4G03 Assignment 3

Shayaan Siddiqui
400247500
siddis41

December 18,2021

Abstract

This third assignment considers symmetric matrices and uses methods to approximate reasonable eigenvalues. Two models will be looked at, the first being evaluating the energy values of a particle confined within a potential well, also known as a quantum potential well problem. The second concept studied is evaluating the lowest energy states of the 1/2 Heisenberg Spin Chain model. The two methods used to evaluate the symmetric matrix are known as the Jacobi's Diagonalization and Lanczos. The Hamiltonian of any model can be described using a matrix. By changing the size of the initial and transformed matrix, the convergence of eigenvalues are further explored by plotting and comparing the data.

1 Introduction

The quantum potential well is a familiar model where a particle is confined within a barrier where the probability of the particle cannot exceed the boundaries of the "box". The energy of the particle within this boundary is always non-zero. This concept can be related to Heisenberg's uncertainty principle. As such the Hamiltonian of the particle can describe the total energy of the system. The Hamiltonian as a Hermitian operator. By applying this operator to the wave function, the observable measured is the energy eigenvalues. Each of these eigenvalues correspond to an eigenvector, known as the state of the systems. This can be used in conjunction to interpret the system. To provide accurate eigenvalues, the Hamiltonian matrix has to be sufficiently large enough to extract the data of interest. As the size of the Hamiltonian approaches infinity, the eigenvalues become more exact. To find the eigenvalues of a matrix, the general diagonalization method would be applied but this is extremely difficult with larger matrices. Thus as approximation known as Jacobi's diagonalization is applied to remove the off diagonal elements. Through a series of matrix multiplications, the eigenvalues will be produced on the diagonal of the final matrix and the product of the matrix operations preformed will equate to the eigenvectors of the system.

The next model that is studied is the 1/2 Heisenberg Spin Chain. A particle is known to consist of either a +1/2 or -1/2 spin. When there is a series of particles interacting with eachother, the spin associated to each particle effect the total energy of the system by interacting with the adjacent particle's spin. Since each particle has 2 potential states, the size of the matrix is known as 2^L where L is the amount of particles within the system. With this formula, it is trivial to assess the size of the matrix becomes exceedingly large. Evaluating these matrices using Jacobian's can bring rise to instability within the method. To compensate, Lanzcos method is implemented. To evaluate, Lanzcos method uses the modified Gram-Schmidt process to minimize the amount of error. This method makes a tridiagonal matrix with smaller dimensions and extracts values of importance to approximate reasonable eigenvalues.

1.1 Quantum Potential Well

The quantum potential well is described using a wave function that has boundaries. By applying the Hamiltonian operator, the energy eigenvalue and eigenstates are extracted. To produce the Hamiltonian operator's matrix, a few formulas have to implemented. The following formulas will describe the

relationship pertaining to a particle confined within a box.

$$H|\psi\rangle = E|\psi\rangle, \quad H = H_0 + V \quad H_0 = -\frac{\hbar^2}{2m}\frac{d^2}{dx^2} \tag{1}$$

This initial equation describes how the Hamiltonian operator is attained and how it acts on the wave function. Since the Hamiltonian is a function of the potential, this value needs to be used.

$$V(x) = \frac{\hbar^2}{2m}\left[\frac{0.05}{sin^2(x)} + \frac{5}{cos^2(x)}\right] \tag{2}$$

The wave function the operator is acting on also has to be described within the boundaries of the box. The following formula shows how the particle behaves in the potential well.

$$\psi_n(x) = \begin{cases} \sqrt{\frac{2}{L}}\sin\left(\frac{n\pi x}{L}\right), 0 < x < L \\ 0 \text{ elsewhere.} \end{cases} \tag{3}$$

The energy of the function can be described with the following formula where n is an integer that represents each respective eigenvalue

$$E_n^0 = \frac{n^2\pi^2\hbar^2}{2mL^2} \tag{4}$$

The inner product of the wave function can be preformed to evaluate at each n and m if the respective wave functions are orthogonal.

$$\frac{2}{L}\int_0^L \sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi x}{L}\right)dx = \delta_{n,m} \tag{5}$$

The equation 5 above is useful as the following formulas 6, 7 show that the Hamiltonian has nxm component which can be described by the delta function. The delta function outputs a 1 when $n = m$ and 0 otherwise.
The initial formula can be rewritten in the following form and the components of the Hamiltonian can be described using the following summation.

$$\sum_{m=1}^{\infty} c_m\left(H_0 + V\right)|\psi_m\rangle = E\sum_{m=1}^{\infty} c_m|\psi_m\rangle \qquad \sum_{m=1}^{\infty} H_{nm}c_m = Ec_n \tag{6}$$

By using linear algebra relationships, each component of the Hamiltonian matrix can be solved for.

$$H_{nm} = \langle\psi_n|\left(H_0 + V\right)|\psi_m\rangle = \delta_{nm}E_n^0 + \langle\psi_n|V|\psi_m\rangle \tag{7}$$

The linear algebra format of the equation is presented as the following when the size of the Hamiltonian matrix equals to 5.

$$\begin{pmatrix} H_{11} & H_{12} & H_{13} & H_{14} & H_{15} \\ H_{21} & H_{22} & H_{23} & H_{24} & H_{25} \\ H_{31} & H_{32} & H_{33} & H_{34} & H_{35} \\ H_{41} & H_{42} & H_{43} & H_{44} & H_{45} \\ H_{51} & H_{52} & H_{53} & H_{54} & H_{55} \end{pmatrix}\begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{pmatrix} = E\begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{pmatrix}. \tag{8}$$

The following equations below will assist solving $\langle\psi_n|V|\psi_m\rangle$.

$$\int_0^{\pi/2}\frac{\sin(2nx)\sin(2mx)}{\sin^2(x)}dx = \pi\min(n,m) \tag{9}$$

$$\int_0^{\pi/2}\frac{\sin(2nx)\sin(2mx)}{\cos^2(x)}dx = (-1)^{|m-n|}\pi\min(n,m) \tag{10}$$

The following formula is the simplified method of solving for what $\langle\psi_n|V|\psi_m\rangle$ is.

$$\langle\psi_n|V|\psi_m\rangle = \frac{2\hbar^2}{m}\min(n,m)((-1)^{|m-n|}5+0.05) \tag{11}$$

Since Equation 7 uses the delta function, note that the energy values will only be present on the diagonal as the delta function returns 0 unless both n and m are the same integer. The following image shows the handwritten work preformed to solve for the Hamiltonian matrix.

