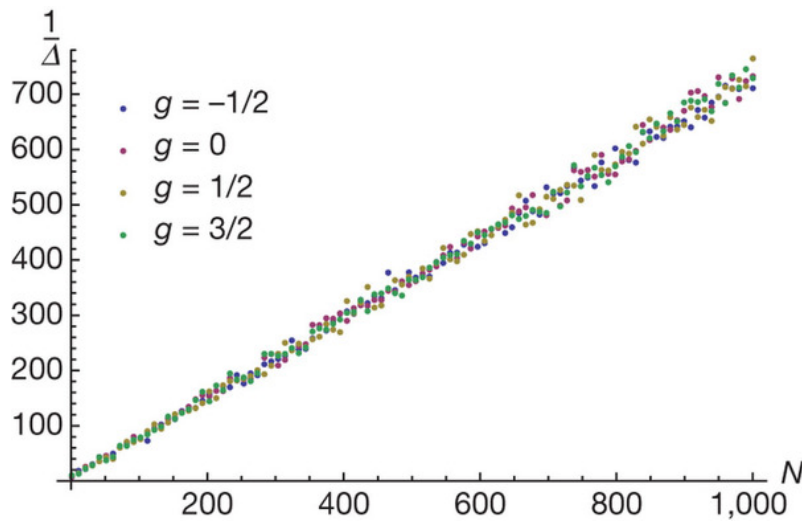
Figure 6.1: **Inverse spectral gap of the quantum Ising model's quantum Metropolis map.** The inverse spectral gap of the quantum Ising model's quantum Metropolis stochastic map, $1/\Delta$, has been plotted as a function of N, the number of spins in the system. The quantum Ising model can be represented as the following Hamiltonian: $\sum_k X_k X_{k+1} + Y_k Y_{k+1} + gZ_k$. A single spin flip is used as the update rule. The linear relationship indicates that the quantum Metropolis algorithm mixes in polynomial time for the specific quantum Ising model used. Reprinted by permission from Macmillan Publishers Ltd: *Nature* 471:87, copyright 2011.

Of course, the Ising model is well-understood; it is also expected that the algorithm will thermalize in polynomial time for realistic physical systems. The difficulty is demonstrating that the gap of the quantum Metropolis map scales polynomially in the system size for the Hamiltonians we are using. Our Hamiltonians are randomly generated but reflect physical systems – they are represented as sparse, Hermitian matrices. We seek to model how the spectral gap of randomly generated Hamiltonians behaves as the system size increases linearly – if the relationship remains polynomial, then the quantum Metropolis sampler can be used to efficiently prepare Gibbs states for the quantum semidefinite programming algorithm, and thus offers an exponential speedup to semidefinite programming.

6.1   Defining Fixed Sparsity

We start by modeling the linear combinations of the input constraint matrices as Hamiltonians, and simulating the behavior of the spectral gaps of the quantum Metropolis maps generated by these Hamiltonians in system size. Since the input

constraint matrices are randomly generated, these Hamiltonians too must be randomly generated; in order for reasonable trends to emerge from the data, it is prudent to classify our Hamiltonians by sparsity. Thus the task at hand becomes modeling the quantum Metropolis map's spectral gap for Hamiltonians of fixed sparsity.

Fixed sparsity is easy to define for matrices when they are not discussed in the context of quantum computing – it is simply the density of the matrices. The density of the matrix can be defined as the fraction of the matrix's total elements that are nonzero. Thus a square matrix that has a total of n^2 elements has a density of 0.3, if there are a total of $\lfloor 0.3n^2 \rfloor$ nonzero elements in the matrix. However, classifying our Hamiltonians with this definition of fixed sparsity yields nonsensical results when the spectral gap's behavior is simulated for varying system size. The inverse spectral gap will appear to decrease exponentially for systems ranging from three to nine qubits, indicating that the mixing time of the quantum Metropolis sampler decreases exponentially as the system size increases from three qubits to nine qubits. At nine qubits, the inverse gap appears to start increasing – not enough data was gathered to determine the behavior of the gap for systems larger than ten qubits, but no more data was needed to determine that this definition of sparsity does not pertain to Hamiltonians used in quantum computing. The results are counterintuitive – the mixing time should increase with the system size.

Since the Hamiltonians will be simulated using the methods discussed in the chapter on sparsity, it is logical that we will classify our Hamiltonians using row sparsity. Indeed, the runtime of the quantum semidefinite programming algorithm discussed in this paper is linear in the row-sparsity of the input matrices. However, a second question arises: how is fixed row sparsity defined in the context of varying system size? We know that a Hamiltonian H acting on n qubits is row sparse if the number of non-zero entries in each row is polynomially bounded in n; we also know that a Hamiltonian is $d - sparse$ if each row has at most d non-zero elements [2]. If we define fixed row sparsity as a constant d, then we sort Hamiltonians by the maximum number of nonzero elements it has in each row. If we define fixed row sparsity as a fixed polynomial that is a function of the system size n, then we sort Hamiltonians by the polynomial function governing the row sparsity. The spectral gap behavior for both interpretations of fixed row sparsity will be presented and discussed in the section on results. Based on the results, it would appear that defining the fixed row sparsity as a fixed polynomial function seems to be the correct course of action.

6.2 Simulating the Runtime

We will sketch a outline of the algorithm used to simulate the runtime of the quantum Metropolis sampler for varying system size and sparsity, and give a brief discussion on its implementation. The chapter on the quantum Metropolis sampling algorithm discusses in detail how the amplitudes $x_l^k \sqrt{f_l^k}$ correspond to the classical definition of transition probabilities; in our case, they represent the probability of transitioning from initial state k to the state l. Given the following:

$$f_l^k = \min(1, \exp(-\beta(E_k - E_l))), \tag{6.2}$$

we can write our quantum Metropolis stochastic map as:

$$N(k, l) = \exp(-\frac{\beta(E_l - E_k)}{2})M(k, l), \tag{6.3}$$

where $M(k, l)$ is defined as:

$$M(k, l) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{4} \sum_{j=0}^{3} |\langle E_k| \sigma_j^i |E_l \rangle|^2 \min(1, \exp(-\beta(E_k - E_l)), \tag{6.4}$$

In short, we have generated a Markov chain using the eigenstates of a Hamiltonian acting on n qubits. The σ_j^i in our stochastic map is defined as a Pauli operator where the identity operator $\sigma_0 = I_2$ operates on all of the qubits except for the i-th qubit; instead, σ_j operates on the i-th qubit, where σ_j is chosen from the Pauli matrices $\{I, X, Y, Z\}$. The Pauli operator σ_j^i is the tensor product of the identity matrices and σ_j. We generate a total of 2^n eigenstates; thus, the indices k and l both range from $[1, 2^n]$.

Our algorithm is as follows:

Algorithm 7 (Quantum Metropolis Map Spectral Gap Estimation)

Set n as the number of qubits. Let sparsity be defined as $s \leq poly(n)$. Let $\beta = 0.00366300366$. Let T be the number of iterations. For $t = 1, ..., T$:

1. *Generate a row-sparse Hamiltonian H with the given n and s.*

2. *Diagonalize H and find all 2^n eigenvalues and eigenvectors.*

3. *Compute the quantum Metropolis stochastic map $N(k, l)$ as defined above.*

4. *Compute the spectral gap by finding the largest two eigenvalues of the quantum Metropolis stochastic map.*

It is trivial to generate a random sparse Hamiltonian, but less trivial to generate a random row-sparse and row-computable Hamiltonian. This is because we must ensure that the number of nonzero elements varies from row to row while staying within the limits defined by the row sparsity. We use random number generators to determine the positions and values of the nonzero elements in each row, which meet the requirements of the black box used to specify Hamiltonians that can be efficiently simulated on quantum computers [3].

We will not attempt to generate Hamiltonians where the value and position of each nonzero element is determined by an algorithm that runs in $polylog(N)$, where N is the system size [2]. The goal of our paper is to model the spectral gap's behavior for random row-sparse Hamiltonians, because we wish to see whether the quantum Metropolis sampling algorithm is a good heuristic method to use during the update step in a quantum semidefinite program solver – such a solver should work for a general semidefinite program, where the number and contents of the input matrices vary arbitrarily so long as they remain positive semidefinite.

The algorithm for generating row-sparse Hamiltonians is as follows:

Algorithm 8 (Row Sparse Hamiltonian Generation)

Set n as the number of qubits. Let sparsity be defined as $s \leq poly(n)$.

1. *Initialize a sparse $2^n x 2^n$ matrix S.*

2. *Determine the number of nonzero elements in each row by sampling from $\mathcal{U}(0, s)$ for each row and store the results in a vector $\boldsymbol{v}$.*

3. *For $t = 1, 2, ...2^n$:*

   - *Assign one-dimensional indices to the elements in the cross formed by the t-th row and column. Determine the positions of the nonzero elements by sampling v_t numbers from $\mathcal{U}(0, 2^{n+1} - 1)$.*

   - *Generate the values of the nonzero elements by sampling from $\mathcal{U}(0, 1)$ and assign the values to the one-dimensional indices.*

   - *Convert the one-dimensional indices to two-dimensional indices in the form of (i, j). Insert the values into the S according to their assigned coordinates.*

4. *Compute the transpose of the matrix S^T and add it to itself to yield $H = S + S^T$.*

5. *For $t = 1, 2, ...2^n$:*

   - *Determine the number of nonzero elements in row t and find their row indices.*

   - *Determine the number of nonzero elements that need to be removed so that the maximum number of nonzero elements in each row is bounded by the sparsity parameter.*

   - *Randomly shuffle the row indices and remove the excess nonzero elements in that order. If (i, j) is removed from the matrix, then (j, i) must also be removed to ensure that the matrix remains Hermitian.*

It must be noted that step 3.2 of the algorithm for generating random row-sparse Hamiltonians can be substituted so that the values of the nonzero elements are sampled from $\mathcal{N}(0, 1)$. However, in order to ensure that the Hamiltonian is positive semidefinite, since the input matrices for our semidefinite program are positive semidefinite, an identity matrix multiplied by a suitable scalar should be added to

the Hamiltonian. To maximize efficiency, we sampled from $\mathcal{U}(0, 1)$ in our program. To ensure that the results do not vary, the spectral gap's behavior in system size is modeled for two polynomial sparsities and two constant sparsities when the nonzero elements are sampled from $\mathcal{N}(0, 1)$.

Optimizing the Implementation

The algorithm used to model the spectral gap behavior was implemented in C++. The following C++ libraries were used: Armadillo 7.200.2 (Plutocratic Climate Change Denialist), ARPACK, SuperLU 5.2.1, OpenBLAS, and LAPACK [4]. In order to optimize the implementation, the sparse matrix structure is used for most of the row-sparse Hamiltonian generation procedure. However, after the transpose of the sparse matrix is added to itself, the sparse matrix is converted to a dense matrix. The sparse matrix eigensolver operates faster than the dense matrix eigensolver, but only if the eigensolver is solving for a limited number of eigenvalues and eigenvectors. Since the generation of the Markov transition matrix requires all the eigenvalues of the sparse matrix, it is more efficient to construct the Hamiltonian as a sparse matrix and then convert it to a dense matrix before diagonalizing. The eigenvectors and the transposes of the eigenvectors are stored in matrices, which are then converted to complex matrices; we will respectively denote the matrices containing the eigenvectors and the transposed eigenvectors as $\mathbf{V}$ and $\mathbf{V}^T$. The Pauli operators are generated as complex matrices – we denote the Pauli operator where σ_j operates on the target qubit as $\mathbf{P}_j$. The target qubit is the qubit where the default operator is not the identity matrix.

Armadillo is optimized so that calculating the Hadamard product using element-wise multiplication is an efficient process. Thus we can compute the stochastic map using the following operations:

$$\sum_{i=1}^{n} \frac{1}{4} \sum_{j=0}^{3} |\langle E_k | \sigma_j^i | E_l \rangle|^2 = \mathbf{V}^T \mathbf{P}_0 \mathbf{V} \circ (\mathbf{V}^T \mathbf{P}_0 \mathbf{V})^* + \mathbf{V}^T \mathbf{P}_1 \mathbf{V} \circ (\mathbf{V}^T \mathbf{P}_1 \mathbf{V})^*$$
$$+ \mathbf{V}^T \mathbf{P}_2 \mathbf{V} \circ (\mathbf{V}^T \mathbf{P}_2 \mathbf{V})^* + \mathbf{V}^T \mathbf{P}_3 \mathbf{V} \circ (\mathbf{V}^T \mathbf{P}_3 \mathbf{V})^*, \tag{6.5}$$

where $(\mathbf{V}^T \mathbf{P}_j \mathbf{V})^*$ denotes the complex conjugate of the matrix $\mathbf{V}^T \mathbf{P}_j \mathbf{V}$, and $\mathbf{A} \circ \mathbf{B}$ denotes the Hadamard product of matrices $\mathbf{A}$ and $\mathbf{B}$. Since the eigenvalues are stored in a vector, we may denote this vector as $\mathbf{U}$ and calculate:

$$\exp\left(-\beta\left(E_k - E_l\right)\right) = \exp\left(-\beta\mathbf{U}\right)\left(\exp\left(\beta\mathbf{U}\right)\right)^T. \tag{6.6}$$

We then compute $\min(1, \exp\left(-\beta\left(E_k - E_l\right)\right)$ by finding all the elements that are larger than 1 in the matrix defined above and replacing them with ones. We can apply this matrix, as well as the Boltzmann weights, to $\sum_{i=1}^{n} \frac{1}{4} \sum_{j=0}^{3} |\langle E_k| \sigma_j^i |E_l\rangle|^2$ through element-wise multiplication. Once the quantum Metropolis stochastic map is completely generated, it is converted back to a sparse matrix. The sparse matrix eigensolver is then used to compute the two largest eigenvalues of the quantum Metropolis map.

