Diagonalization Techniques for Sparse Matrices

Rowan W. Hale *

May 17, 2012

Abstract

We discuss several diagonalization techniques that fall into categories of exact or iterative and direct or stochastic. Our discussion of these techniques has an emphasis on the runtime and memory usage (and accuracy where applicable) of the MATLAB code that we used to implement them, as well as the feasbility of these values. We also discuss a technique for generating random instances of Constraint Satisfaction Problems that we used to create interesting problems to work with.

* *rowanh@cs.washington.edu*. Department of Computer Science, University of Washington.

1 Introduction

When working with matrices that model systems, it is often useful to find the eigenvalues and eigenvectors of these matrices, because the eigenvalues and eigenvectors can correpsond to important information about the system. The process of finding these eigenvalues and eigenvectors, called diagonalization, comes in several forms: diagonalization can be exact, where the answer is found exactly, or it can be iterative, where the answer is approached as the number of iterations goes to infinity. In addition, these diagonalization calculations can be done either directly or stochastically, and in this paper we examine diagonalization techniques that fall into several of these classifications.We also discuss a technique for generating random instances of problems with which to work. All of the discussed algorithms are implemented using MATLAB code.

2 Exact Diagonalization of Sparse Matrices

2.1 Limitations

In almost all of the systems we are interested in modeling, such as quantum adiabatic optimization Hamiltonians and random walk transition matrices, the most frequent calculation was finding the eigenvalues of large sparse matrices. MATLAB has a built in exact diagonalization function for sparse matrices, *eigs()*, that was able to do these calculations, but because these problems are modelling systems of n qubits, the Hamiltonians grow like $O(2^n)$, which introduces the problem of memory consumption; memory is finite and even doubling or quadrupling the memory efficiency of an algorithm will only allow the computations to work for slightly higher values of n. Using *eigs()*, we were able to successfully model these problems for up to 20 qubits before MATLAB ran out of physical memory with which to do its exact diagonalization.

2.2 Runtime Analysis

When we ran out of memory, it was important for us to ask what our limiting resource was; it would seem that the answer is memory, but if this process of exact diagonalization is producing runtimes that are as poor as the memory consumption is, then perhaps there is an underlying issue that needs to be accounted for. For adiabatic Hamiltonians of the form

$$H = (1 - s)H_0 + sH_f \tag{1}$$

where

$$H_0 = -\sum_{i=1}^{n} \sigma_x^i \tag{2}$$

and H_f is the problem Hamiltonian of choice, the following plot shows the runtimes of the construction and exact diagonalization of numerous Hamiltonians H for $n \in \{1, 2, \ldots, 19\}$.

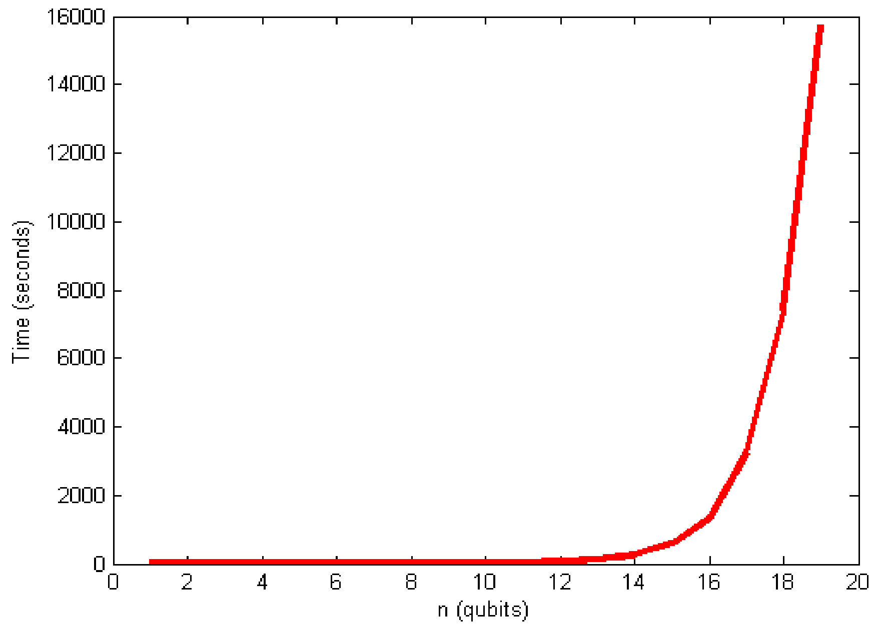

Figure 1: Runtimes of the construction and exact diagonalization of numerous Hamiltonians H, as defined in (1).