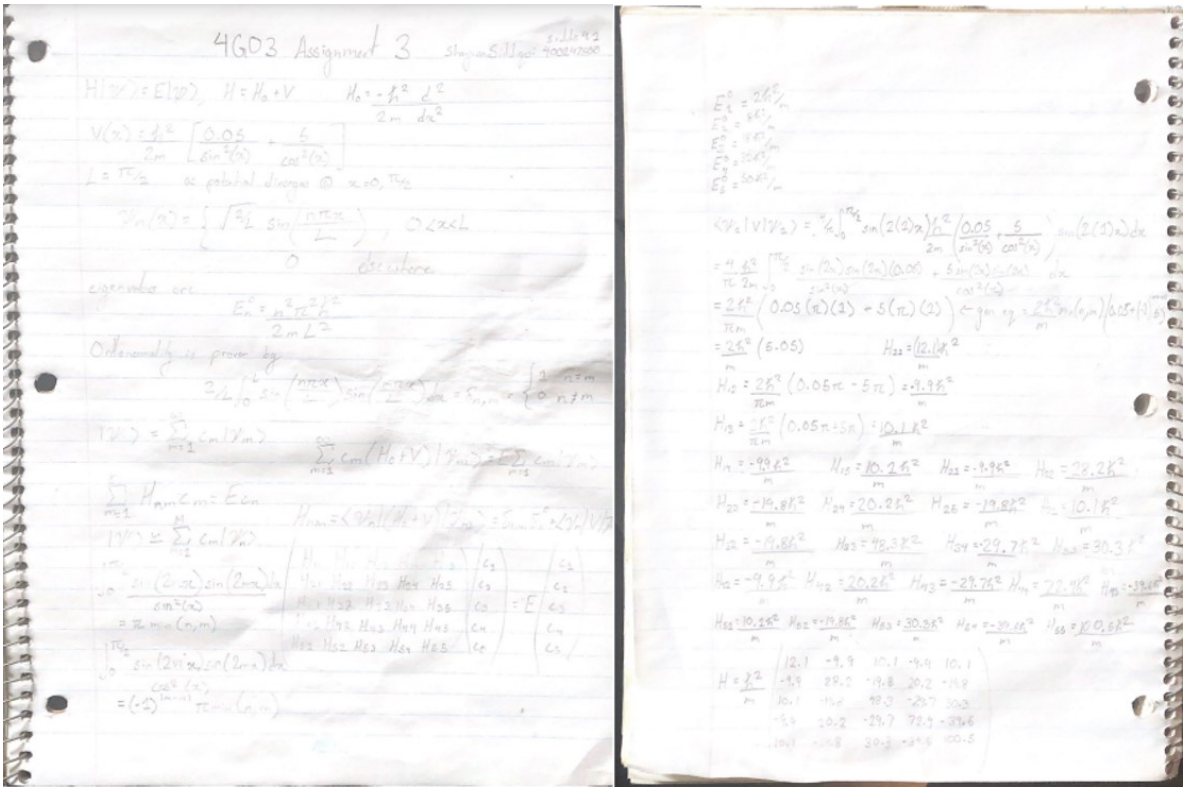

Figure 1: Hamiltonian 5x5 Matrix Production

In Figure 1, the elements of the Hamiltonian matrix are in units of $\frac{\hbar^2}{m}$. Diagonalizing a matrix of this size can become extremely difficult using conventional techniques, therefor a method known as Jacobi's Diagonalization is preformed to produce a diagonal matrix. There are various ways to implement this but the formulas are all consistent with eachother such that the evaluation of the final matrix is always the same The C++ implementation shown later will use the following steps.

Jacobi Diagonalization

1. Obtain a symmetric matrix and find the off diagonal element with the highest value
2. Find sum of squares of off diagonal elements
3. Obtain the angle using this element found in 1 and build rotation matrix in respect to angle
4. Sum of squares changes in respect to twice the element squared.
5. Find transpose of rotation matrix
6. Multiply original matrix by rotation matrix and transpose rotation matrix
7. Repeat all steps until sum of squares is sufficiently close to 0

Now that the basic implementation of Jacobi's diagonalization is understood, each step will be explained in depth.
The first step is trivial and does not require the max off diagonal element but using it will cause convergence quicker. Another method, using sweeps to go through each off diagonal element in the

upper triangle of the matrix is also reasonable but requires many more iterations. The second step is describing the rotation angle of the rotation matrix that is to be applied onto the initial matrix. To solve for the angle the following formulas can be applied.

$$\phi = cot(2\theta) = \frac{c^2 - s^2}{2cs} = \frac{a_{qq} a_{pp}}{2a_{pq}} \tag{12}$$

$$t = \tan\theta = \phi \pm \sqrt{\phi^2 + 1} = \frac{\text{sgn}(\phi)}{\phi + \sqrt{\phi^2 + 1}} \tag{13}$$

Using the above relationships, theta can be solved for by solving t. It is not necessary to solve for theta as $c = \frac{1}{\sqrt{1+t^2}}$ and $s = ct$ where c and s are the respective trigonometric functions. The general Rotation matrix is seen in the figure below.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 2: Three dimensional rotation matrix

To build a higher dimension rotation matrix, all diagonal elements are 1 except a_{qq}, a_{pp} where they are filled with c. All off diagonal elements are 0 except a_{qp}, a_{pq} where they equal to $\pm s$.

$$S = \sum_{r \neq s} |a_{rs}|^2 \tag{14}$$

$$S' = S - 2\,|a_{pq}|^2 \tag{15}$$

Equations 14 and 16 are the sum of the squares. This approaches to a value of 0 as the off diagonal elements are removed by preforming each similarity transformation. Now that the construction of the rotation matrix is defined, the application of this matrix will be explored. The rotation matrix in the following equations will be defined as J and the matrix that it will act on is known as H.

$$H_n = J_n^T ..J_2^T J_1^T H J_1 J_2 .. J_n \tag{16}$$

Since this is an orthogonal matrix, the similarity transformation can be expressed using transpose matrices rather than inverse matrix as they are the same thing. When $S'_n \approx 0$, H_n will be a diagonal matrix and the diagonal elements will be the eigenvalues of the system. The eigenvectors is obtained by the product of all rotation matrices applied and each column is a unique eigenstate of the system. Note that the eigenvalues will not be ordered and randomly arranged. By applying the rotation matrix, only the p and q elements of the matrix will be effected and with enough iterations, produce a diagonal matrix

The C++ implementation used built functions to obtain the max off diagonal element, matrix multiplication, transposing a matrix and building a rotation matrix. The main function holds the bulk of the algorithm applying these functions that were built. The following figure shows how this looks.

```
while (abs(sum) > 0.0000001) {//resolution based on offdiagonal elements

    auto IN = maxx(A);//finds coordinates
    double p = IN.first, q = IN.second;
    sum -= 2 * abs(A[p][q]) * abs(A[p][q]);

    double phi = (A[q][q] - A[p][p]) / (2 * A[p][q]);

    double t = phi + sqrt(phi * phi + 1);//obtatins angle
    double theta = atan(t);

    Matrix J = (Rotation(p, q, A.size(), theta));//rotation matrix

    J1 = Mult(J1, J);//eigen vectors

    A = Mult(A, J);
    A = Mult(Transpose(J), A);//applies rotation



}
```