6.3 Spectral Gap Behavior

We will first examine how the spectral gap behaves when the system size is increased linearly from three to ten qubits, and the row sparsity is fixed as a constant. We will mainly examine Hamiltonians where the nonzero values are generated from $\mathcal{U}(0, 1)$, but two plots of the spectral gap behavior in system size will be attached for fixed sparsities of $s = 3$ and $s = 6$ when the nonzero elements of the Hamiltonians are sampled from $\mathcal{N}(0, 1)$. The differences do not appear to be significant, as will be demonstrated below. We will then examine how the spectral gap and the inverse spectral gap behave when the system size is increased linearly from three to ten qubits, but the row sparsity is fixed as a polynomial function of the system size. Once again, we will mainly examine Hamiltonians where the nonzero values are generated from $\mathcal{U}(0, 1)$, but two plots of the spectral gap behavior in system size will be attached for fixed sparsities of $s = n$ and $s = n^2 - n + 1$ when the nonzero elements of the Hamiltonians are sampled from $\mathcal{N}(0, 1)$. Finally, the spectral gap behavior for a fixed system size and varying sparsity will be examined for system sizes $n = 6$ and $n = 8$. Further details about how the data was gathered – and the limitations of our methods – can be found in Appendices C through E.

Spectral Gap Behavior for Varying System Size and Constant Row Sparsities
It is apparent from the graphs below that the spectral gaps appear to decrease and then start increasing from fixed sparsities of $s = 6$ and $s = 8$, and steadily increase for a fixed sparsity of $s = 3$. An increasing spectral gap indicates that the running time decreases as the system size increases, which does not make sense physically. Thus we discard the notion of using a fixed constant as row sparsity for increasing system sizes, and turn to using a polynomial function of the number of qubits to determine row sparsity.

Figure 6.2: The spectral gap behavior is modeled for varying system size from three to ten qubits for a fixed row sparsity of $s = 1$. The values of the nonzero elements were generated from a uniform distribution $\mathcal{U}(0, 1)$. The mean spectral gaps for the different system sizes are used as the data points. We use a constant fit of $y(x) = a$ and obtain $a = 0.0588073$, $\sigma_a = 0.00368339$, and $\chi^2/(n-1) = 0.0582212$.

Figure 6.3: The spectral gap behavior is modeled for varying system size from three to ten qubits for a fixed row sparsity of $s = 3$. The values of the nonzero elements were generated from a uniform distribution $\mathcal{U}(0, 1)$. The mean spectral gaps for the different system sizes are used as the data points. We use a linear fit of $y(x) = a + bx$ and obtain $a = 0.224419$, $\sigma_a = 0.0620334$, $b = 0.00286737$, $\sigma_b = 0.00722124$, and $\chi^2/(n-2) = 0.0265796$.

Figure 6.4: The spectral gap behavior is modeled for varying system size from three to ten qubits for a fixed row sparsity of $s = 3$. The values of the nonzero elements were generated from a normal distribution $\mathcal{N}(0, 1)$. The mean spectral gaps for the different system sizes are used as the data points. We use a linear fit of $y(x) = a + bx$ and obtain $a = 0.217747$, $\sigma_a = 0.0559457$, $b = 0.00277423$, $\sigma_b = 0.00637303$, and $\chi^2/(n - 2) = 0.0437704$.
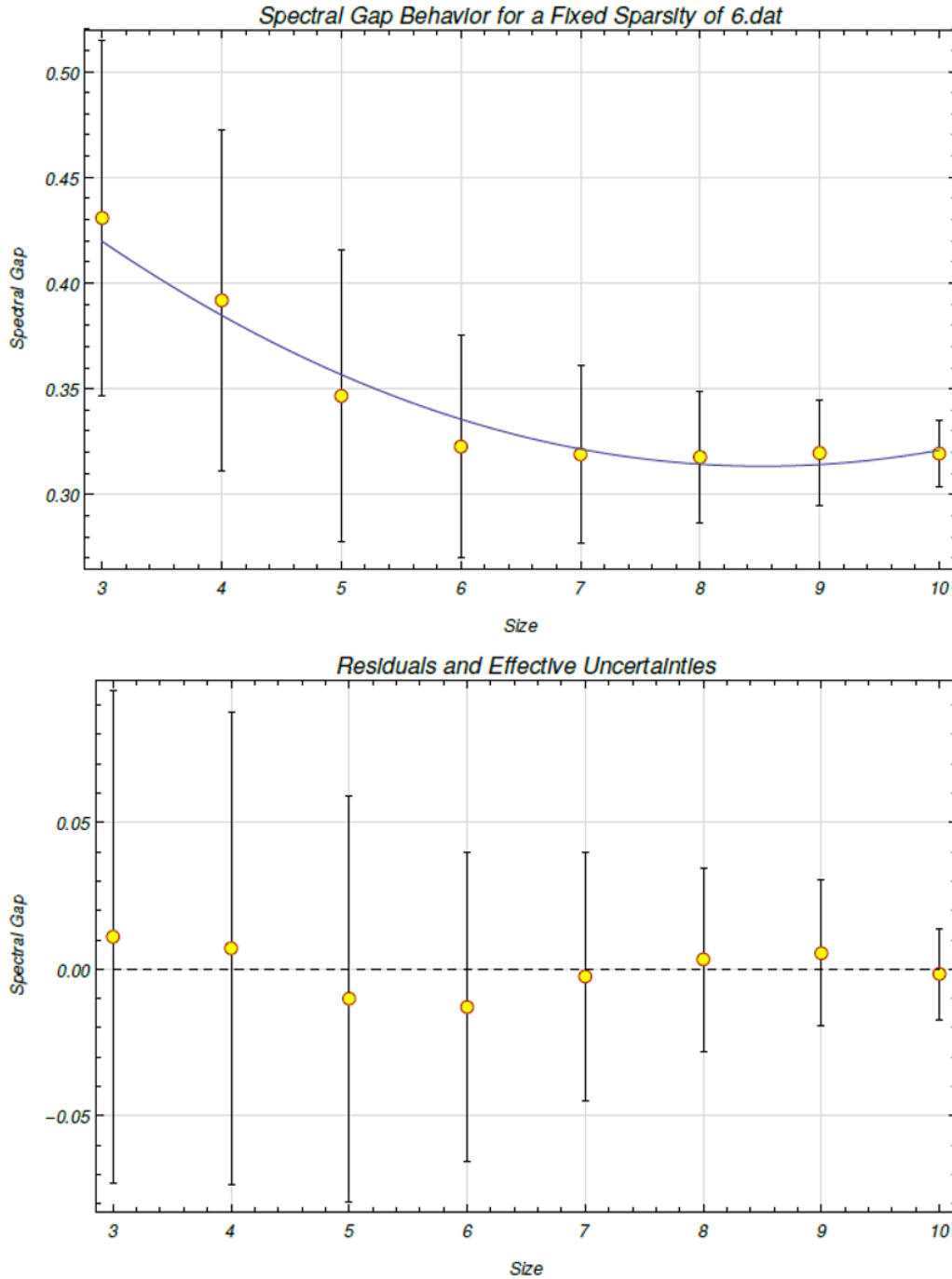
Figure 6.5: The spectral gap behavior is modeled for varying system size from three to ten qubits for a fixed row sparsity of $s = 6$. The values of the nonzero elements were generated from a uniform distribution $\mathcal{U}(0, 1)$. The mean spectral gaps for the different system sizes are used as the data points. We use a quadratic fit of $y(x) = a + bx + cx^2$ and obtain $a = 0.566395$, $\sigma_a = 0.174746$, $b = -0.0593442$, $\sigma_b = 0.0485003$, $c = 0.00348011$, $\sigma_c = 0.00322192$, and $\chi^2/(n - 3) = 0.0358664$.
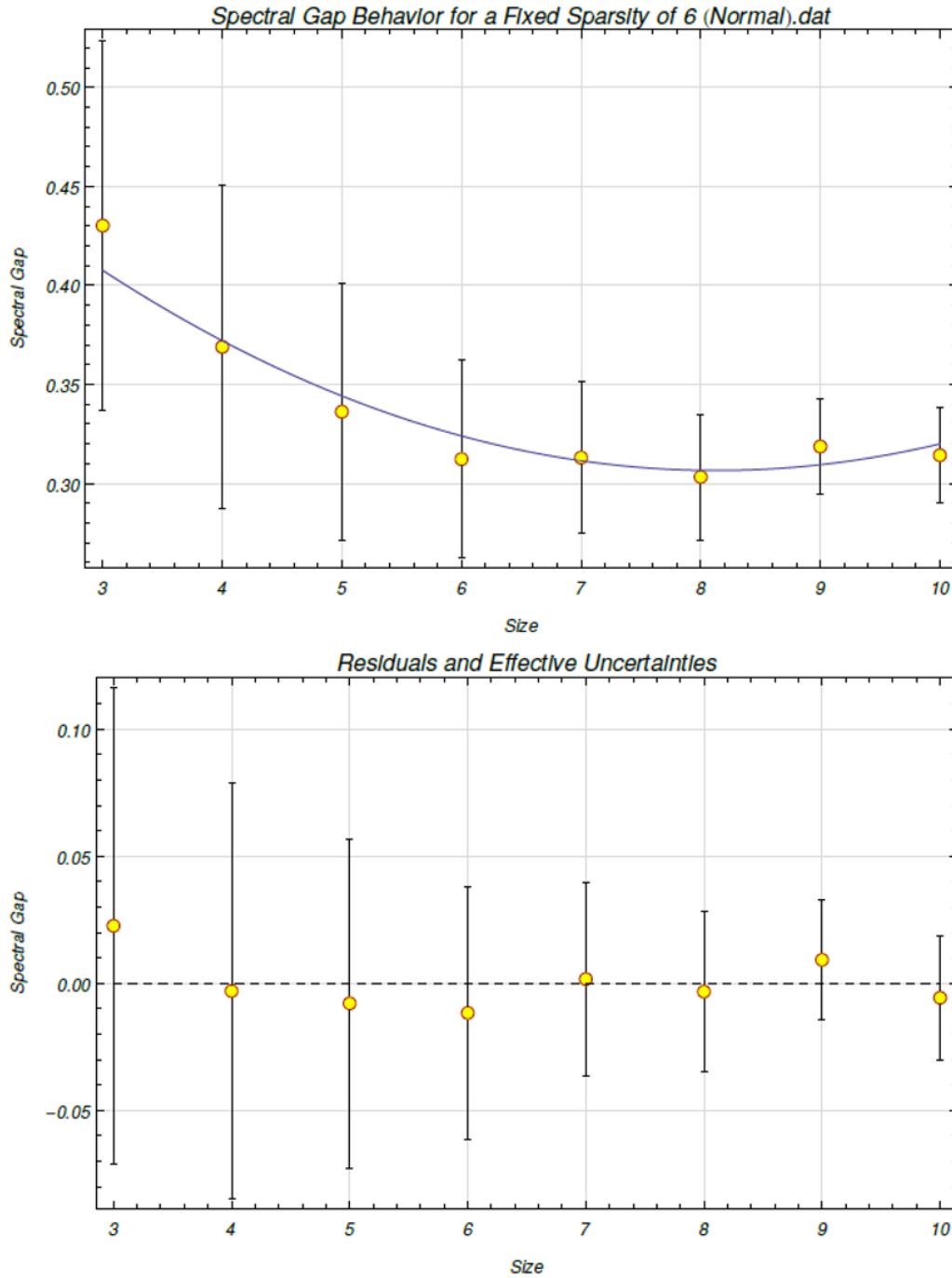
Figure 6.6: The spectral gap behavior is modeled for varying system size from three to ten qubits for a fixed row sparsity of $s = 6$. The values of the nonzero elements were generated from a normal distribution $\mathcal{N}(0, 1)$. The mean spectral gaps for the different system sizes are used as the data points. We use a quadratic fit of $y(x) = a + bx + cx^2$ and obtain $a = 0.560406$, $\sigma_a = 0.190018$, $b = -0.0624437$, $\sigma_b = 0.0533355$, $c = 0.00384023$, $\sigma_c = 0.00360865$, and $\chi^2/(n-3) = 0.0696112$.
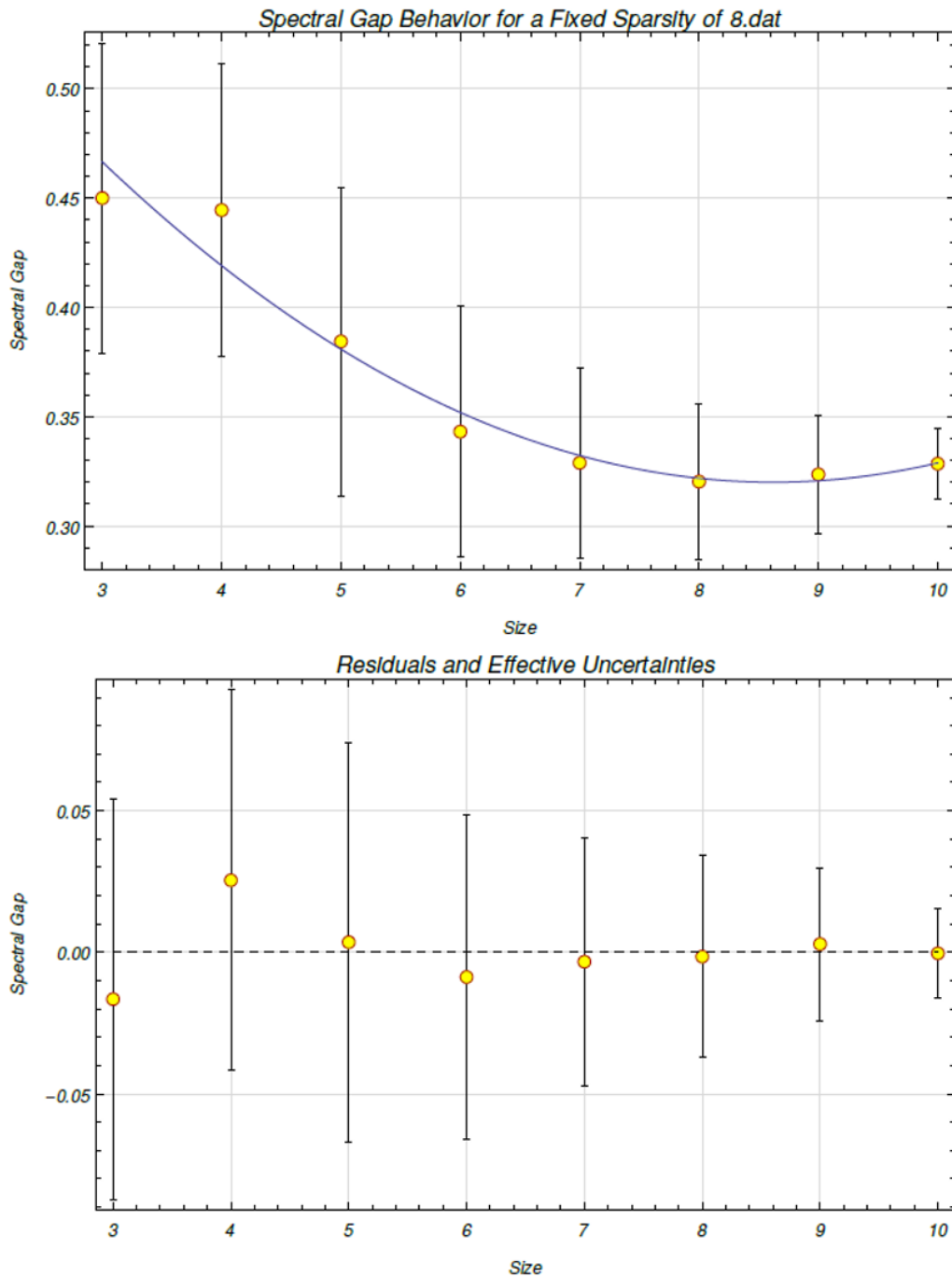
Figure 6.7: The spectral gap behavior is modeled for varying system size from three to ten qubits for a fixed row sparsity of $s = 8$. The values of the nonzero elements were generated from a uniform distribution $\mathcal{U}(0, 1)$. The mean spectral gaps for the different system sizes are used as the data points. We use a quadratic fit of $y(x) = a + bx + cx^2$ and obtain $a = 0.664627$, $\sigma_a = 0.159184$, $b = -0.0799266$, $\sigma_b = 0.0461287$, $c = 0.00463503$, $\sigma_c = 0.00314$, and $\chi^2/(n-3) = 0.048682$.

