Homework 3

Zachary Giles

October 2022

Exercise 1

In this exercise we will use matrix methods to find the eigenvalues of the Hamiltonian matrix for a particle in a potential

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

Our basis are the solution to the infinite square well of length L. In this case since the potential diverges at $x = \pi/2$ we have $L = \pi/2$. Since the matrix is infinite dimensional we will have to truncate it. An arbitrary matrix element is given by

$$\begin{aligned} H_{nm} &= \delta_{nm}E_n^0 + \langle\psi_n|\, V\, |\psi_m\rangle \\ &= \delta_{nm}\frac{2n^2\hbar^2}{m} + \int_0^{\frac{\pi}{2}} \frac{4\hbar^2}{2m\pi}\sin(2nx)\sin(2mx)\left(\frac{0.05}{\sin^2(x)} + \frac{5}{\cos^2(x)}\right)\mathrm{d}x \\ &= \frac{\hbar^2}{2m}\left(4n^2\delta_{nm} + \frac{4}{\pi}\left(0.05\pi\min(n,m) + 5(-1)^{|m-n|}\pi\min(n,m)\right)\right) \\ &= \frac{\hbar^2}{2m}\left(4n^2\delta_{nm} + 4\min(n,m)(0.05 + 5(-1)^{|n-m|})\right) \end{aligned}$$

For $M = 5$ this gives the following matrix

$$\frac{\hbar^2}{2m}\begin{pmatrix} 24.2 & -19.8 & 20.2 & -19.8 & 20.2 \\ -19.8 & 56.4 & -39.6 & 40.4 & -39.6 \\ 20.2 & -39.6 & 96.6 & -59.4 & 60.6 \\ -19.8 & 40.4 & -59.4 & 144.8 & -79.2 \\ 20.2 & -39.6 & 60.6 & -79.2 & 201 \end{pmatrix}$$

Notice this is a symmetric matrix. We want to compute the eigenvalues of this matrix using Jacobi rotations. Our program consists of a Makefile, main program with a Kronecker delta function (`int delta(int n, int m)`) and function to compute H_{nm} (`double H(int n, int m)`), and a file `diag.cc` with a sign function `int sgn(double val)`, the Jacobi rotation function `void jacdiag`

`(Matrix & A, vector<double> & d)` and a modified Jacobi rotation function which works on the top half of a matrix `void jacuppdiag(Matrix & A, vector<double> & d)`.

For the naive implementation go through each point in the upper diagonal (p, q). We then march down the p and q rows and columns and perform our computations

```
// Compute variabes for calculation
double alpha = (A[q][q]-A[p][p])/(2*A[p][q]);
double t = sgn(alpha)/(fabs(alpha)+sqrt(pow(alpha,2)+1));
double c = 1/sqrt(pow(t,2)+1);
double s = t*c;
// Save these values so we don't use the new value
// in a later calculation
double Apq = A[p][q];
double App = A[p][p];
double Aqq = A[q][q];
// Essentially we march along the p and q columns
// and row, with special calculations for (p,p),
// (q,q) and (p,q)
for(int k = 0; k < size; k++){
    if(k == p){
        A[p][p] = pow(c,2)*App+pow(s,2)*Aqq-2*s*c*Apq;
        A[p][q] = 0;
        A[q][p] = 0;
    } else if (k == q){
        A[q][q] = pow(s,2)*App+pow(c,2)*Aqq+2*s*c*Apq;
    } else {
        // Save old values so new values don't
        // change calculation
        double Akp = A[k][p];
        double Akq = A[k][q];
        double Apk = A[p][k];
        double Aqk = A[q][k];
        A[p][k] = c*Apk-s*Aqk;
        A[q][k] = s*Apk+c*Aqk;
        A[k][p] = c*Akp-s*Akq;
        A[k][q] = s*Akp+c*Akq;
    }
}
```