Figure 3: Jacobi's Diagonalization Algorithm

Note the code also has an algorithm to build the Hamiltonian matrix of any size. Thus the code output should confirm the elements found in Figure 1 and produce a diagonal matrix after applying the algorithm.

```
Hamiltonian Matrix M=5:

6.05 -4.95 5.05 -4.95 5.05
-4.95 14.1 -9.9 10.1 -9.9
5.05 -9.9 24.15 -14.85 15.15
-4.95 10.1 -14.85 36.2 -19.8
5.05 -9.9 15.15 -19.8 50.25
M of matrix: 5

eigen vectors:

-0.102962 0.180673 0.129048 0.325237 -0.913413
0.221771 -0.359377 -0.258781 -0.767974 -0.406096
-0.330702 0.656964 0.394485 -0.550166 0.0270622
0.556549 0.635466 -0.533555 0.04183 0.00247246
0.721876 -0.0527877 0.689985 -0.00196744 0.00496831


eigen values:



24.2825, 15.3731, 78.8803, 8.52858, 3.68558,

3 lowest eigen values:

3.68558
8.52858
15.3731


Sum of Squares:2.52155e-08
```

```
M of matrix: 5

24.2825 0 0 0 0
0 15.3731 0 0 0
0 0 78.8803 0 0
0 0 0 8.52858 0
0 0 0 0 3.68558
```

Figure 4: Hamiltonian matrix of size 5

The elements are in agreement with Figure 1 although is units of $\frac{\hbar^2}{2m}$. This does indeed produce a matrix that is purely diagonal where the off diagonal elements are 0.The eigenstates are also shown but it is likely that the accuracy of it is not very robust. Now a for loop is implemented to run this algorithm multiple times to vary the dimensions of the Hamiltonian matrix to explore how the eigenvalues change.

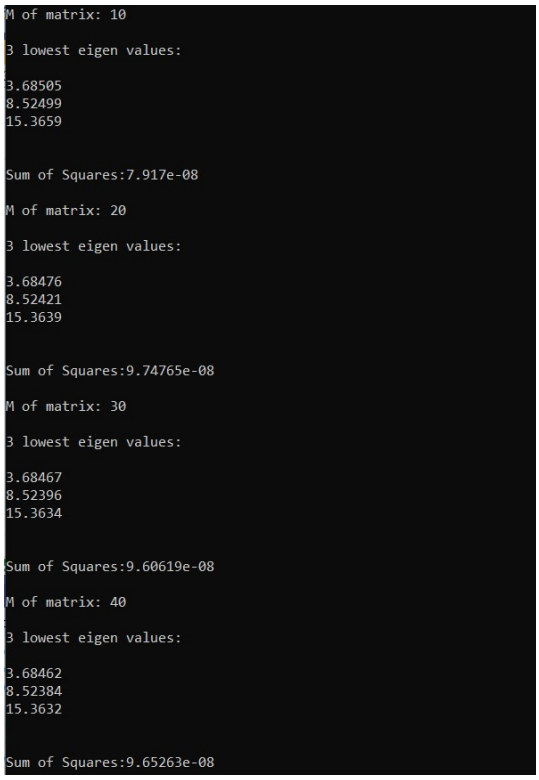

Figure 5: Eigenvalues changing M

The precision of the method did increase although the condition of the for loop to maintain Jacobi's method is that the sum is greater than 0.0000001. The majority of the outputs shown in Figure 5 show an accuracy of order $1e^{-8}$. Thus the values have been accurately determined to 8 digits.

2 Lanczos method on 1/2 Heisenberg Spin Chains

Lanczos method is used when the matrix size is extremely large. The purpose of Lanczos method is to greatly reduce the size of this matrix such that when diagonalizing, the eigenvalues found are reasonable approximations of the actual eigenvalues. This method will be used on a model known as the Heisenberg Spin Chain. This model is used when there are a group of particles that contain a spin interact with each other. The spin of each particle interacting with the adjacent particle participates in influencing the total energy of the system. This effect of the spin is noticeable when applying the Pauli spin operators as they extract the effect of the spin from a wave function.
The Hamiltonian of a Spin chain can be described using the following formula.

$$H = \sum_{i=0}^{L-1} \vec{S}_i \cdot \vec{S}_{i+1} = \frac{1}{2}\left(\sum_{i=0}^{L-1} P_{ij} - \frac{L}{2} I\right) \tag{17}$$

Each of these spin operators act on a particle i and the particle beside $i+1$ extracting the spin. This Hamiltonian operator can also be written in the form of a permutation matrix. The spin operator is known as $S = 1/2\,(\sigma_i^x, \sigma_i^y, \sigma_i^z)$. The sigma operators are the Pauli matrices. Preforming the Hamiltonian matrix on a wave function will produce output the energy that is caused by the spins of the system. Since the spin is either up or down, the system can be reduced to a binary unit which acts upon a vector either turning on or off a component of the vector the operator is applied on. The function provided in the HW sheet known as "hv" does this where it outputs a vector y after the Hamiltonian operator has acted upon a random vector v. The size of the Hamiltonian is dependant on how many particles are within the system, known as 2^L where there are L amount of particles. Since this matrix can get exceedingly large, Lanczos method is applied.

Lanczos method takes a large matrix and builds a tridiagonal matrix which becomes quite simple to evaluate using the previous Jacobi Diagonalization method built previously. The method is as followed.
1. Create a random vector and orthonormalize it. This is known as v_0
2. Apply the Hamiltonian operator on this vector. This is known as w
3. Apply the inner product with v_0 and w. This is known as alpha. Each alpha will relate to the diagonal elements of the Lanczos matrix per iteration.
4. Create new vector u_0 which is defines as $w - \alpha_0 * v_0$
5. Create a for loop from 0 to M-2 where M is the size of Lanczos matrix
6. Normalize u_n to solve for beta. These are the surrounding diagonal elements of the lanczos matrix
7. v_{n+1} is u_n vector divided by beta 8. w is the Hamiltonian acting upon v_{n+1} minus $\beta_n * v_n$
9. α_{n+1} is the inner product of v_{n+1} and w
10. u_{n+1} is the w vector minus α_{n+1} times v_{n+1} 11. End loop and apply Jacobi on Lanczos matrix made by alpha and beta terms

The above steps are known as the modified Gram-Schmidt process. The final matrix should look like the following.

$$\text{Lanczos} = \begin{pmatrix} \alpha_0 & \beta_0 & & & & & & & \\ \beta_0 & \alpha_1 & \beta_1 & & & & & & \\ & \beta_1 & \alpha_3 & \beta_2 & & & & & \\ & & \beta_2 & \alpha_4 & \beta_3 & & & & \\ & & & .. & .. & .. & & & \\ & & & & .. & .. & .. & & \\ & & & & & .. & .. & .. & \\ & & & & & & \beta_{M-3} & \alpha_{M-2} & \beta_{M-2} \\ & & & & & & & \beta_{M-2} & \alpha_{M-1} \end{pmatrix} \tag{18}$$