Spectral Gap Behavior for Varying System Size and Fixed Polynomial Row Sparsities

There appears to be some odd behavior for systems of three and four qubits, but that may be due to large uncertainties in the spectral gap behavior for very small systems. As the system sizes increase, the uncertainties decrease; however, we only have data going up to ten qubits. The χ^2 values are within a reasonable range. Nevertheless, while the results imply that the spectral gaps of the quantum Metropolis stochastic map behave as desired, i.e. polynomially in system size, we cannot say decisively that the Gibbs states of random row-sparse Hamiltonians can be efficiently prepared on a quantum computer using quantum Metropolis sampling for all system sizes.
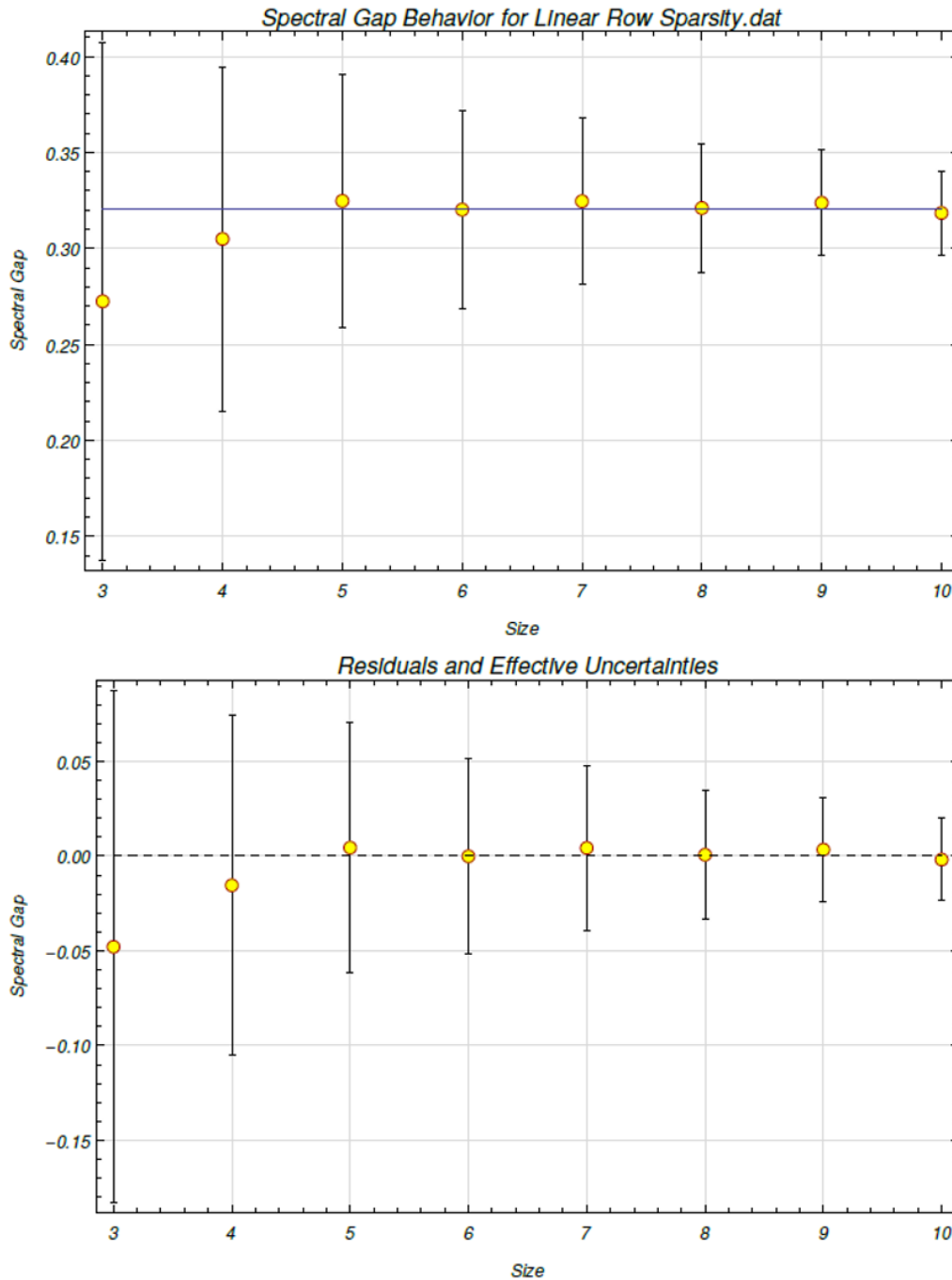
Figure 6.8: The spectral gap behavior is modeled for varying system size from three to ten qubits for a linear row sparsity of $s = n$, where n is the number of qubits. The values of the nonzero elements were generated from a uniform distribution $\mathcal{U}(0, 1)$. The mean spectral gaps for the different system sizes are used as the data points. We use a constant fit of $y(x) = a$ and obtain $a = 0.320328$, $\sigma_a = 0.0133158$, and $\chi^2/(n-1) = 0.027536$.
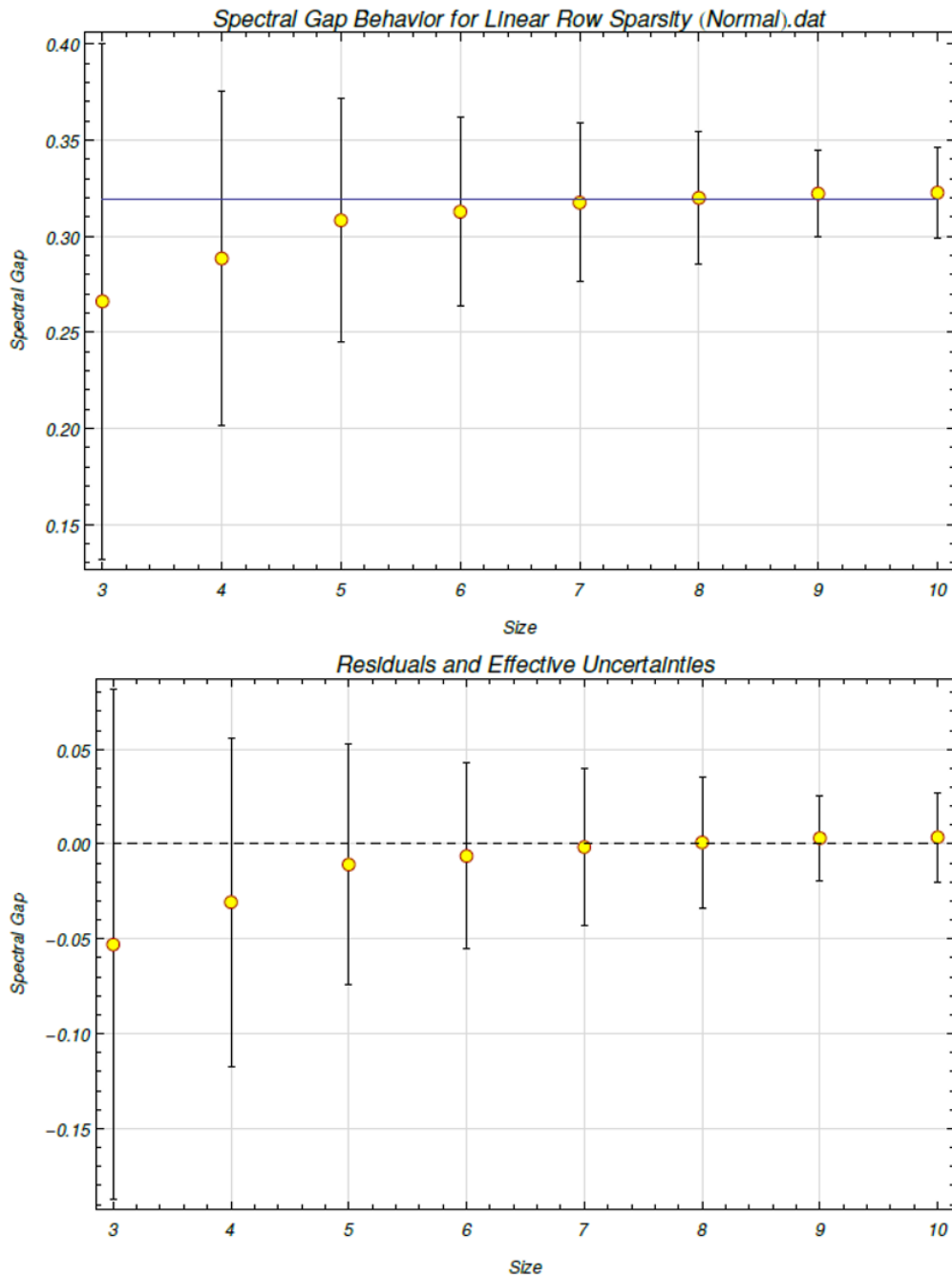