This just manually performs the rotation on all the elements. For the upper diagonal we modify slightly to only march down the rows and columns to the diagonal and then reflect the values to conserve symmetry.

```
// Compute variables for the calculations
double alpha = (A[q][q]-A[p][p])/(2*A[p][q]);
```

```
double t = sgn(alpha)/(fabs(alpha)+sqrt(pow(alpha,2)+1));
double c = 1/sqrt(pow(t,2)+1);
double s = t*c;
double tau = s/(1+c);
// Create a copy of A so that the newly calculated
// values don't affect calculaitons down the line
Matrix copy (A);
// Note that all these calculation stop at the
// main diagonal and their values are reflected
// over the diagonal. March down p and q columns
for(int k = 0; k < q; k++){
    if(k < p){
        A[k][p] = copy[k][p]-s*(copy[k][q]+tau*copy[k][p]);
        A[k][q] = copy[k][q]+s*(copy[k][p]-tau*copy[k][q]);
        A[p][k] = A[k][p];
        A[q][k] = A[k][q];
    } else {    // When k > p only q column is above diagonal
        A[k][q] = copy[k][q]+s*(copy[k][p]-tau*copy[k][q]);
        A[q][k] = A[k][q];
    }
}
// March along p and q rows
for(int k = size-1; k > p; k--){
    if(k > q){
        A[p][k] = copy[p][k]-s*(copy[q][k]+tau*copy[p][k]);
        A[q][k] = copy[q][k]+s*(copy[p][k]-tau*copy[q][k]);
        A[k][p] = A[p][k];
        A[k][q] = A[q][k];
    } else {    // When k < q only p row is above diagonal
        A[p][k] = copy[p][k]-s*(copy[q][k]+tau*copy[p][k]);
        A[k][p] = A[p][k];
    }
A[p][p] = copy[p][p]-t*copy[p][q];
A[q][q] = copy[q][q]+t*copy[p][q];
A[p][q] = 0;
A[q][p] = 0;
}
```

For each the sum of squares of the upper diagonal is computed to ensure convergence.

We will look at the lowest 3 eigenvalues of the H matrix for sizes $M = 10$, 20,

30, and 40. The eigenvalues we obtain are

$M = 10$: 14.7402, 34.1, 61.4636
$M = 20$: 14.739, 34.0968, 61.4556
$M = 30$: 14.7387, 34.0959, 61.4536
$M = 40$: 14.7385, 34.0954, 61.4527

We see that by $M = 40$ we have about 3 decimal places of reliability. Our eigenvalues are thus

14.739, 34.095, 61.453

Exercise 2

We know look at applying Jacobi rotation to the spin chain system. To do this we want a simpler matrix to work with, so we will use Lanczos' method to create a tridiagonal matrix representing our system.

Our program consists of a Makefile, main program, a file `hv.cc` which contains a function `void hv(vector<double> & y, const vector<double> & x)` which takes a vector x and computes Hx, and a file *Lan.cc* which contains a function `void makelan(Matrix& Lan, int m, int N, int L)` which takes a matrix and generates the tridiagonal matrix using Lanczos' method.

Our implementation of Lanczos' method is pretty simple. We run the algorithm to generate the first α and β (which are computed slightly different form the other ones)

```
for (int i = 0; i < N; i++){
    v1[i] = v1[i]/mag;
}
hv(omega, v1, L);
alpha.push_back(dotprod(v1, omega));
beta.push_back(sqrt(dotprod(omega, omega)-pow(alpha[0],2)));
for(int i = 0; i < N; i++){
    v2[i] = (omega[i]-alpha[0]*v1[i])/beta[0];
}
```

This also gives us $|v_1\rangle$, which we store in `v2` while the old values are stored in `v1`, and `omega` stores $H|v_0\rangle$. With these initial values we can then run the algorithm

```
for (int i = 1; i < m-1; i++){
    hv(omega, v2, L);
    for (int j = 0; j < N; j++){
        omega[j] = omega[j] - beta[i-1]*v1[j];
    }
```

```
        alpha.push_back(dotprod(v2, omega));
        beta.push_back(sqrt(dotprod(omega,omega)-pow(alpha[i],2)));
        for (int j = 0; j < N; j++){
            v1[j] = v2[j];
            v2[j] = (omega[j] - alpha[i]*v1[j])/beta[i];
        }
    }
```

This is a literal implementation of the algorithm with the formula for $|f_i\rangle$ being substituted instead of using another to store its values. We take advantage of the orthogonality to simplify the equation.