The size of this matrix is significantly smaller than the initial Hamiltonian matrix that would have to be diagonalized. Thus when implementing the Jacobians on this, it is quite simple to solve. The C++ code has functions to preform linear algebra operations such as bra ket multiplication, constants multiplied to a vector and adding vectors with each other. There is a functions for each of the operations and also for normalization of a vector. For vector subtraction or a vector divided by a constant, by manipulating the addition and constant multiplication functions, these can be evaluated correctly. The figure below shows the following algorithm to produce the lanczos matrix.
The code found in the figure above compute the steps provided above exactly. Now that the algorithm

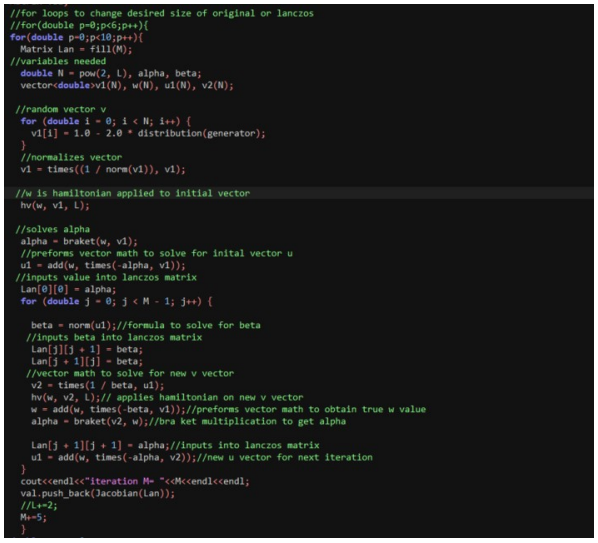

Figure 6: Lanczos Algorithm

has been created, the output of the algorithm can be explored. The initial desired measurements are

to explore the results after changing the size of the Lanczos matrix. The size of this matrix will range from 5-50 incrementing in values of 5. The following figures show the initial eigenvalues and the convergence of the eigen values. The size of the Hamiltonian matrix will be 2^10.

The data in the figure above shows that the trend when increasing the size of the Lanczos matrix

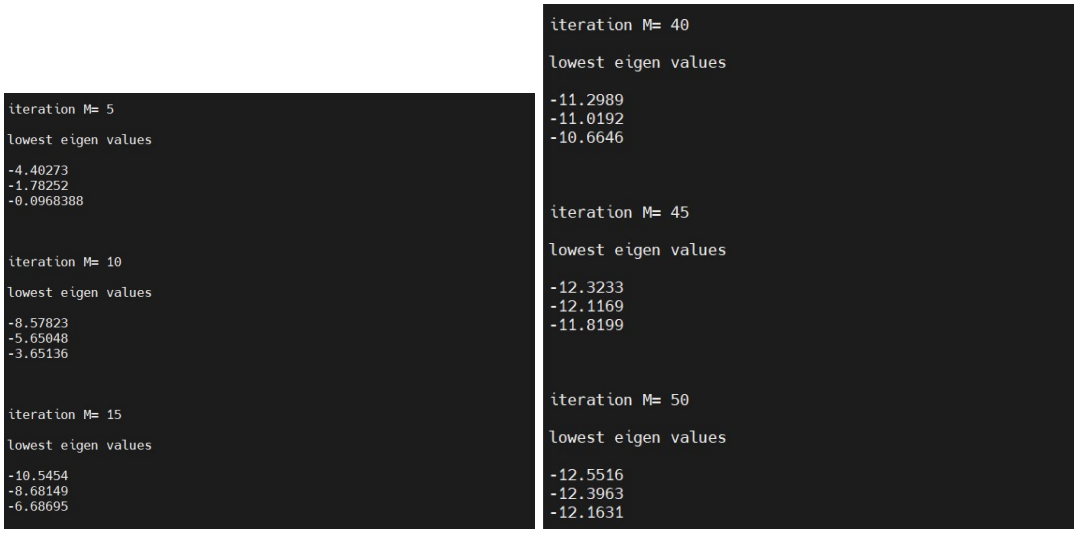

Figure 7: Eigenvalues when changing size of Lanczos matrix

refines the eigenvalues such that a convergence value is approached. This is likely due to the fact that the size is closer to the size of the initial matrix so there are more terms that are able to give importance to the construction of the eigenvalues of the matrix. This can be further explored by changing the amount of particle in the system from 10 to 14. This means the size of the Hamiltonian matrix will be 2^14. If the trend is the same, this clarifies that the larger the size of the lanczos matrix the more accurate the eigenvalues.

Both respective sizes of the Hamiltonian matrix produce data that follows the same trend. By

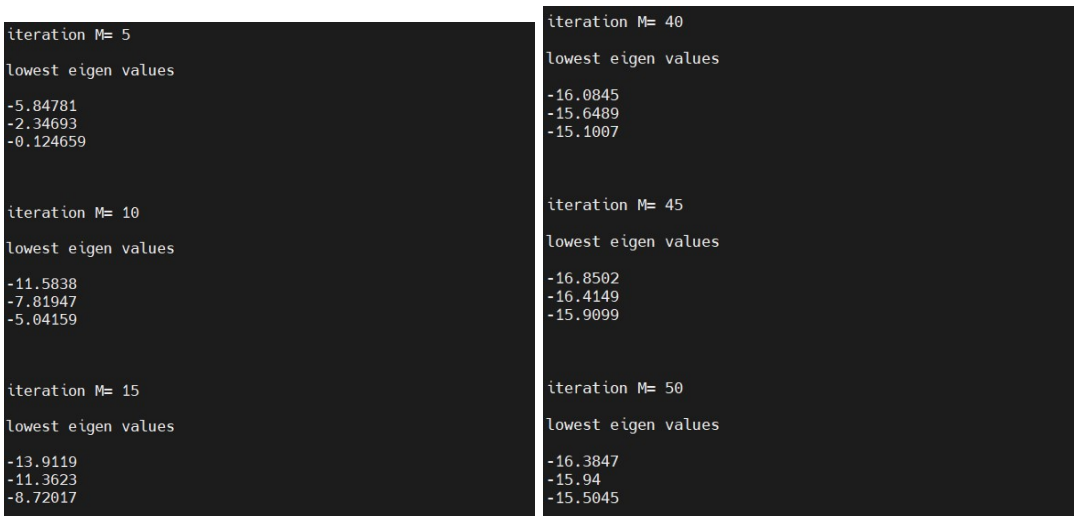

Figure 8: Eigenvalues after changing amount of particles in system

increasing the size of the Lanczos matrix, a convergence value is reached for the lowest eigenvalues of the system.

A piece of data given in the HW sheet is that $E^0_{L=10} = -4.5154463544$). Note that the values found in the convergence process computed is not the same for the Hamiltonian with 10 particles. This may be due to a stability issue in the Jacobian's algorithm or the implementation of the modified Gram Schmidt decomposition. To explore this, the lowest eigen values are plotted and the deviance from the

actual convergence value.