Figure 6.9: The spectral gap behavior is modeled for varying system size from three to ten qubits for a linear row sparsity of $s = n$. The values of the nonzero elements were generated from a normal distribution $\mathcal{N}(0, 1)$. The mean spectral gaps for the different system sizes are used as the data points. We use a constant fit of $y(x) = a$ and obtain $a = 0.31912$, $\sigma_a = 0.0129121$, and $\chi^2/(n - 1) = 0.0528075$.
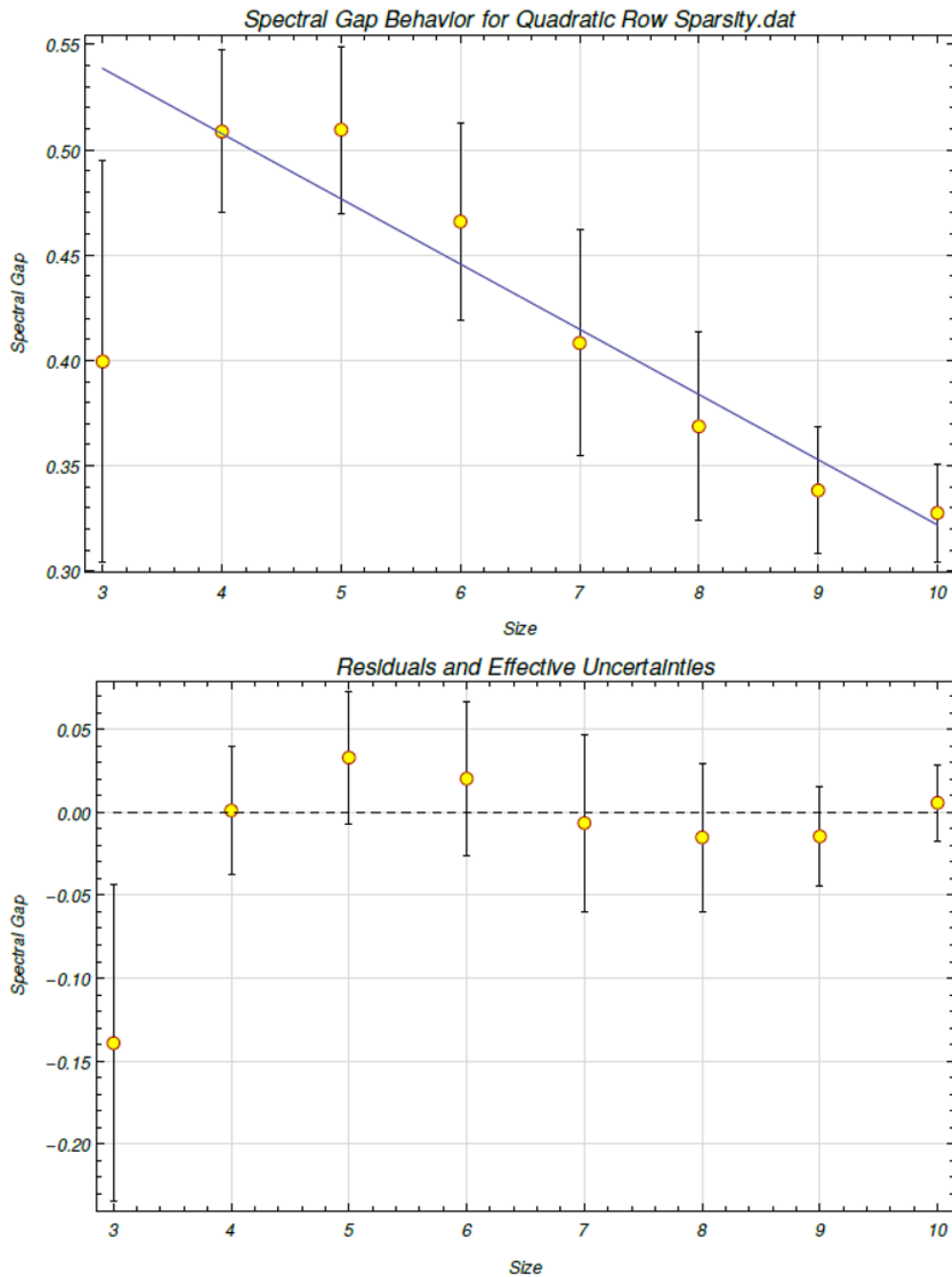
Figure 6.10: The spectral gap behavior is modeled for varying system size from three to ten qubits for a quadratic row sparsity of $s = n^2 - 4$. The values of the nonzero elements were generated from a uniform distribution $\mathcal{U}(0, 1)$. The mean spectral gaps for the different system sizes are used as the data points. We use a linear fit of $y(x) = a + bx$ and obtain $a = 0.631437$, $\sigma_a = 0.0467774$, $b = -0.0309497$, $\sigma_b = 0.00580634$, and $\chi^2/(n-2) = 0.573525$.
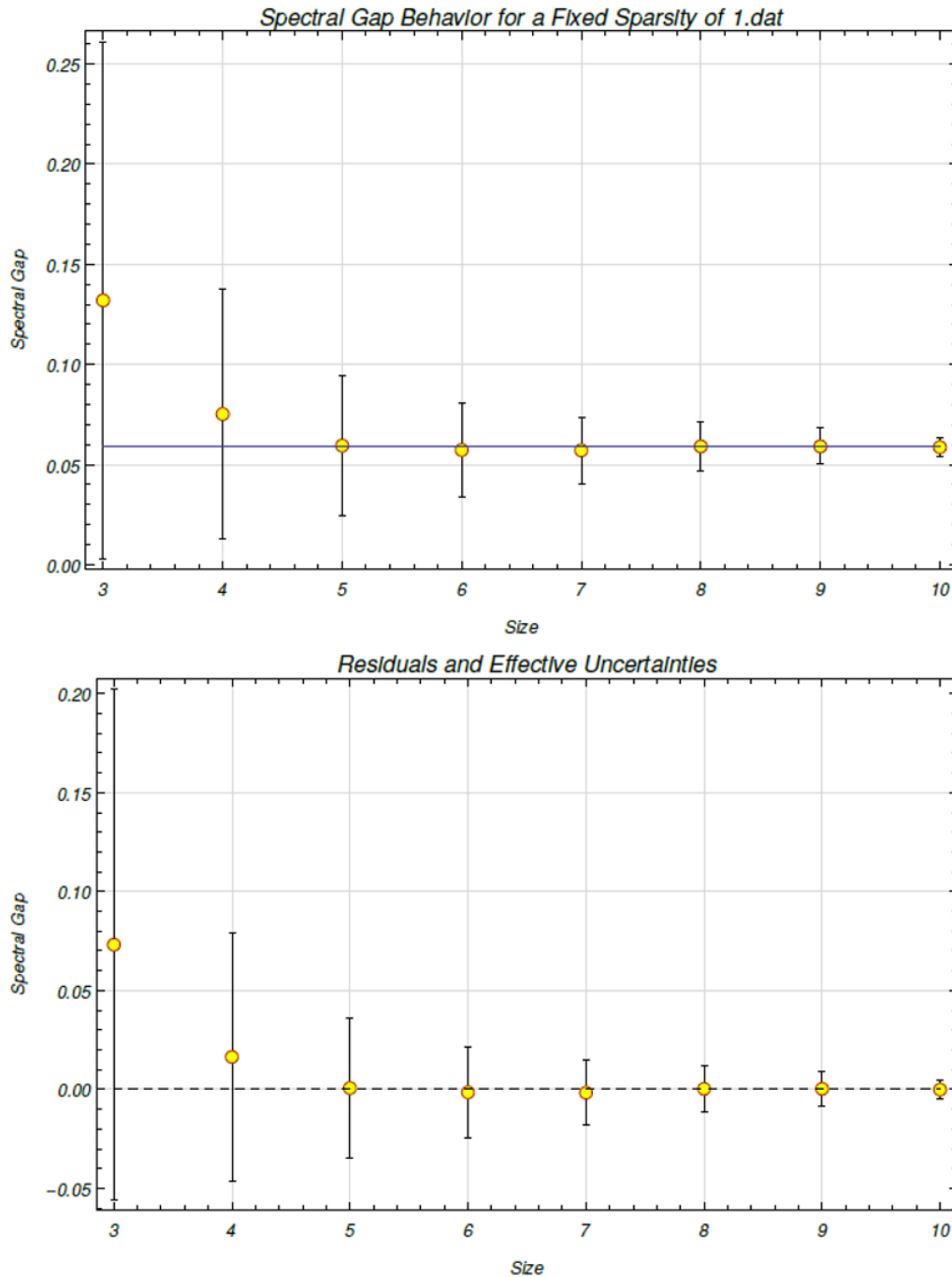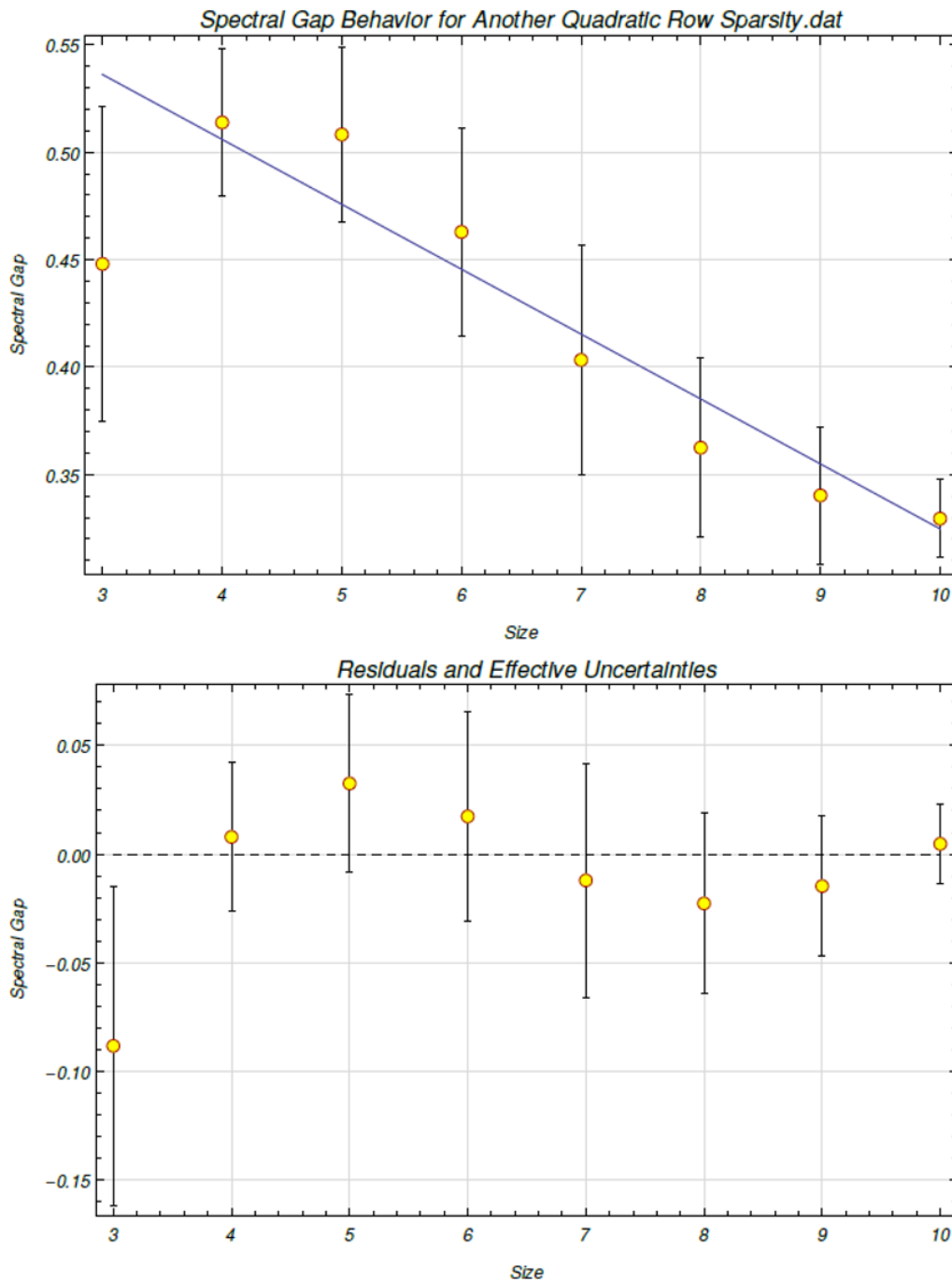
Figure 6.11: The spectral gap behavior is modeled for varying system size from three to ten qubits for a quadratic row sparsity of $s = n^2 - n + 1$. The values of the nonzero elements were generated from a uniform distribution $\mathcal{U}(0, 1)$. The mean spectral gaps for the different system sizes are used as the data points. We use a linear fit of $y(x) = a + bx$ and obtain $a = 0.626823$, $\sigma_a = 0.0421773$, $b = -0.0302081$, $\sigma_b = 0.0050857$, and $\chi^2/(n - 2) = 0.480315$.