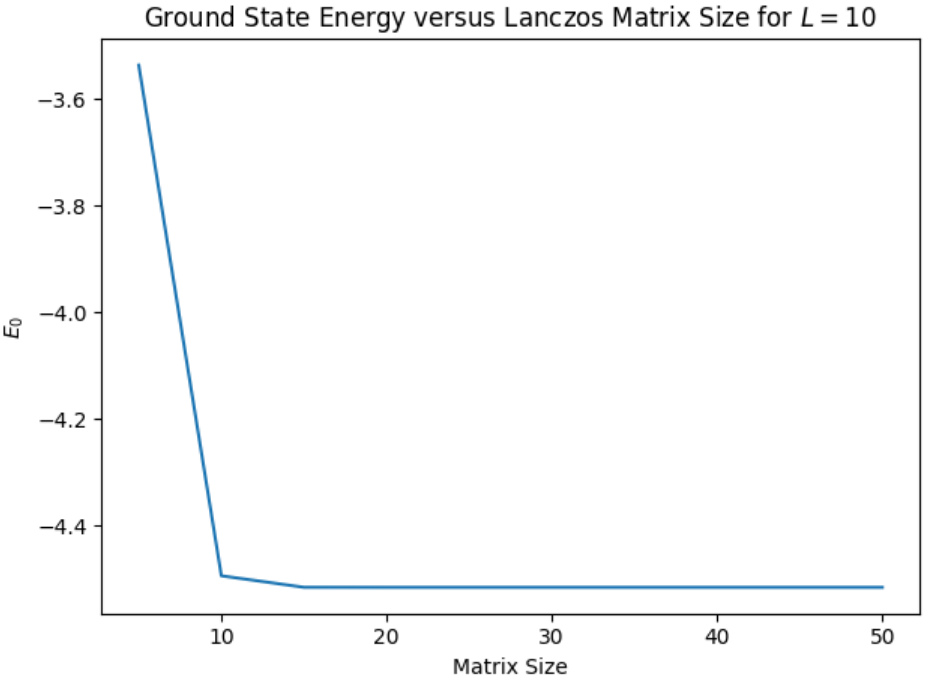
$$\langle f_i|f_i\rangle = \langle\omega|\omega\rangle - \alpha_i^2$$

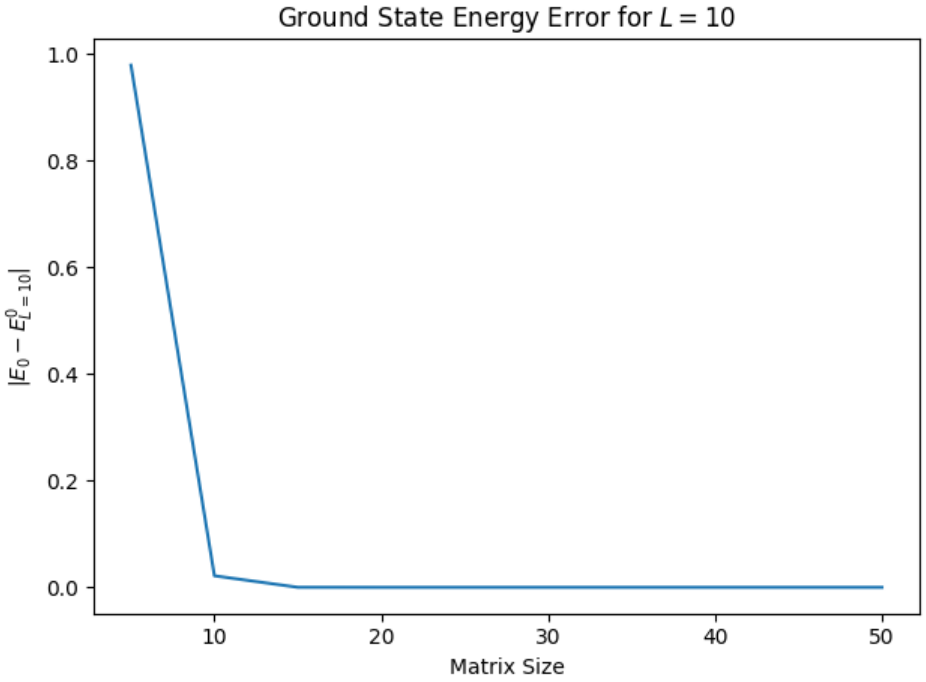
With α and β computed we can create our matrix