The plot shows that the lowest energy levels energy gap is quite small as the size of the Lanczos matrix

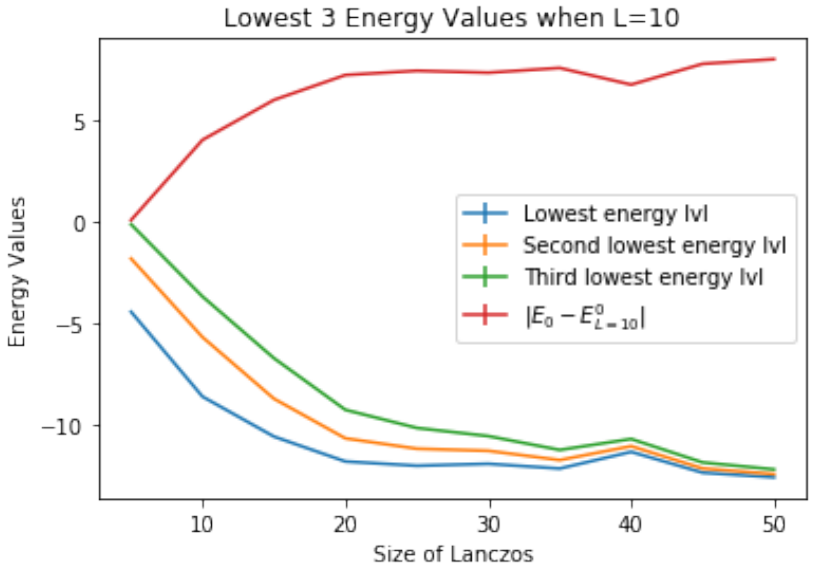

Figure 9: Deviance of Eigenvalues with 10 particles

increases. Also as the size of the Lanczos matrix increase the deviance from the actual convergence value starts to become constant. This may be reasonable as the actual eigenvalue can be extracted by applying an operation to the eigenvalue found in the simulation using the constant. This may be caused by the robustness of the algorithm built. This property can also be explored with the system that contains 14 particles. Although the convergence energy is not the same as the system with 10 particles, for the sake of observing the behavior of the data, this value is used again for the convergence.

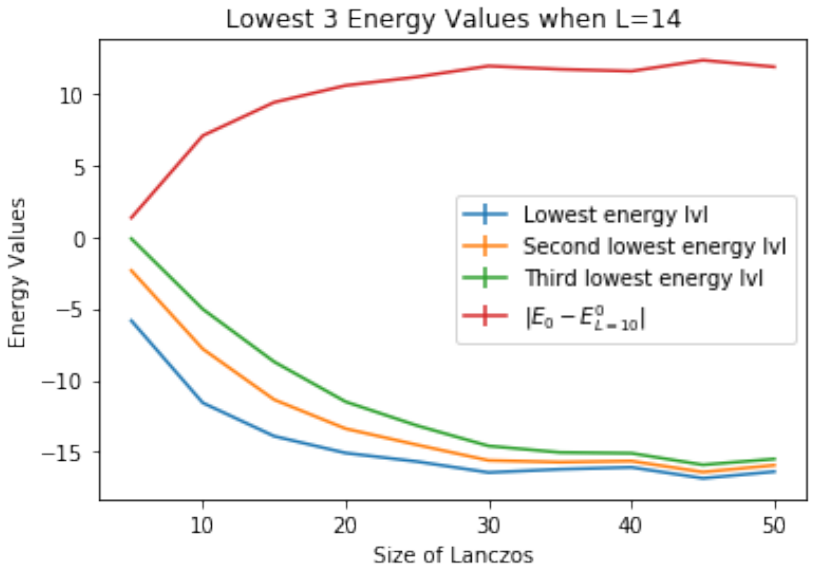

Figure 10: Deviance of Eigenvalues with 14 particles

The behavior is the exact same as the previous data set as expected. As the size of the Lanczos matrix reaches 30, the values of the data start to become reasonably consistent even while increasing the size of the matrices. As such the next part will use a Lanczos matrix with a size of 30x30 elements. The next concept studied is proof of the thermodynamic energy limit by measuring the energy gap

between the lowest two energy values while increasing the amount of particles. The amount of particles initially used is 6 and the final amount of particles is 18. This data set is only considering amount of particles that are even.

Conceptually the energy gap decreasing seems counter intuitive as with more particles, the amount

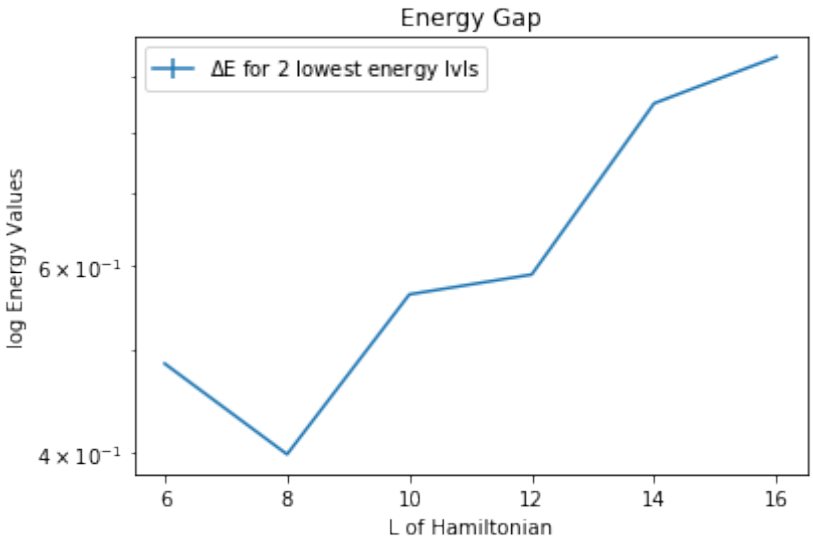

Figure 11: Energy Gap

of energy required for the ground state to achieve an excited state should be more. This is shown by the data as the more particles, the greater the energy gap of the eigenvalues. Quantum physics states and energy values are quantized thus it is reasonable to assume that with more particles in a system, the ground state will be considerably lower then the next energy level as there are more particles that need to be excited. This next level of excitation can be conceptualized as the sum of all there excitation energies required for each individual particle within the system. This is also consistent with thermodynamics as the most relaxed state should have a lower energy than a disorderly system caused by an excitation in terms of magnitude, thus causing an energy gap that is greater than 0. Quantum physics also states that no particle can ever attain an energy of zero but also a system has known energy values which has to have an energy gap such that a system can have relaxed and excited states.

3 Conclusion

This report showed how linear algebra can be applied to complex quantum physics problems and to provide more accurate data readings. Since data generally gets more accurate with more data sets provided by creating a method that takes important values and preforming a method of diagonalizing matrices to obtain eigenvalues, substantial pieces of information can be gathered pertaining to each system.