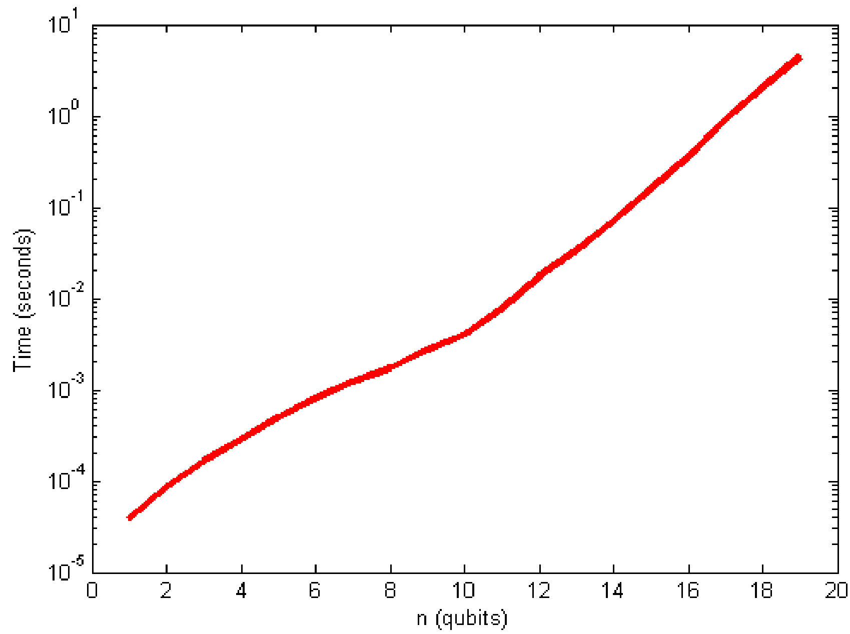

Figure 2: Logarithmic plot of the runtimes from figure 2.

It is clear from figures 1 and 2 that the runtimes are scaling exponentially in the number of qubits, and even for very small values of n, the times are infeasible. Upon examination of the runtimes of just *eig()* on matrices of increasing size, we can see that the runtime of *eig()* is scaling just as poorly.

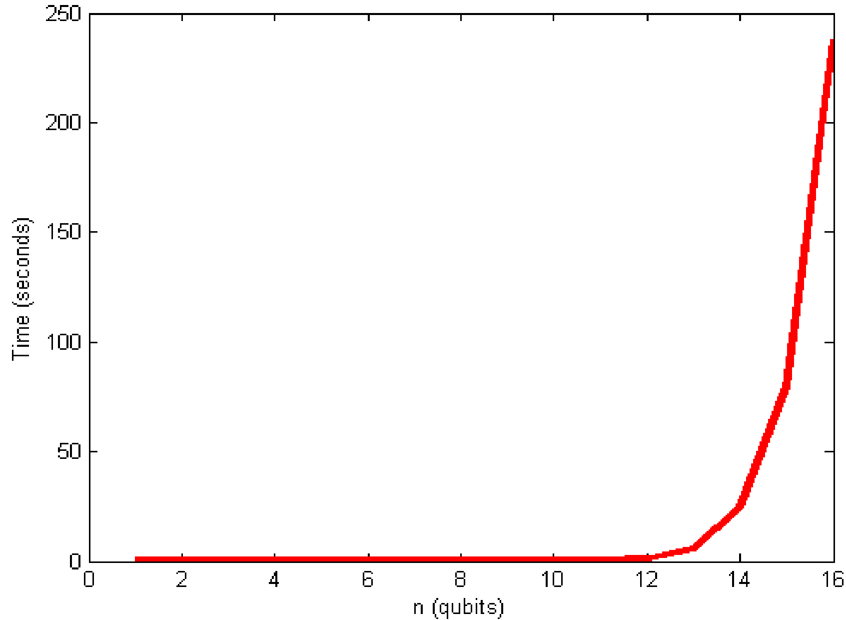

Figure 3: Runtime of *eigs()* on matrices of size $2^n \times 2^n$.