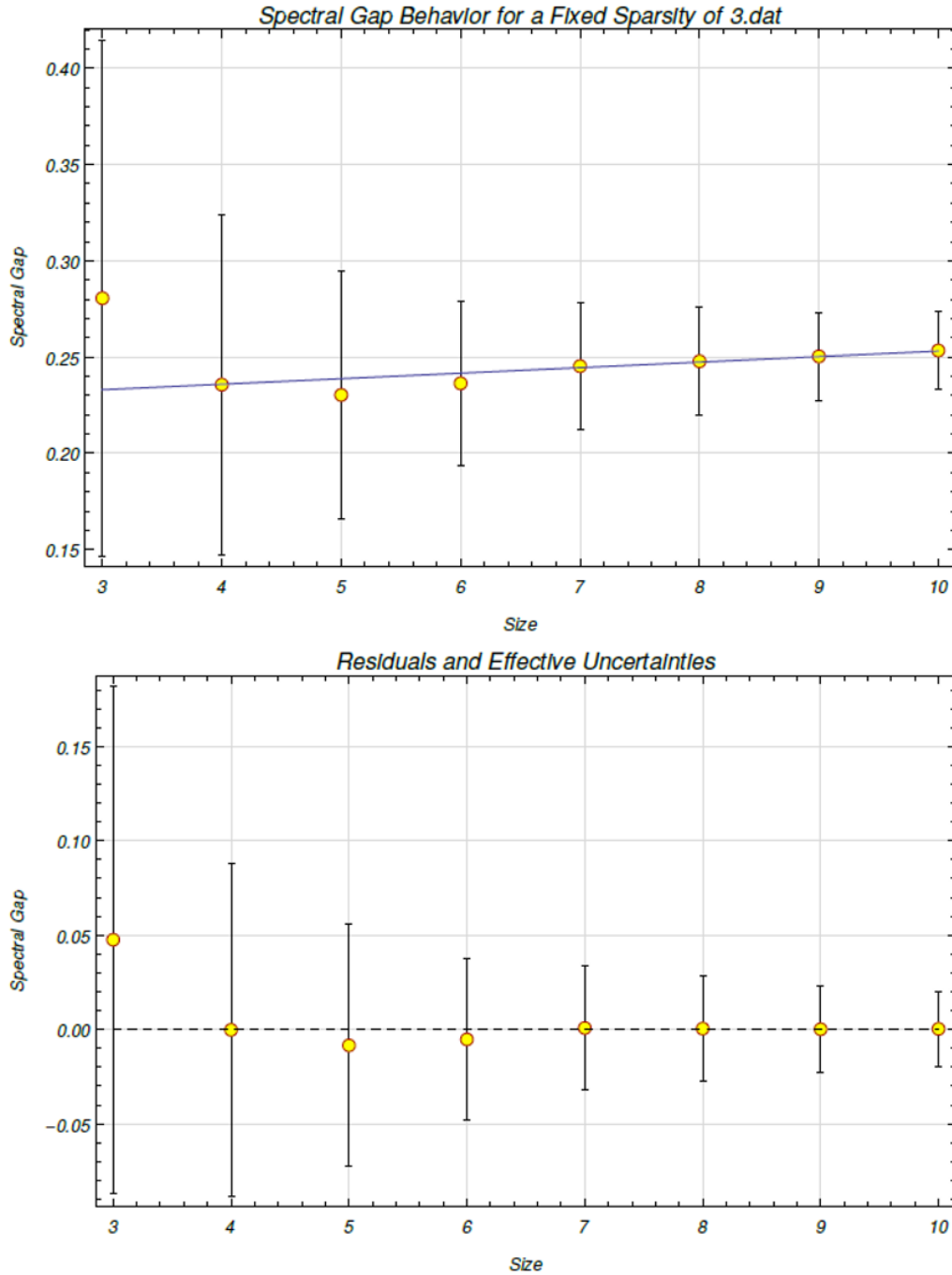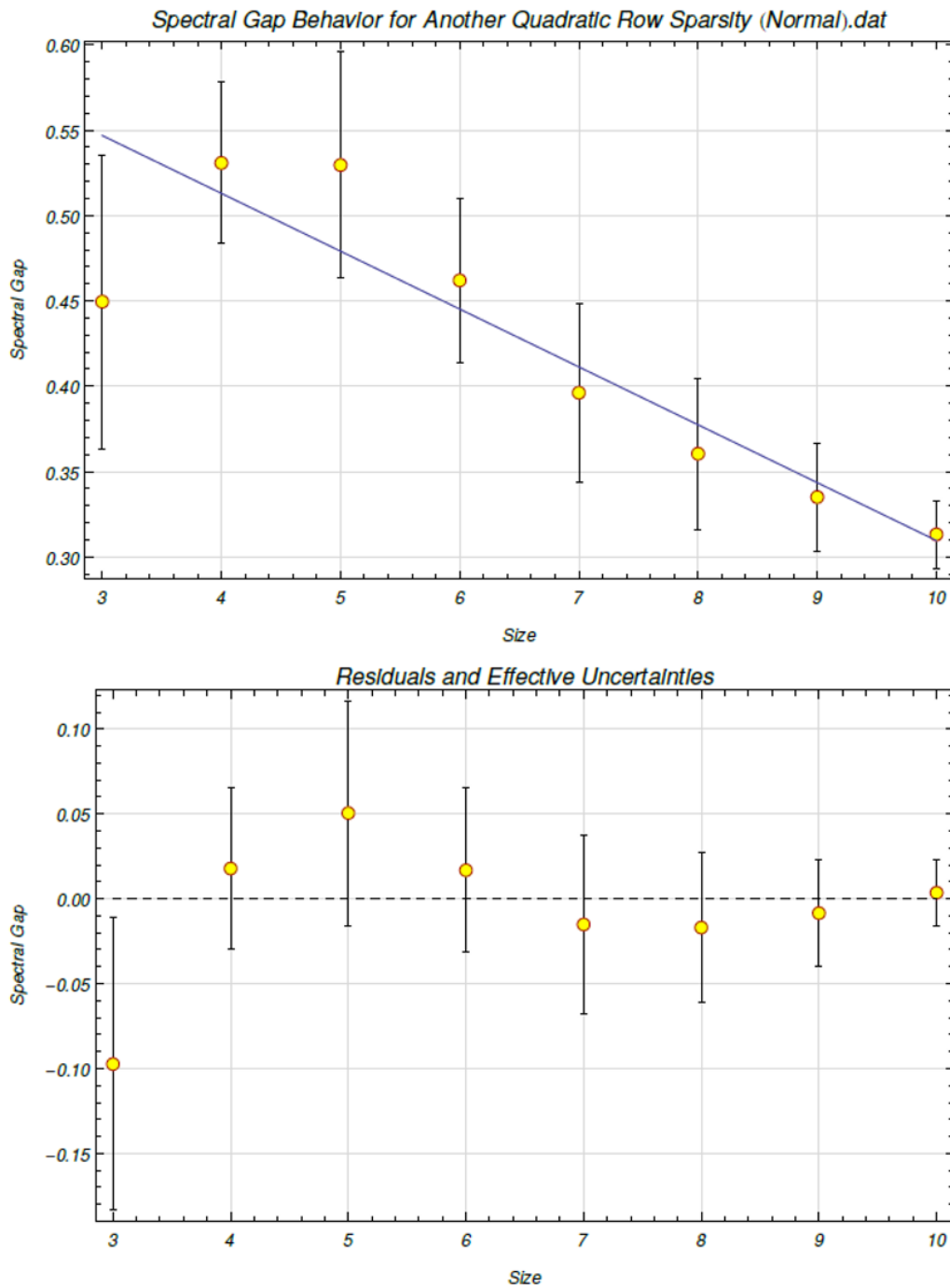
Figure 6.12: The spectral gap behavior is modeled for varying system size from three to ten qubits for a quadratic row sparsity of $s = n^2 - n + 1$. The values of the nonzero elements were generated from a normal distribution $\mathcal{N}(0, 1)$. The mean spectral gaps for the different system sizes are used as the data points. We use a linear fit of $y(x) = a + bx$ and obtain $a = 0.648627$, $\sigma_a = 0.0538921$, $b = -0.0339071$, $\sigma_b = 0.00628929$, and $\chi^2/(n - 2) = 0.409645$.
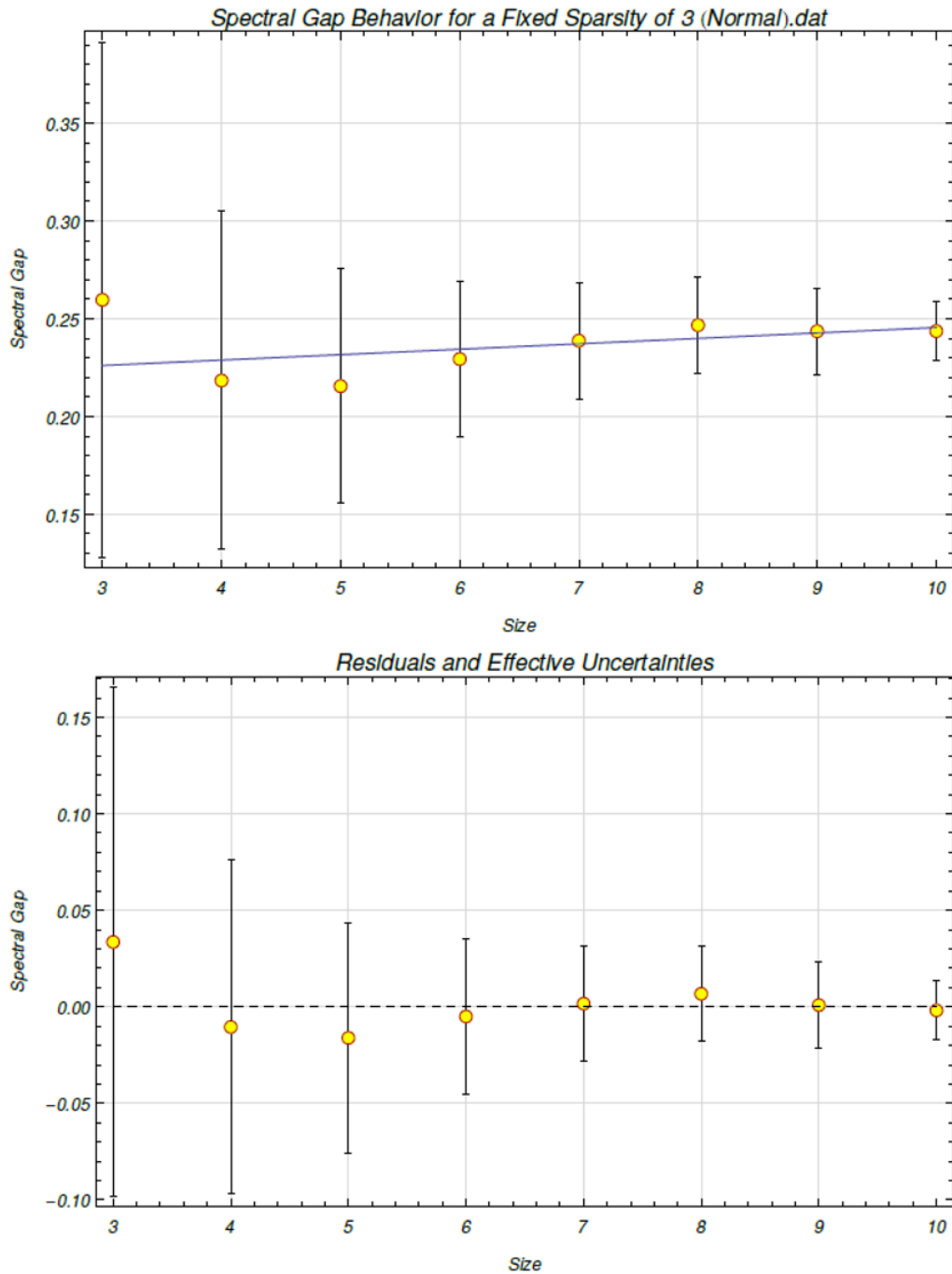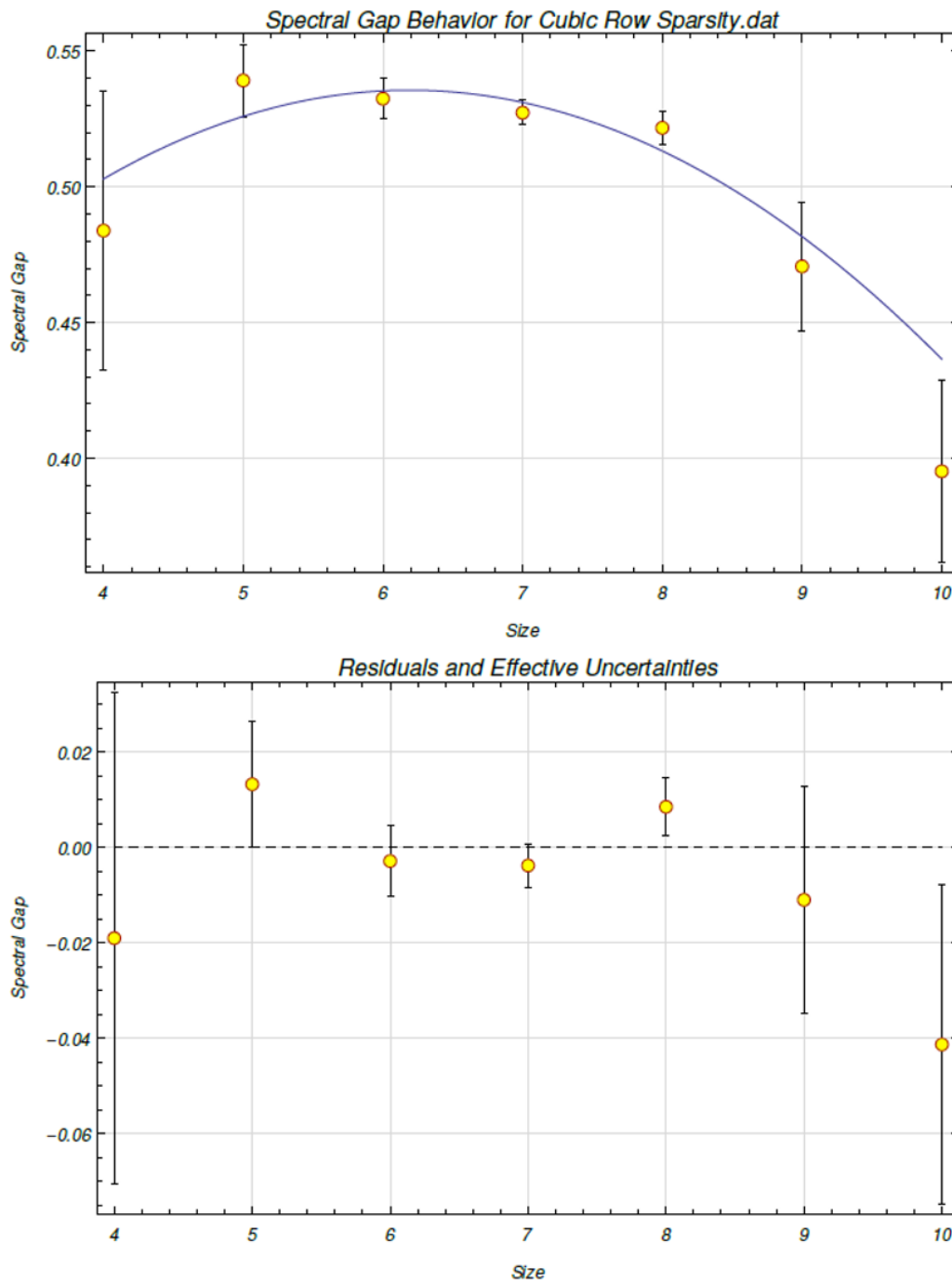
Figure 6.13: The spectral gap behavior is modeled for varying system size from three to ten qubits for a cubic row sparsity of $s = \lfloor 0.5n^3 - n^2 - n - 2 \rfloor$. The values of the nonzero elements were generated from a uniform distribution $\mathcal{U}(0, 1)$. The mean spectral gaps for the different system sizes are used as the data points. We use a quadratic fit of $y(x) = a + bx + cx^2$ and obtain $a = 0.274312$, $\sigma_a = 0.110236$, $b = 0.0844276$, $\sigma_b = 0.0317495$, $c = -0.00682071$, $\sigma_c = 0.00227522$, and $\chi^2/(n-3) = 1.41193$.