4 Code Appendix

4.1 Code Q1

```
// 4G03_assign23_q1.cpp : This file contains the 'main' function. Program execution begins and ends t
//
#include <iostream>
#include <math.h>
#include<stdio.h>
#include <iomanip>
#include <ctime>
#include <vector>
#include <random>
#include <algorithm>
using namespace std;
using Matrix = vector<vector<double>>;
//function to return minimum value given 2 numbers
double min(double p, double q) {
        double val;
        if (p < q) {
                val = p;
        }
        else {
                val = q;
        }
        return val;
}
//creating hamiltonian matrix given set size
Matrix Hamiltonian(double L) {
        Matrix C;
        for (double q = 0; q < L; q++) {

                vector<double>v;
                for (double o = 0; o < L; o++) {
                        v.push_back(1);
                }
                C.push_back(v);
        }
        for (double p = 0; p < (L - 1); p++) {
                double q = p + 1;
                for (q; q < L; q++) {
                        double pp = p + 1, qq = q + 1;
                        C[p][q] = min(pp, qq) * (0.05 + 5 * pow((-1), abs(pp - qq)));
                        C[q][p] = C[p][q];
                }
        }
        for (double i = 0; i < L; i++) {
                C[i][i] = pow((i + 1), 2) + (i + 1) * (0.05 + 5 * pow(-1, abs((i + 1) - (i + 1))));
        }
        return C;
}
//taking diagonal elements of final matrix and storing in vector for eigenvalues
vector<double>EN(Matrix A) {
        vector<double> Eigenvalues;
        for (double i = 0; i < A.size(); i++) {
                Eigenvalues.push_back(A[i][i]);
        }
```

```
        return Eigenvalues;
}//printing a vector
double printv(vector<double> A) {
        for (double i = 0; i < A.size(); i++) {
                if (abs(A[i]) < 0.000001) {//precision
                        A[i] = 0;
                }
                cout << A[i] << ", ";
        }
        return 0;
}//print matrix used for debugging
double print_M(Matrix A) {
        for (int i = 0; i < A.size(); i++) {
                for (int j = 0; j < A.size(); j++) {
                        if (abs(A[i][j]) < 0.0001) {//precision
                                A[i][j] = 0;
                        }
                        cout << A[i][j] << " ";
                }
                cout << endl;
        }
        return 0;
}
//used to ensure the matrix approaches a completely diagonalized matrix
double sumofsqr(Matrix A) {
        double sum = 0;
        for (double p = 0; p < (A.size() ); p++) {

                for (double q=0; q < A.size(); q++) {
                        if (q == p) {
                                sum += 0;
                        }else{
                                sum +=abs(A[p][q]) * abs(A[p][q]);
                        }

                }
        }return sum;
}

//returns p,q values of a matrix
pair<double, double> maxx(Matrix A) {
        double num = 0; pair<double, double>pn;

        for (double p = 0; p < (A.size() - 1); p++) {
                double q = p + 1;
                for (q; q < A.size(); q++) {
                        if (abs(A[p][q]) > abs(num)) {
                                num = abs(A[p][q]);
                                pn.first = p; pn.second = q;
                        }
                }
        }
        return pn;
}
//preforms matrix multiplication
Matrix Mult(Matrix A, Matrix B) {
```

```
        Matrix C;
        for (double q = 0; q < A.size(); q++) {

                vector<double>v;
                for (double o = 0; o < A.size(); o++) {
                        v.push_back(1);
                }
                C.push_back(v);
        }


        for (int i = 0; i < A.size(); i++) {
                for (int j = 0; j < B.size(); j++) {
                        C[i][j] = 0;
                        for (int k = 0; k < A.size(); k++)
                                C[i][j] += A[i][k] * B[k][j];

                }

        }
        return C;
}
//transposes any matrix
Matrix Transpose(Matrix A) {
        Matrix B;
        for (double i = 0; i < A.size(); i++) {
                vector<double>v;
                for (double n = 0; n < A.size(); n++)
                {
                        v.push_back(0);
                }
                B.push_back(v);
        }
        for (int i = 0; i < A.size(); ++i) {
                for (int j = 0; j < A.size(); ++j) {
                        B[i][j] = A[j][i];
                }
        }
        return B;
}//creates rotation given pq, size of matrix and angle
Matrix Rotation(double p, double q, double size, double theta/*t*/) {
        Matrix roti;
        for (double i = 0; i < size; i++) {
                vector<double>v;
                for (double n = 0; n < size; n++)
                {
                        v.push_back(0);
                }
                roti.push_back(v);
        }
        for (double o = 0; o < size; o++)
        {
                roti[o][o] = 1;
        }
        /*double c = 1 / (sqrt(t * t + 1));
        double s = t * c;*/
```

```
        roti[p][p] = cos(theta);
        roti[q][q] = cos(theta);
        roti[q][p] = sin(theta);
        roti[p][q] = -sin(theta);
        /*roti[p][p] = c;
        roti[q][q] = c;
        roti[q][p] = s;
        roti[p][q] = -s;*/
        return roti;
}//returns lowest 3 eigen values
double lEV(vector<double> A) {
        sort(A.begin(), A.end());
        cout << "3 lowest eigen values: " << endl << endl;
        for (double i = 0; i < 3; i++) {
                cout << A[i] << endl;
        }
        return 0;
}
double sgn(double x) {
        double sign = x / abs(x);
        return sign;
}
int main()
{
        //commented sections used for debugging

        double L = 5;
        for (double e = 1; e < 2; e++) {//loop to change M values

                Matrix A = Hamiltonian(L);
                cout << "Hamiltonian Matrix M=5: " << endl << endl;
                print_M(A);

                //cout << "Hamiltonian @ desired M value" << endl << endl;

                double sum = sumofsqr(A);

                auto IN = maxx(A);
                double p = IN.first, q = IN.second;//coordinates
                sum -= 2 * abs(A[p][q]) * abs(A[p][q]);//sum of squares for off diagonal elements

                double phi = (A[q][q] - A[p][p]) / (2 * A[p][q]);//finding phi angle

                double t = phi + sqrt(phi * phi + 1);//using t relationship

                //double t=sgn(phi) / (abs(phi) + sqrt(phi * phi + 1));
                double theta = atan(t);//finds theta angle

                Matrix J1 = (Rotation(p, q, A.size(), theta));//preforms rotation

                A = Mult(A, J1);
                A = Mult(Transpose(J1), A);//applies rotation matrix to transform matrix


                //first iteration outside of loop to store eigenvectors if preferred
                while (abs(sum) > 0.0000001) {//resolution based on offdiagonal elements
```

```
                        auto IN = maxx(A);//finds coordinates
                        double p = IN.first, q = IN.second;
                        sum -= 2 * abs(A[p][q]) * abs(A[p][q]);

                        double phi = (A[q][q] - A[p][p]) / (2 * A[p][q]);

                        double t = phi + sqrt(phi * phi + 1);//obtatins angle
                        double theta = atan(t);

                        Matrix J = (Rotation(p, q, A.size(), theta));//rotation matrix
;
                        J1 = Mult(J1, J);//eigen vectors

                        A = Mult(A, J);
                        A = Mult(Transpose(J), A);//applies rotation



                }
                cout << endl << endl;
                cout << "M of matrix: " << L << endl << endl;

                print_M(A); cout << endl << endl;
                cout << "eigen vectors: " << endl << endl;
                print_M(J1);
                cout << endl << endl;
                cout << "eigen values: " << endl << endl;
                cout << endl << endl;
                printv(EN(A));
                cout << endl << endl;
                lEV(EN(A));
                cout <<endl<<endl<<"Sum of Squares:"<<sum<< endl << endl;

                L += 10;
        }

}
```