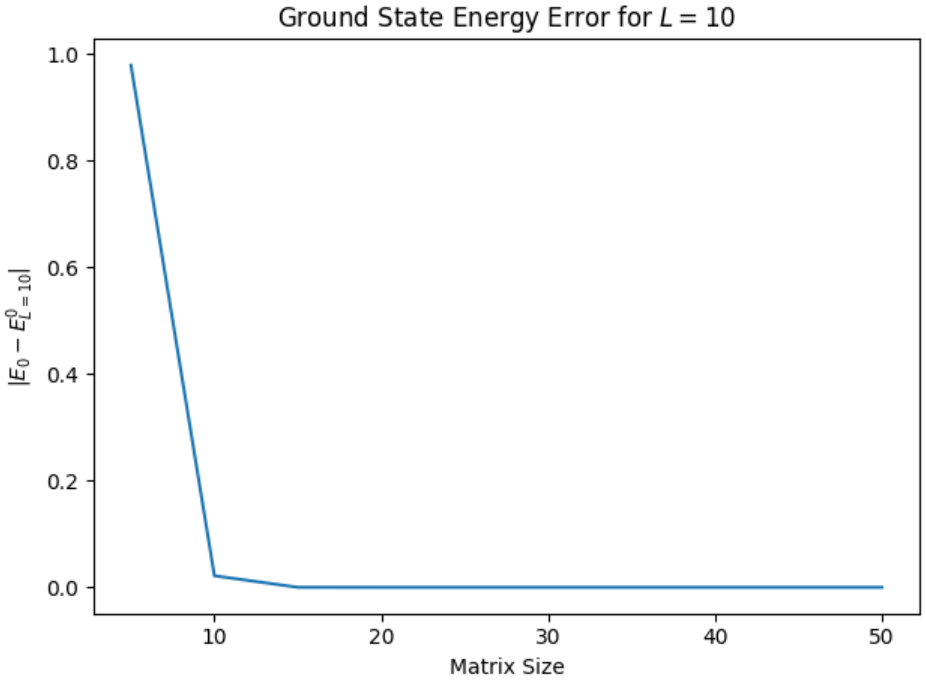
```
    for (int i = 0; i < m; i++){
        if ((i>0)&&(i<m-1)){
            Lan[i][i-1] = beta[i-1];
            Lan[i][i] = alpha[i];
            Lan[i][i+1] = beta[i];
        } else if (i == 0){
            Lan[i][i] = alpha[i];
            Lan[i][i+1] = beta[i];
        } else {
            Lan[i][i-1] = beta[i-1];
            Lan[i][i] = alpha[i];
        }
    }
```

This goes through each row and inserts the α and β as appropriate. With this we can now find the energies for systems of various sizes. We first look at the convergence of these methods for $L = 10$ and $L = 14$.