Spectral Gap Behavior for Fixed System Size and Varying Row Sparsities

The spectral gap behavior for a fixed system size and varying row sparsity is plotted below for six qubits and eight qubits; it appears from the graphs that the spectral gap behaves as a logarithmic function of the row sparsity for a fixed system size.
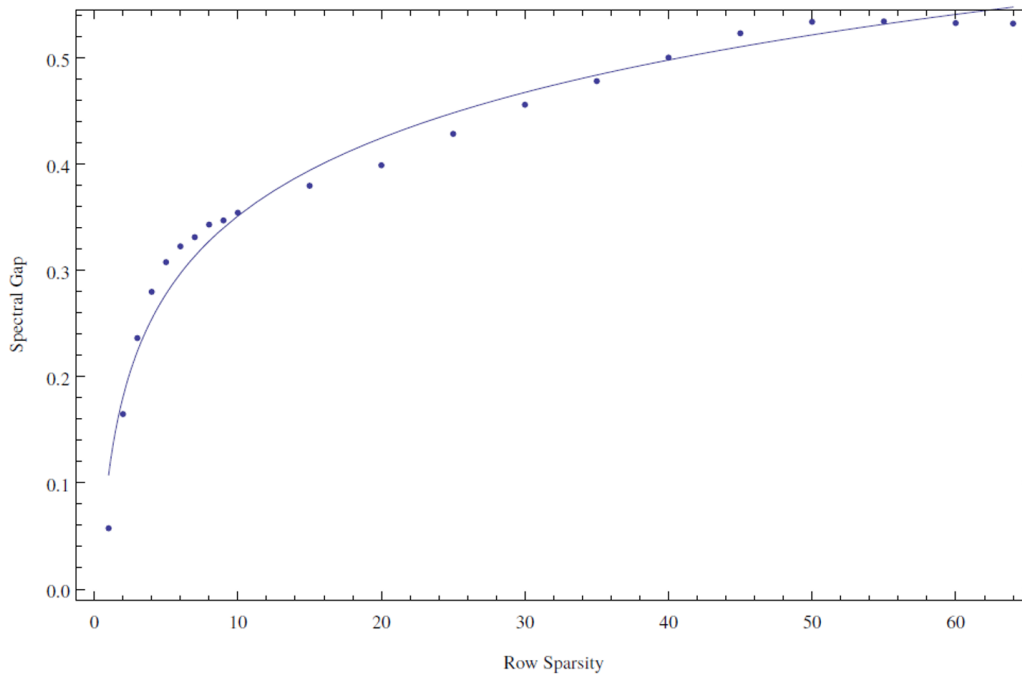
Figure 6.14: The spectral gap behavior is modeled for varying row sparsity from 1 to 2^n, where n is the number of qubits, for a fixed $n = 6$. We fit the model with the function $y(x) = a \log(bx)$ and obtain $a = 0.105908$, $\sigma_a = 0.00371134$, $b = 2.7578$, and $\sigma_b = 0.368026$. The p-values for a and b are respectively 4.59×10^{-17} and 4.37×10^{-7}.
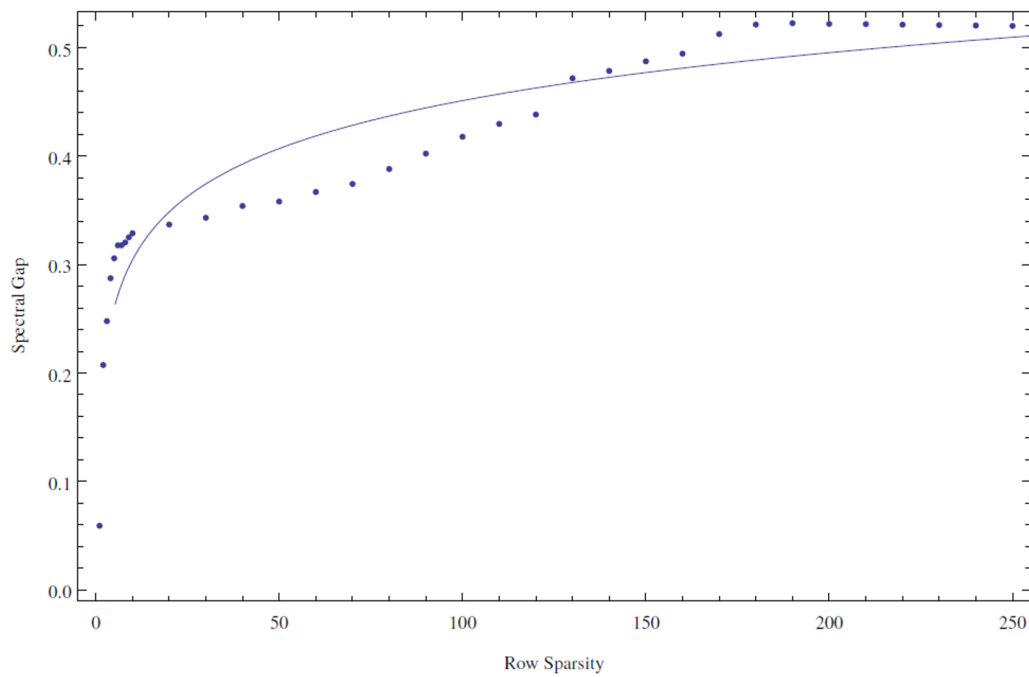
Figure 6.15: The spectral gap behavior is modeled for varying row sparsity from 1 to 2^n, where n is the number of qubits, for a fixed $n = 8$. We fit the model with the function $y(x) = a \log(bx)$ and obtain $a = 0.0637553$, $\sigma_a = 0.00385493$, $b = 11.8402$, and $\sigma_b = 4.61932$. The p-values for a and b are respectively 3.09×10^{-17} and 0.015.

References

[1] K. Temme, T. J. Osborne, K. G. Vollbrecht, D. Poulin, and F. Verstraete, "Quantum metropolis sampling", Nature **471**, 87–90 (2011).

[2] D. Aharonov and A. Ta-Shma, "Adiabatic quantum state generation and statistical zero knowledge", Proceedings of the 35th Annual ACM symposium on Theory of Computing (STOC'03), 20–29 (2003).

[3] D. W. Berry, A. M. Childs, and R. Kothari, "Hamiltonian simulation with nearly optimal dependence on all parameters", Proceedings of the 56th IEEE Symposium on Foundations of Computer Science, 792–809 (2015).

[4] C. Sanderson and R. R. Curtin, "Armadillo: a template-based c++ library for linear algebra", Journal of Open Source Software **1**, 26 (2016).

Chapter 7

QUANTUM SPEEDUPS

While the numerical data appears to indicate that the quantum Metropolis map's spectral gap [1] behaves polynomially in system size for random Hamiltonians with a fixed polynomial row sparsity, we can only speak confidently for system sizes ranging from three to ten qubits. Further data is needed for larger systems – indeed, a pressing question to answer is whether this behavior holds as the system size increases to a few hundred qubits, or the size of a small scale quantum computer. Currently we can only expand on the result of Brandão *et al.* and say that quadratic speed-ups in the parameters n and m, respectively the Hilbert space size and the number of input matrices, are possible in the worst case, and exponential speedups are possible under the specific circumstances outlined in [2] and possible for small, random Hamiltonians.

Brandão *et al.* enumerate a list of open questions to solve in regards to this semidefinite programming algorithm in [2]. One of the questions listed is improving the scaling of the parameters R and δ – even if an exponential speedup is achieved in n, it will still be beneficial to work towards an algorithm that enjoys optimal scaling in all of the relevant parameters.