4.2 Code Q2

```
#define BIT_SET(a,b) ((a) |= (1U<<(b)))
#define BIT_CLEAR(a,b) ((a) &= ~(1U<<(b)))
#define BIT_FLIP(a,b) ((a) ^= (1U<<(b)))
#define BIT_CHECK(a,b) ((bool)((a) & (1U<<(b))))
// Set on the condition f else clear
bool f; // conditional flag
unsigned int m; // the bit mask
unsigned int w; // the word to modify: if (f) w |= m; else w &= ~m;
#define COND_BIT_SET(a,b,f) ((a) = ((a) & ~(1U<<(b))) | ((-(unsigned int)f) & (1U<<(b))))
#include <vector>
#include <climits>
#include <cmath>
#include <iostream>
#include <stdio.h>
#include <iomanip>
#include <ctime>
#include <random>
#include <algorithm>
#include <iterator>
#include <fstream>
using namespace std;
using Matrix = vector<vector<double>>;
//function to return minimum value given 2 numbers
double min(double p, double q) {
        double val;
        if (p < q) {
                val = p;
        }
        else {
                val = q;
        }
        return val;
}
//taking diagonal elements of final matrix and storing in vector for eigenvalues
vector<double>EN(Matrix A) {
        vector<double> Eigenvalues;
        for (double i = 0; i < A.size(); i++) {
                Eigenvalues.push_back(A[i][i]);
        }
        return Eigenvalues;
}//printing a vector
double printv(vector<double> A) {
        for (double i = 0; i < A.size(); i++) {
                if (abs(A[i]) < 0.000001) {
                        A[i] = 0;
                }
                cout << A[i] << ", ";
        }
        return 0;
}//print matrix used for debugging
double print_M(Matrix A) {
        for (int i = 0; i < A.size(); i++) {
                for (int j = 0; j < A.size(); j++) {
                        if (abs(A[i][j]) < 0.00001) {
                                A[i][j] = 0;
```

```
                        }
                        cout << A[i][j] << " ";
                }
                cout << endl;
        }
        return 0;
}//used to ensure the matrix approaches a completely diagonalized matrix
double sumofsqr(Matrix A) {
        double sum = 0;
        for (double p = 0; p < (A.size()); p++) {

                for (double q = 0; q < A.size(); q++) {
                        if (q == p) {
                                sum += 0;
                        }
                        else {
                                sum += abs(A[p][q]) * abs(A[p][q]);
                        }

                }
        }return sum;
}//returns p,q values of a matrix
pair<double, double> maxx(Matrix A) {
        double num = 0; pair<double, double>pn;

        for (double p = 0; p < (A.size() - 1); p++) {
                double q = p + 1;
                for (q; q < A.size(); q++) {
                        if (abs(A[p][q]) > abs(num)) {
                                num = abs(A[p][q]);
                                pn.first = p; pn.second = q;
                        }
                }
        }
        return pn;
}//preforms matrix multiplication
Matrix Mult(Matrix A, Matrix B) {
        Matrix C;
        for (double q = 0; q < A.size(); q++) {

                vector<double>v;
                for (double o = 0; o < A.size(); o++) {
                        v.push_back(1);
                }
                C.push_back(v);
        }



        for (int i = 0; i < A.size(); i++) {
                for (int j = 0; j < B.size(); j++) {
                        C[i][j] = 0;
                        for (int k = 0; k < A.size(); k++)
                                C[i][j] += A[i][k] * B[k][j];

                }
```

```
        }
        return C;
}//transposes any matrix
Matrix Transpose(Matrix A) {
        Matrix B;
        for (double i = 0; i < A.size(); i++) {
                vector<double>v;
                for (double n = 0; n < A.size(); n++)
                {
                        v.push_back(0);
                }
                B.push_back(v);
        }
        for (int i = 0; i < A.size(); ++i) {
                for (int j = 0; j < A.size(); ++j) {
                        B[i][j] = A[j][i];
                }
        }
        return B;
}//creates rotation given pq, size of matrix and angle
Matrix Rotation(double p, double q, double size, double theta) {
        Matrix roti;
        for (double i = 0; i < size; i++) {
                vector<double>v;
                for (double n = 0; n < size; n++)
                {
                        v.push_back(0);
                }
                roti.push_back(v);
        }
        for (double o = 0; o < size; o++)
        {
                roti[o][o] = 1;
        }

        roti[p][p] = cos(theta);
        roti[q][q] = cos(theta);
        roti[q][p] = sin(theta);
        roti[p][q] = -sin(theta);
        return roti;
}//sgn function
double sgn(double x) {
        double sign = x / abs(x);
        return sign;
}//returns lowest 3 eigen values
vector<double> lEV(vector<double> A) {
        sort(A.begin(), A.end());
  vector<double> B(3);

        for (double i = 0; i < 3; i++) {

    B[i]=A[i];
        }

        return B;
```

```
}
//applied hamiltonian operator on a vector and changes parameter vector y
void hv(vector<double>& y, const vector<double>& x, int L)
{
        for (vector<double>::iterator it = y.begin(); it != y.end(); ++it)
                *it = 0.0;
        bool b;
        unsigned int k;
        for (unsigned int i = 0; i < x.size(); i++) {
                if (abs(x[i]) > 2.2e-16) {
                        int jm = L - 1;
                        double xov2 = x[i] / 2.0;
                        for (int j = 0; j < L; j++) {
                                k = i;
                                COND_BIT_SET(k, jm, BIT_CHECK(i, j));
                                COND_BIT_SET(k, j, BIT_CHECK(i, jm));
                                y[k] += xov2;
                                jm = j;
                        }
                }
        }
        for (unsigned int i = 0; i < x.size(); i++)
                y[i] = y[i] - ((double)L) / 2.0 * x[i] / 2.0;
}//jacobi diagonalization algorithm
vector<double> Jacobian(Matrix A) {

        double count = 1;

        double sum = sumofsqr(A);

        auto IN = maxx(A);
        double p = IN.first, q = IN.second;
        sum -= 2 * abs(A[p][q]) * abs(A[p][q]);

        double phi = (A[q][q] - A[p][p]) / (2 * A[p][q]);

        double t = phi + sqrt(phi * phi + 1);
        double theta = atan(t);

        Matrix J1 = (Rotation(p, q, A.size(), theta));

        A = Mult(A, J1);
        A = Mult(Transpose(J1), A);

        count += 1;

        while (abs(sum) > 0.0000001) {

                auto IN = maxx(A);
                double p = IN.first, q = IN.second;
                sum -= 2 * abs(A[p][q]) * abs(A[p][q]);

                double phi = (A[q][q] - A[p][p]) / (2 * A[p][q]);

                double t = phi + sqrt(phi * phi + 1);
                double theta = atan(t);
```

```
                Matrix J = (Rotation(p, q, A.size(), theta));

                J1 = Mult(J1, J);

                A = Mult(A, J);
                A = Mult(Transpose(J), A);

                count += 1;

        }

  cout << "lowest eigen values";
  cout << endl << endl;
  for(double l=0;l<3;l++){
    cout<<lEV(EN(A))[l]<<endl;
  }
        //row ,column
  cout<<endl<<endl;
        return lEV(EN(A));
}//bra ket multiplication
double braket(vector<double>a, vector<double>b) {
        double sum = 0;
        if (a.size() != b.size()) {
                cout << "Cannot compute, size of vectors do not match" << endl;
        }
        else {
                for (double i = 0; i < a.size(); i++) {
                        sum += a[i] * b[i];
                }

        }return sum;
}//fills matrix with 0
Matrix fill(double M) {
        Matrix A;
        for (double i = 0; i < M; i++) {
                vector<double>v;
                for (double n = 0; n < M; n++)
                {
                        v.push_back(0);
                }
                A.push_back(v);
        }
        return A;
}//normalization of vector
double norm(vector<double> v) {
        vector<double> v1 = v;
        double n = (sqrt(braket(v1, v1)));
        return n;
}//constant multiplied to vector
vector<double> times(double a, vector <double> b) {
        vector<double>p(b.size());
        for (double i = 0; i < b.size(); i++) {
                p[i] = a * b[i];
        }
        return p;
```

```
}//adding two vectors
vector<double> add(vector<double> a, vector <double> b) {
        vector<double>p(b.size());
        for (double i = 0; i < b.size(); i++) {
                p[i] = a[i] + b[i];
        }
        return p;
}
int main() {
        mt19937 generator;
        uniform_real_distribution<double> distribution(0, 1.0);
        //double L = 6,M = 30; //2.3
  double L = 10,M = 5;//size of original and new matrix respectively 2.2
  Matrix val;
  //for loops to change desired size of original or lanczos
  //for(double p=0;p<6;p++){
  for(double p=0;p<10;p++){
    Matrix Lan = fill(M);
  //variables needed
        double N = pow(2, L), alpha, beta;
        vector<double>v1(N), w(N), u1(N), v2(N);

   //random vector v
        for (double i = 0; i < N; i++) {
                v1[i] = 1.0 - 2.0 * distribution(generator);
        }
    //normalizes vector
        v1 = times((1 / norm(v1)), v1);

   //w is hamiltonian applied to initial vector
        hv(w, v1, L);

   //solves alpha
        alpha = braket(w, v1);
    //preforms vector math to solve for inital vector u
        u1 = add(w, times(-alpha, v1));
   //inputs value into lanczos matrix
        Lan[0][0] = alpha;
        for (double j = 0; j < M - 1; j++) {

                beta = norm(u1);//formula to solve for beta
     //inputs beta into lanczos matrix
                Lan[j][j + 1] = beta;
                Lan[j + 1][j] = beta;
     //vector math to solve for new v vector
                v2 = times(1 / beta, u1);
                hv(w, v2, L);// applies hamiltonian on new v vector
                w = add(w, times(-beta, v1));//preforms vector math to obtain true w value
                alpha = braket(v2, w);//bra ket multiplication to get alpha

                Lan[j + 1][j + 1] = alpha;//inputs into lanczos matrix
                u1 = add(w, times(-alpha, v2));//new u vector for next iteration
        }
    cout<<endl<<"iteration M= "<<M<<endl<<endl;
    val.push_back(Jacobian(Lan));
    //L+=2;
```

```
    M+=5;
    }
  double count=0;
  //ofstream MyFile("Eigenvalues_L.txt");//2.3

  ofstream MyFile("Eigenvalues_M.txt");//2.2
        //for (double j = 6; j <= 18; j = j + 2) {//2.3
  for (double j = 5; j <= 50; j = j + 5) {
    MyFile << j << "," <<val[count][0]<<","<<val[count][1]<<","<< val[count][2]<<","<<abs(val[count][
              //MyFile << j << "," <<val[count][1] -val[count][0]<< endl;//2.3
              count += 1;
        }
        MyFile.close();
}
```