These results shows us that both the memory consumption and runtime of exact diagonalization are scaling out of control when performed on matrices of exponentially increasing size. As such, we must find a new method of producing eigenvalues that is able to overcome either the runtime or memory consumption scaling.

3 Conjugate Gradient

3.1 Explanation and Benefits

When we were generating eigenvalues using *eigs()*, in almost all cases we were only interested in a few of the eigenvalues of the system, particularly the highest or lowest. Thus, it was suggested that we try generating eigenvalues using an algorithm called Conjugate Gradient, which has similar properties to those of *eigs()*, to see if it provided better runtimes or lower memory consumption. Conjugate Gradient is an iterative algorithm that offers a few levels of freedom:

1. It produces only 1 highest or lowest eigenvalue and eigenvector of the system, and successive runs with different initial guesses yield the successive lowest or highest eigenvalue. This allows us to choose how many eigenvalues we want to generate.

2. It is an iterative algorithm that approaches the exact solution, so we are able to choose a desired accuracy with which we produce results.

While these probably won't affect the memory consumption used in finding eigenvalues, they could be tweaked in an attempt to control the runtime of the algorithm.

3.2 Runtime Analysis

While the levels of freedom in using an iterative eigenvalue solver sound like they could have produced speedups, the runtime of our implementation turned out to be much worse than the runtime of *eigs()*. For small matrices, the runtime of both *eigs()* and our conjugate gradient implementation were fairly similar, but as n increased, our implementation took longer to produce a single eigenvalue than MATLAB took to produce a vector of all of the eigenvalues of the system in question. At this point it was obvious that we were best off just using *eigs()*, since our Conjugate Gradient implementation was providing such slow runtimes.

4 Quantum Monte Carlo Algorithms

4.1 Explanation and Benefits

The original difficulty with exact diagonalization discussed in section 2 was that both the memory consumption and runtime were scaling out of control, because the matrices on which exact diagonalization was being performed grew in $O(2^n)$. This is where Quantum Monte Carlo algorithms (QMC) become useful. While exact diagonalization finds eigenvalues by doing operations directly on matrices, QMC performs these operations stochastically by manipulating bit strings. The benefit of using an algorithm such as the one defined in [1] is that the algorithm is focused on manipulating bit strings, which consume very little memory, and most importantly these bit strings do not grow exponentially with n—in fact, they grow linearly. It should be noted that the QMC algorithm we used produces only the lowest eigenvalue, but for our purposes this was sufficient.

4.2 Accuracy and Runtime

While the memory cost of QMC is significantly better than exact diagonalization, there is a tradeoff. Because QMC is not a direct eigenvalue solver, the results that are obtained are based on factors that can be varied such as number of QMC steps and length of the path of bit strings that are being manipulated. As these factors are increased, so are both the runtime and accuracy of results. Figure 4 shows a comparison of exact diagonalzation and QMC when finding the lowest eigenvalue E of a Hamiltonian H