We expanded upon the second question listed, which asks if there are more specific instances where a more significant speedup can be achieved in n. In the context of our thesis, we asked the questions: how does sparsity affect the behavior of the quantum Metropolis stochastic map's spectral gap in system size? Does the sparsity of a Hamiltonian affect how efficiently the Gibbs state can be prepared on a quantum computer with the quantum Metropolis algorithm? In our numerical simulations, we fixed row sparsity as a constant, which yielded nonsensical results, and then fixed row sparsity as a polynomial function of the system size, or the number of qubits. While there aren't enough data points to say definitively that the quantum Metropolis stochastic map's spectral gap is absolutely polynomial in system size, the results seem to indicate so for small Hamiltonians. They also appear to imply a relationship between the row sparsity function and the relationship between the spectral gap and the system size. We also explored whether varying the random distribution from which the nonzero elements' values came from changed the behavior, but

the changes are not significant. One question that we believe is worth exploring, asides from the spectral gap behavior for larger systems, is whether there are sparsity patterns that guarantee an efficient mixing time, and whether it's viable to classify Hamiltonians based on sparsity patterns. Another question worth exploring, if the results for the spectral gap behavior for larger system sizes are positive, is whether the spectral gap behavior in system size is related to the first order derivative of the sparsity function, or if the spectral gap behavior can be modeled with a polynomial function of order d, if the sparsity pattern is of order $d + 1$. The results from the two different quadratic row sparsity functions imply that the spectral gap behavior is predominantly dependent on the highest order term in the row sparsity function. The results for row sparsity functions of a higher polynomial order can be explored if the system size is increased.

References

[1] K. Temme, T. J. Osborne, K. G. Vollbrecht, D. Poulin, and F. Verstraete, "Quantum metropolis sampling", Nature **471**, 87–90 (2011).

[2] F. Brandão and K. Svore, "Quantum speed-ups for semidefinite programming", (2016), arXiv:1609.05537 [quant-ph].

Appendix A

PAULI OPERATOR GENERATOR

```cpp
#include <math.h>
#include <cmath>
#include <complex>
#include <armadillo>

using namespace std;
using namespace arma;

class Pauli
    {
    public:

    //
    // Constructors
    //

    Pauli(int size, int gate, int order);

    operator cx_mat() { init(); return matrix; }

    //
    // Functions
    //

    // returns the Pauli matrix corresponding to the integer.
    cx_mat
    matrix_generator(int order);

    // returns the tensor product.
    cx_mat
    tensor_product_generator(int size, int gate, int order);
```

```cpp
    private:

    ////////
    int size_,
        gate_,
        order_;
    bool initted_;
    cx_mat matrix;

    ////////

    void
    init();

    };

inline Pauli::
Pauli(int size, int gate, int order)
    :
    initted_(false)
    {
    size_ = size;
    gate_ = gate;
    order_ = order;
    }

cx_mat inline Pauli::
matrix_generator (int order)
// generates the Pauli matrix I, X, Y, or Z
// corresponding to the integer n.
    {
    cx_mat m(2, 2);
    if (order == 0)
        {
        m.at(0, 0) = cx_double(1.0, 0.);
```

```
        m. at (1 , 0) = cx_double (0. , 0.);
        m. at (0 , 1) = cx_double (0. , 0.);
        m. at (1 , 1) = cx_double (1.0 , 0.);
         }
    else if ( order == 1)
         {
        m. at (0 , 0) = cx_double (0. , 0.);
        m. at (1 , 0) = cx_double (1.0 , 0.);
        m. at (0 , 1) = cx_double (1.0 , 0.);
        m. at (1 , 1) = cx_double (0. , 0.);
         }
    else if ( order == 2)
         {
        m. at (0 , 0) = cx_double (0. , 0.);
        m. at (1 , 0) = cx_double (0. , 1.0);
        m. at (0 , 1) = cx_double (0. , −1.0);
        m. at (1 , 1) = cx_double (0. , 0.);
         }
    else if ( order == 3)
         {
        m. at (0 , 0) = cx_double (1.0 , 0.);
        m. at (1 , 0) = cx_double (0. , 0.);
        m. at (0 , 1) = cx_double (0. , 0.);
        m. at (1 , 1) = cx_double (−1.0 , 0.);
         }
    return m;
    }


cx_mat inline Pauli ::
tensor_product_generator (int size , int gate , int order)
// generates the tensor product operator corresponding
// to the number of qubits , denoted as size ,
// the qubit that the operator operates on , denoted as
// gate , and the Pauli operator chosen , denoted as order .
    {
    cx_mat m;
```

```cpp
    cx_mat identity = matrix_generator(0);
    cx_mat pauli_matrix = matrix_generator(order);
    if (gate == 1)
        {
        m = pauli_matrix;
        for (int i = 1; i < size; ++i)
            { m = kron(m, identity); }
        }
    else if (gate == size)
        {
        m = identity;
        for (int i = 1; i < size -1; ++i)
            { m = kron(m, identity); }
        m = kron(m, pauli_matrix);
        }
    else
        {
        m = identity;
        for (int i = 1; i < gate; ++i)
            { m = kron(m, identity); }
        m = kron(m, pauli_matrix);
        for (int i = gate + 1; i < size; ++i)
            { m = kron(m, identity); }
        }
    return m;
    }

void inline Pauli::
init()
    {
    if(initted_) return;
    matrix = tensor_product_generator(size_, gate_, order_);
    initted_ = true;
    }
```

Appendix B

SPECTRAL GAP ESTIMATOR

```cpp
#include <iostream>
#include <iomanip>
#include <stdio.h>
#include <complex>
#include <random>
#include <math.h>
#include <cmath>
#include <vector>
#include <algorithm>
#include <armadillo>
#include "pauli.h"

using namespace std;
using namespace arma;

// Armadillo documentation is available at:
// http://arma.sourceforge.net/docs.html

int
main(int argc, char** argv)
  {
  // Input parameters.
  int qubits = 10;
  int n = pow (2, qubits);
  int iterations = 20;
  // Rather than use the naive definition of sparsity,
  // we choose to use the definition of row-sparsity,
  // which will either be polynomially bound
  // in n or be a fixed constant.
  int sparsity = qubits;
  float beta = 0.00366300366;
```

```cpp
double sanity_check;

// Generate the Hamiltonian

// Seed the random generator.
arma_rng::set_seed_random();
// Define a vector for collecting the spectral gaps
// of the quantum Metropolis stochastic map.
vec spectral_gaps(iterations);
// Ensure that duplicate locations do not stop
// the program in its tracks.
bool add_values = true;
bool sort_locations;
bool check_for_zeros;
// Start the iterations.
for (int i = 0; i < iterations; ++i)
    {
    cout << i << endl;
    sp_mat A = sp_mat(n, n);
    // Determine the number of nonzero elements in each row.
    vec initial_row_sparsity = randi<vec>(n, \
    distr_param(0, sparsity));
    // Start building the Hamiltonian with cross shapes,
    // i.e. generating values for the j-th row
    // and the j-th column for j = 0,...n.
    for (int j = 0; j < n; ++j)
        {
        // Determine the one-dimensional indices
        // of the nonzero elements.
        vec row_col_indices = randi<vec>(initial_row_sparsity(j), \
        distr_param(0, 2*n-1));
        // Generate the values of the nonzero elements.
        vec values = randu<vec>(row_col_indices.n_elem);
        // Generate the locations matrix that contains
        // the indices in (i, j) format.
        umat locations = umat(2, row_col_indices.n_elem);
```

```
            // Convert the one−dimensional indices
            // to (i, j) format.
            for (int k = 0; k < row_col_indices.n_elem; ++k)
                {
                if (row_col_indices(k) < n)
                    {
                    locations(0, k) = j;
                    locations(1, k) = row_col_indices(k);
                    }
                else
                    {
                    locations(0, k) = row_col_indices(k) − n;
                    locations(1, k) = j;
                    }
                }
        // Generate the sparse matrix for the j−th cross.
        sp_mat B(add_values, locations, values, n, \
        n, sort_locations=true, check_for_zeros=true);
        // Add it to the main sparse matrix.
        A = A + B;
        }
// Generate the Hamiltonian by adding the sparse matrix's
// transpose to itself so that the matrix is symmetrical.
sp_mat Hamiltonian = A.t() + A;
// Convert the sparse Hamiltonian to a dense matrix.
mat denseHamiltonian(Hamiltonian);
// Make sure the row−sparsity is bound by
// the sparsity parameter.
for (int j = 0; j < n; ++j)
    {
    rowvec row = denseHamiltonian.row(j);
    // Generate vector containing the indices
    // of the nonzero elements in each row.
    uvec indices_of_nonzeros = find(row, 0);
    // Determine the number of nonzero
    // elements that need to be removed.
```

```cpp
        int removed_elements = indices_of_nonzeros.n_elem \
        - sparsity;
        // Remove elements if the row-sparsity
        // exceeds the set constraint.
        if (removed_elements > 0)
            {
            // Shuffle the indices of the nonzero elements
            // to determine the first elements to remove.
            uvec order_of_removal = \
            shuffle(indices_of_nonzeros);
            for (int k = 0; k < order_of_removal.n_elem; ++k)
                {
                // Remove the extra elements from the row.
                denseHamiltonian(j, order_of_removal(k)) = 0;
                // Remove the corresponding elements from
                // the column.
                denseHamiltonian(order_of_removal(k), j) = 0;
                }
            }
        }

// Calculate all the eigenvalues and the eigenvectors.

vec eigval;
mat eigvec;
// Compute all of the eigenvalues and eigenvectors.
eig_sym(eigval, eigvec, denseHamiltonian);
// Make the Hamiltonian complex.
mat zeros(n, n, fill::zeros);
cx_mat eigenvectors(eigvec, zeros);
cx_mat t_eigenvectors = eigenvectors.st();

// Compute the quantum Metropolis map.

cx_mat metropolis_map(n, n, fill::zeros);
for (int j = 0; j < qubits; ++j)
```

```
    {
    cx_mat pauli0 = Pauli(qubits, j+1, 0);
    cx_mat pauli_i = t_eigenvectors*pauli0*eigenvectors;
    cx_mat fin_pauli_i = pauli_i % conj(pauli_i);
    cx_mat pauli1 = Pauli(qubits, j+1, 1);
    cx_mat pauli_x = t_eigenvectors*pauli1*eigenvectors;
    cx_mat fin_pauli_x = pauli_x % conj(pauli_x);
    cx_mat pauli2 = Pauli(qubits, j+1, 2);
    cx_mat pauli_y = t_eigenvectors*pauli2*eigenvectors;
    cx_mat fin_pauli_y = pauli_y % conj(pauli_y);
    cx_mat pauli3 = Pauli(qubits, j+1, 3);
    cx_mat pauli_z = t_eigenvectors*pauli3*eigenvectors;
    cx_mat fin_pauli_z = pauli_z % conj(pauli_z);
    metropolis_map = metropolis_map + fin_pauli_i + \
    fin_pauli_x + fin_pauli_y + fin_pauli_z;
    }
vec weight1 = exp(-beta*eigval);
rowvec weight2 = exp(beta*eigval).t();
mat weights = weight1*weight2;
weights.elem( find(weights > 1.) ).ones();
weights = weights/(4.*qubits);
// Calculate and apply the Boltzmann weights.
vec gibbs_weight1 = exp(beta*eigval/2.);
rowvec gibbs_weight2 = exp(-beta*eigval/2.).t();
mat gibbs_weights = gibbs_weight1*gibbs_weight2;
mat final_weights = weights % gibbs_weights;
metropolis_map = final_weights % metropolis_map;
// Convert the quantum Metropolis map to a sparse matrix.
sp_mat sparse_metropolis(conv_to<mat>::from(metropolis_map));
// Compute the two largest eigenvalues of the map.
vec fin_eigval = eigs_sym(sparse_metropolis, 2, "la");
// Solve for the spectral gap.
double spectral_gap = fin_eigval[1] - fin_eigval[0];
sanity_check = fin_eigval[1];
spectral_gaps[i] = spectral_gap;
}
```

```
cout << mean(spectral_gaps) << endl;
cout << sanity_check << endl;
cout << stddev(spectral_gaps) << endl;
return 0;
}
```

Note that the values of the nonzero elements in the row-sparse Hamiltonians can be generated from a random uniform distribution using `randu` or a random normal distribution using `randn`. If the behavior of spectral gaps for polynomial row sparsity is of interest, the sparsity can be set to a function polynomial in the number of qubits. If the behavior of spectral gaps for constant row sparsity is of interest, then the sparsity should be a fixed integer. Keeping the number of qubits constant and varying the sparsity linearly will give plots describing the behavior of the spectral gap for varying sparsities.

Appendix C

TABLES OF SPECTRAL GAPS FOR CONSTANT ROW SPARSITIES

We must note beforehand that the program was run 1000 times for system sizes ranging from 3 to 7 qubits, 200 times for 8 qubits, 100 times for 9 qubits, and 20 times for 10 qubits. This is due to both the decreasing width of the spectral gap distribution for an increasing system size, and to the exponentially increasing run time of the program.