4.3 Python Plotting for Magnetization Histogram

```
import matplotlib.pyplot as plt
import numpy as np

X1,Y1,Y2,Y3,Y4= np.loadtxt('Eigenvalues_M14.txt', delimiter=',', unpack=True)
fig1, ax = plt.subplots()
plt.title('Lowest 3 Energy Values when L=14')
plt.xlabel('Size of Lanczos')
plt.ylabel('Energy Values')
ax.errorbar(X1, Y1, label='Lowest energy lvl',
            yerr=0,
            capsize=0)


ax.errorbar(X1, Y2, label='Second lowest energy lvl',
            yerr=0,
            capsize=0)



ax.errorbar(X1, Y3, label='Third lowest energy lvl',
            yerr=0,
            capsize=0)
ax.errorbar(X1, Y4, label='$|E_0-E_{L=10}^0|$',
            yerr=0,
            capsize=0)
plt.legend()
plt.show()
```

4.4 Python Plotting for rest of data

```
import matplotlib.pyplot as plt
import numpy as np

X1,Y1,Y2,Y3,Y4= np.loadtxt('Eigenvalues_M.txt', delimiter=',', unpack=True)
fig1, ax = plt.subplots()
plt.title('Lowest 3 Energy Values when M=10')
plt.xlabel('Size of Lanczos')
plt.ylabel('Energy Values')
ax.errorbar(X1, Y1, label='Lowest energy lvl',
            yerr=0,
            capsize=0)


ax.errorbar(X1, Y2, label='Second lowest energy lvl',
            yerr=0,
            capsize=0)



ax.errorbar(X1, Y3, label='Third lowest energy lvl',
            yerr=0,
            capsize=0)
ax.errorbar(X1, Y4, label='$|E_0-E_{L=10}^0|$',
            yerr=0,
            capsize=0)
plt.legend()
plt.show()
```

4.5 Python Plotting for rest of data

```
import matplotlib.pyplot as plt
import numpy as np
import math

X1,Y1= np.loadtxt('Eigenvalues_L.txt', delimiter=',', unpack=True)
fig1, ax = plt.subplots()

plt.title('Energy Gap')
plt.xlabel('L of Hamiltonian')
plt.ylabel('log Energy Values')
ax.errorbar(X1, Y1, label='$\Delta$E for 2 lowest energy lvls',
            yerr=0,
            capsize=0)

plt.yscale("log")
plt.legend()
plt.show()
```