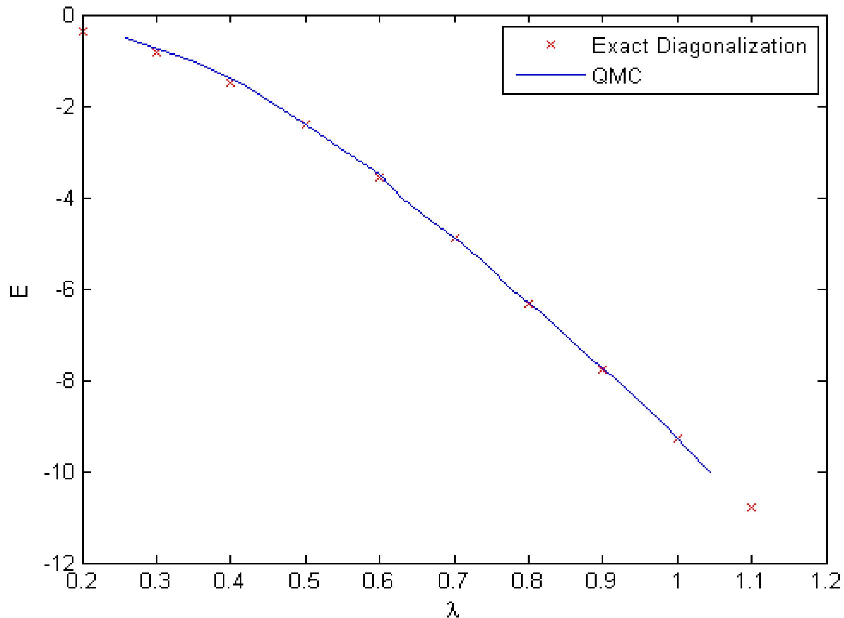

Figure 4: Comparison of exact diagonalization and QMC when finding the lowest eigenvalue of H, as defined in (3).

$$H = H_0 + \lambda H_f \tag{3}$$

where H_0 is defined in (2) and H_f is the problem Hamiltonian of choice. From figure 4 we can see that QMC provides an accurate way of finding the lowest eigenvalue of a system. The low memory consumption of QMC has already been established with the fact that all calculations are done on bitstrings, but the question still remains: how does the runtime scale? Figure 5 shows the runtime of QMC for $n \in \{2, 3, \ldots, 16\}$ with a fixed number of QMC steps and bitstring length, and the results are clear: the runtime of QMC isn't dominated by n. In practice this does not mean that the runtime for increasing n will stay the same, because for increasing n the number of QMC steps will need to be increased in order for the stochastic process to fully mix.

5 Generating Random Instances of CSPs

5.1 Naive Approach

While we have discussed many ways to find the eigenvalues of a system, the obvious prerequisite for doing so is finding a system to work with. One difficulty we were faced with was finding a random instance of a Constraint

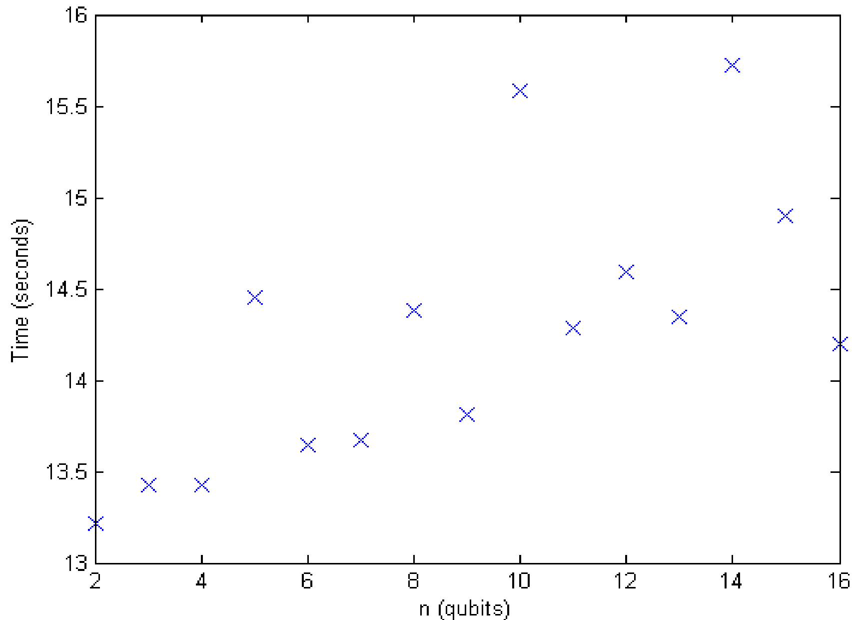

Figure 5: Runtime of QMC for a fixed number of QMC steps and varying n.