System Size	Mean Spectral Gap	Standard Deviation
3	0.131948	0.12906
4	0.0752339	0.0625002
5	0.0594832	0.0350691
6	0.0573202	0.0232473
7	0.0571217	0.0165054
8	0.059097	0.0120008
9	0.0591058	0.00900822
10	0.0586839	0.00457439

Table C.1: **Table of Spectral Gaps for a Constant Row Sparsity of 1.** The spectral gaps of random Hamiltonians with varying system sizes and a constant row sparsity of one are recorded in this table. The Hamiltonians were generated using a uniform distribution, and the system sizes range from three to ten qubits. The spectral gaps for each size and row sparsity form a Gaussian distribution over the iterations; the values chosen to generate the plots are the means of the distribution. The standard deviations are recorded in this table for bookkeeping purposes.

System Size	Mean Spectral Gap	Standard Deviation
3	0.28057	0.134129
4	0.235667	0.0882728
5	0.230318	0.0641031
6	0.236346	0.0425841
7	0.245281	0.0329522
8	0.2478	0.0278092
9	0.250355	0.0228699
10	0.253413	0.0200079

Table C.2: **Table of Spectral Gaps for a Constant Row Sparsity of 3 (Uniform).** The spectral gaps of random Hamiltonians with varying system sizes and a constant row sparsity of three are recorded in this table. The Hamiltonians were generated using a uniform distribution, and the system sizes range from three to ten qubits. The spectral gaps for each size and row sparsity form a Gaussian distribution over the iterations; the values chosen to generate the plots are the means of the distribution. The standard deviations are recorded in this table for bookkeeping purposes.

System Size	Mean Spectral Gap	Standard Deviation
3	0.259648	0.131798
4	0.218389	0.086442
5	0.21552	0.0598587
6	0.229392	0.0399262
7	0.238811	0.0298309
8	0.246696	0.0245005
9	0.2436	0.0220569
10	0.243649	0.015332

Table C.3: **Table of Spectral Gaps for a Constant Row Sparsity of 3 (Normal).** The spectral gaps of random Hamiltonians with varying system sizes and a constant row sparsity of three are recorded in this table. The Hamiltonians were generated using a normal distribution, and the system sizes range from three to ten qubits. The spectral gaps for each size and row sparsity form a Gaussian distribution over the iterations; the values chosen to generate the plots are the means of the distribution. The standard deviations are recorded in this table for bookkeeping purposes.

System Size	Mean Spectral Gap	Standard Deviation
3	0.430745	0.0838608
4	0.391826	0.0805519
5	0.34664	0.0692611
6	0.322693	0.0527776
7	0.319027	0.0423004
8	0.317716	0.0312893
9	0.319619	0.0249236
10	0.319361	0.0154517

Table C.4: **Table of Spectral Gaps for a Constant Row Sparsity of 6 (Uniform).** The spectral gaps of random Hamiltonians with varying system sizes and a constant row sparsity of six are recorded in this table. The Hamiltonians were generated using a uniform distribution, and the system sizes range from three to ten qubits. The spectral gaps for each size and row sparsity form a Gaussian distribution over the iterations; the values chosen to generate the plots are the means of the distribution. The standard deviations are recorded in this table for bookkeeping purposes.

System Size	Mean Spectral Gap	Standard Deviation
3	0.430213	0.0935425
4	0.368988	0.0817553
5	0.336248	0.0646967
6	0.312305	0.0497601
7	0.313092	0.0381403
8	0.303306	0.0317458
9	0.318659	0.0237178
10	0.314262	0.0243659

Table C.5: **Table of Spectral Gaps for a Constant Row Sparsity of 6 (Normal).** The spectral gaps of random Hamiltonians with varying system sizes and a constant row sparsity of six are recorded in this table. The Hamiltonians were generated using a normal distribution, and the system sizes range from three to ten qubits. The spectral gaps for each size and row sparsity form a Gaussian distribution over the iterations; the values chosen to generate the plots are the means of the distribution. The standard deviations are recorded in this table for bookkeeping purposes.

System Size	Mean Spectral Gap	Standard Deviation
3	0.450002	0.0708793
4	0.444523	0.06722
5	0.384432	0.0705576
6	0.343187	0.0573291
7	0.32892	0.0437149
8	0.320327	0.0357085
9	0.323633	0.0270114
10	0.328503	0.0158702

Table C.6: **Table of Spectral Gaps for a Constant Row Sparsity of 8.** The spectral gaps of random Hamiltonians with varying system sizes and a constant row sparsity of eight are recorded in this table. The Hamiltonians were generated using a uniform distribution, and the system sizes range from three to ten qubits. The spectral gaps for each size and row sparsity form a Gaussian distribution over the iterations; the values chosen to generate the plots are the means of the distribution. The standard deviations are recorded in this table for bookkeeping purposes.

A p p e n d i x D

TABLES OF SPECTRAL GAPS FOR POLYNOMIAL ROW SPARSITIES

Again, we make note of the fact that the program was run 1000 times for system sizes ranging from 3 to 7 qubits, 200 times for 8 qubits, 100 times for 9 qubits, and 20 times for 10 qubits. This is due to both the decreasing width of the spectral gap distribution for an increasing system size, and to the exponentially increasing run time of the program.

System Size	Mean Spectral Gap	Standard Deviation
3	0.272414	0.135028
4	0.304903	0.0898785
5	0.324757	0.0660137
6	0.320201	0.0513793
7	0.324599	0.0434523
8	0.321034	0.0337077
9	0.323744	0.0273422
10	0.318483	0.021793

Table D.1: **Table of Spectral Gaps for a Linear Row Sparsity (Uniform).** The spectral gaps of random Hamiltonians with varying system sizes and a linear row sparsity of $s = n$, where n is the system size, are recorded in this table. The Hamiltonians were generated using a uniform distribution, and the system sizes range from three to ten qubits. The spectral gaps for each size and row sparsity form a Gaussian distribution over the iterations; the values chosen to generate the plots are the means of the distribution. The standard deviations are recorded in this table for bookkeeping purposes.

System Size	Mean Spectral Gap	Standard Deviation
3	0.266152	0.134391
4	0.288471	0.0867833
5	0.308333	0.0635762
6	0.31281	0.0490572
7	0.317535	0.0412417
8	0.320002	0.0346115
9	0.32223	0.0226928
10	0.322739	0.0236867

Table D.2: **Table of Spectral Gaps for a Linear Row Sparsity (Normal).** The spectral gaps of random Hamiltonians with varying system sizes and a linear row sparsity of $s = n$ are recorded in this table. The Hamiltonians were generated using a normal distribution, and the system sizes range from three to ten qubits. The spectral gaps for each size and row sparsity form a Gaussian distribution over the iterations; the values chosen to generate the plots are the means of the distribution. The standard deviations are recorded in this table for bookkeeping purposes.

System Size	Mean Spectral Gap	Standard Deviation
3	0.399488	0.0951201
4	0.508677	0.0385238
5	0.509545	0.0396221
6	0.465924	0.0466627
7	0.408253	0.0535563
8	0.368671	0.0446162
9	0.338309	0.029967
10	0.32759	0.0230782

Table D.3: **Table of Spectral Gaps for a Quadratic Row Sparsity with No Linear Term.** The spectral gaps of random Hamiltonians with varying system sizes and a quadratic row sparsity of $s = n^2 - 4$ are recorded in this table. The Hamiltonians were generated using a normal distribution, and the system sizes range from three to ten qubits. The spectral gaps for each size and row sparsity form a Gaussian distribution over the iterations; the values chosen to generate the plots are the means of the distribution. The standard deviations are recorded in this table for bookkeeping purposes.

System Size	Mean Spectral Gap	Standard Deviation
3	0.448036	0.0734268
4	0.513912	0.0343196
5	0.508291	0.0408527
6	0.46293	0.0483015
7	0.403323	0.0536587
8	0.362505	0.0416005
9	0.340279	0.0320261
10	0.329493	0.0182348

Table D.4: **Table of Spectral Gaps for a Quadratic Row Sparsity (Uniform).** The spectral gaps of random Hamiltonians with varying system sizes and a quadratic row sparsity of $s = n^2 - n + 1$ are recorded in this table. The Hamiltonians were generated using a uniform distribution, and the system sizes range from three to ten qubits. The spectral gaps for each size and row sparsity form a Gaussian distribution over the iterations; the values chosen to generate the plots are the means of the distribution. The standard deviations are recorded in this table for bookkeeping purposes.

System Size	Mean Spectral Gap	Standard Deviation
3	0.449535	0.0861218
4	0.530828	0.047366
5	0.529576	0.0661663
6	0.461994	0.0482608
7	0.396148	0.0523911
8	0.360419	0.0443507
9	0.335	0.0314762
10	0.313142	0.019848

Table D.5: **Table of Spectral Gaps for a Quadratic Row Sparsity (Normal).** The spectral gaps of random Hamiltonians with varying system sizes and a quadratic row sparsity of $s = n^2 - n + 1$ are recorded in this table. The Hamiltonians were generated using a normal distribution, and the system sizes range from three to ten qubits. The spectral gaps for each size and row sparsity form a Gaussian distribution over the iterations; the values chosen to generate the plots are the means of the distribution. The standard deviations are recorded in this table for bookkeeping purposes.

System Size	Mean Spectral Gap	Standard Deviation
4	0.483848	0.0514376
5	0.539165	0.0132614
6	0.532461	0.00739548
7	0.527287	0.00453272
8	0.521715	0.00613425
9	0.470675	0.0236658
10	0.395194	0.0335142

Table D.6: **Table of Spectral Gaps for a Cubic Row Sparsity.** The spectral gaps of random Hamiltonians with varying system sizes and a cubic row sparsity of $s = \lfloor 0.5n^3 - n^2 - n - 2 \rfloor$ are recorded in this table. The Hamiltonians were generated using a uniform distribution, and the system sizes range from four to ten qubits. The spectral gaps for each size and row sparsity form a Gaussian distribution over the iterations; the values chosen to generate the plots are the means of the distribution. The standard deviations are recorded in this table for bookkeeping purposes.

Appendix E

TABLES OF SPECTRAL GAPS FOR VARYING SPARSITIES

We note beforehand that the program was run 1000 times for every row sparsity value when the system size was fixed at six qubits. The program was run 200 times for every row sparsity value when the system size was fixed at eight qubits.

System Size	Mean Spectral Gap	Standard Deviation
1	0.0573202	0.0232473
2	0.164776	0.047922
3	0.236346	0.0425841
4	0.279917	0.0452841
5	0.307812	0.0514049
6	0.322693	0.0527776
7	0.33135	0.0540615
8	0.343187	0.0573291
9	0.347102	0.0584836
10	0.354374	0.0592948
15	0.379749	0.0619381
20	0.399062	0.0636982
25	0.428592	0.0596389
30	0.456042	0.0524913
35	0.478275	0.0425512
40	0.500415	0.0317799
45	0.52329	0.0235721
50	0.534084	0.0120081
55	0.534482	0.00770926
60	0.532918	0.00730745
64	0.532431	0.00725479

Table E.1: **Table of Spectral Gaps for Varying Sparsity at 6 Qubits.** The spectral gaps of random Hamiltonians with varying row sparsities for a fixed system size of six qubits are recorded in this table. The Hamiltonians were generated using a uniform distribution. The spectral gaps for each size and row sparsity form a Gaussian distribution over the iterations; the values chosen to generate the plots are the means of the distribution. The standard deviations are recorded in this table for bookkeeping purposes.

System Size	Mean Spectral Gap	Standard Deviation
1	0.059097	0.0120008
2	0.207349	0.0244815
3	0.2478	0.0278092
4	0.287381	0.0312335
5	0.30578	0.0317456
6	0.317716	0.0312893
7	0.317794	0.0350901
8	0.320327	0.0357085
9	0.325122	0.0359037
10	0.328836	0.0370275
20	0.336872	0.0411096
30	0.343144	0.0395401
40	0.354005	0.0418522
50	0.358094	0.0411936
60	0.366935	0.0466386
70	0.374319	0.0441991
80	0.388093	0.0477802
90	0.402302	0.0447742
100	0.417807	0.0454497
110	0.429714	0.0443658
120	0.438333	0.109476
130	0.471791	0.0306441
140	0.478555	0.0267372
150	0.487398	0.0179878
160	0.494481	0.0169873
170	0.512532	0.0172845
180	0.521308	0.00888657
190	0.522598	0.00283208
200	0.52192	0.00295168
210	0.52172	0.00304521
220	0.521245	0.00307243
230	0.520854	0.00299041
240	0.520439	0.0031063
250	0.52004	0.00278052

Table E.2: **Table of Spectral Gaps for Varying Sparsity at 8 Qubits.** The spectral gaps of random Hamiltonians with varying row sparsities for a fixed system size of eight qubits are recorded in this table. The Hamiltonians were generated using a uniform distribution. The spectral gaps for each size and row sparsity form a Gaussian distribution over the iterations; the values chosen to generate the plots are the means of the distribution. The standard deviations are recorded in this table for bookkeeping purposes.

INDEX