Satisfaction Problem to work with that met a certain criterion, which I will discuss shortly. The main CSPs that we worked with were 2-SAT, 3-SAT, and Exact Cover-3, and it was necessary to generate random instances of these problems as well as Hamiltonians that correspond to these problem instances. The naive process isnt particularly difficult, and can be summarized as follows:

1. Given input parameter n, generate some number of clauses $(x_a, x_b, \ldots, x_k)$ for the given problem, with $a, b, \ldots, k \leq n$.

2. Convert the clauses to matrix form. This can be done by generating a Hamiltonian for each of the clauses and summing them, or going down the diagonal of an initially empty Hamiltonian and setting each entry based on the generated clauses. This step can be skipped if only the clauses are desired, rather than the resulting Hamiltonian.

While this process makes it easy to generate random instances of the problems we were looking at, it became more important to look at the interesting cases of these problems–specifically the case in which the problem has a unique satisfying assignment, or USA (i.e. the diagonal of the problem Hamiltonian has a single zero entry). This adds a third step to our generation algorithm, assuming that only 1 clause was generated in step 1:

3. Count the number of solutions for the generated clauses, and if this number is not 1, go to step 1.

As you can imagine, using this algorithm has a potentially huge overhead, because each Hamiltonian with a number of solutions other than 1 is immediately discarded, and the number of solutions of each generated Hamiltonian is random. This excess overhead resulted in the runtimes ramping up much faster than we had hoped, and so it was necessary to look into another method of generating Hamiltonians with a USA: planted solutions.

5.2 Planted Solutions

The naive approach to generating problem Hamiltonians is essentially to generate a random set of clauses and check that they have a USA. Using planted solutions reverses this process by first imposing the requirement of having a USA, then generating clauses and checking that they correspond to this USA. The new process for generating a problem Hamiltonian is the following:

1. Given an input parameter n, choose an assignment of terms $x_1, x_2, \ldots, x_n$. This is the USA.

2. Generate a clause $(x_a, x_b, \ldots, x_k)$ for the given problem, with $a, b, \ldots, k \leq n$. If the assignment of terms in the USA causes this clause to be false, discard the clause and begin step 2 again.

3. Add the new clause to the list of generated clauses and check the number of satisfying assignments for these clauses. If this number is 0, remove the most recent clause and go back to step 2. If this number is greater than 1, go back to step 2. Otherwise, we have a set of clauses whose USA is the USA that we chose in step 1, so the algorithm is complete.

We know that checking the number of satisfying assignments for a set of clauses is an $O(2^n)$ operation, so in no way is this algorithm "fast". The runtime of the algorithm is proportional to the number of times that this operation is invoked, and that is where the speedup over the naive algorithm comes into play. By using the planted solutions algorithm, we were able to generate random instances of CSPs for up to $n = 20$ in the time it took the naive algorithm for $n = 16$, showing that while the planted solutions algorithm does not change the exponential scaling of the problem, it does provide a significant speedup.

6 Conclusion

In this paper we have discussed several different diagonalization techniques, and their associated runtimes and memory usages. With matrices growing in $O(2^n)$ and the time it takes to process such a matrix growing with the size, we have shown that it is infeasbile to use exact diagonalization or Conjugate Gradient, because they both use matrices and thus suffer from the $O(2^n)$ space and time complexity. This is the reason that the QMC algorithm is favorable, because it does not have $O(2^n)$ space or time complexity, but rather scales linearly with the number of QMC steps that we specify, and this makes QMC feasible to use for problems that grow exponentially with n, like 2-SAT, 3-SAT, and Exact Cover-3. We have shown that the naive algorithm is needlessly wasteful when generating random instances of CSPs, and we have given a solution to this waste, in the form of planted solutions.

Acknowledgements

I would like to thank Isaac Crosson and Professor Aram Harrow of the University of Washington for their enormous help in my time working with them, be that in the form of explanations of algorithms or the discussion of results and techniques.

References

[1] Edward Farhi, Jeffrey Goldstone, David Gosset, Harvey B. Meyer. A Quantum Monte Carlo Method at Fixed Energy, 2009. `http://arxiv.org/abs/0912.4271`