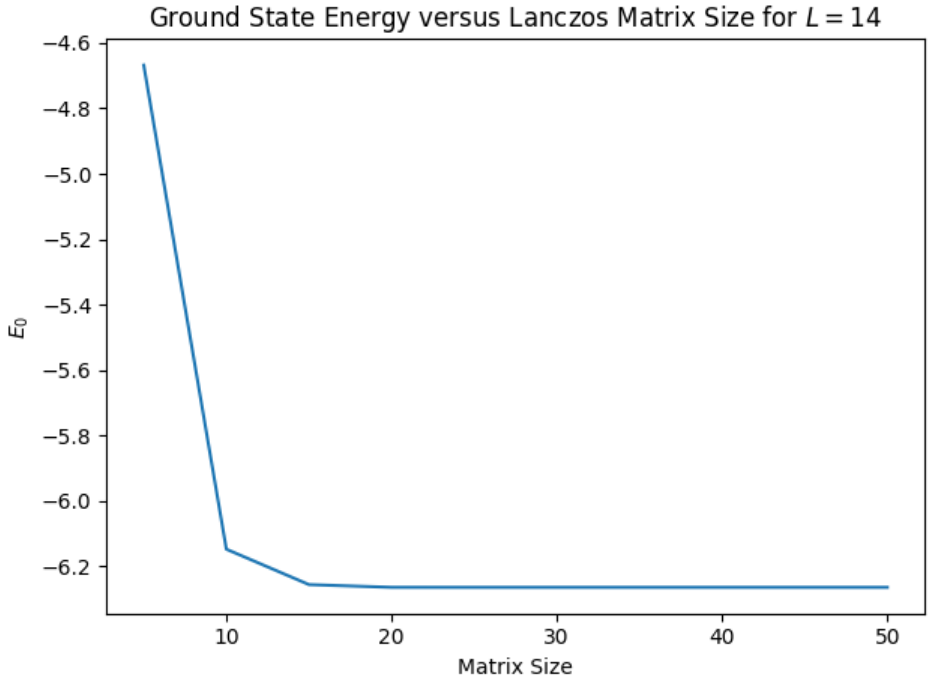
For $L = 10$ we present E_0, E_1 and the error $E_0 - E^0_{L=10}$ as functions of matrix size M.

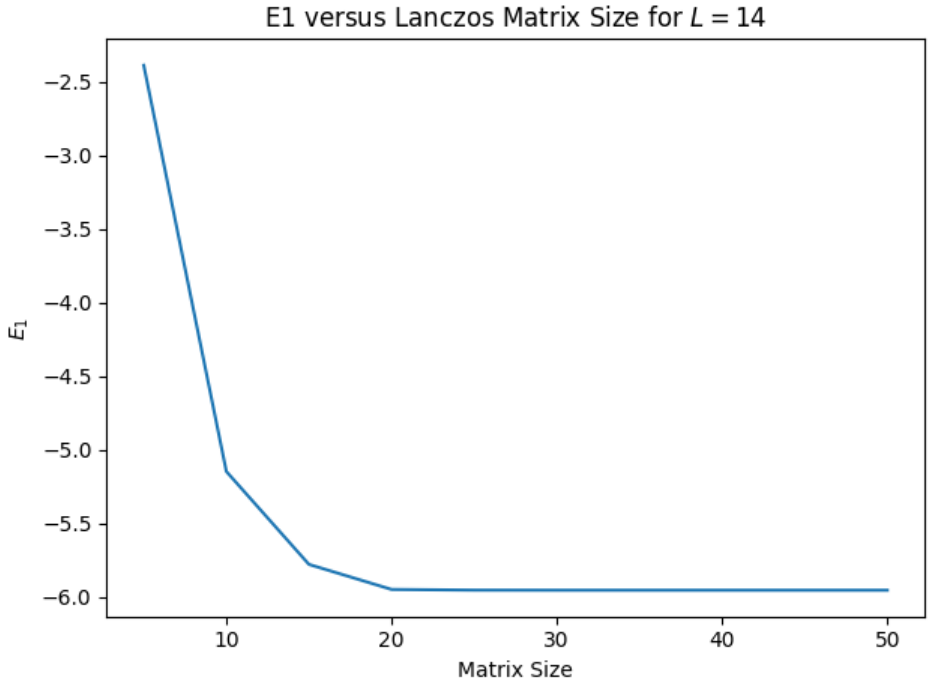

Ground State Energy versus Lanczos Matrix Size for $L = 10$
E_0
−3.6
−3.8
−4.0
−4.2
−4.4
10
20
30
40
50
Matrix Size


Ground State Energy Error for $L = 10$
$|E_0 - E^0_{L=10}|$
1.0
0.8
0.6
0.4
0.2
0.0
10
20
30
40
50
Matrix Size

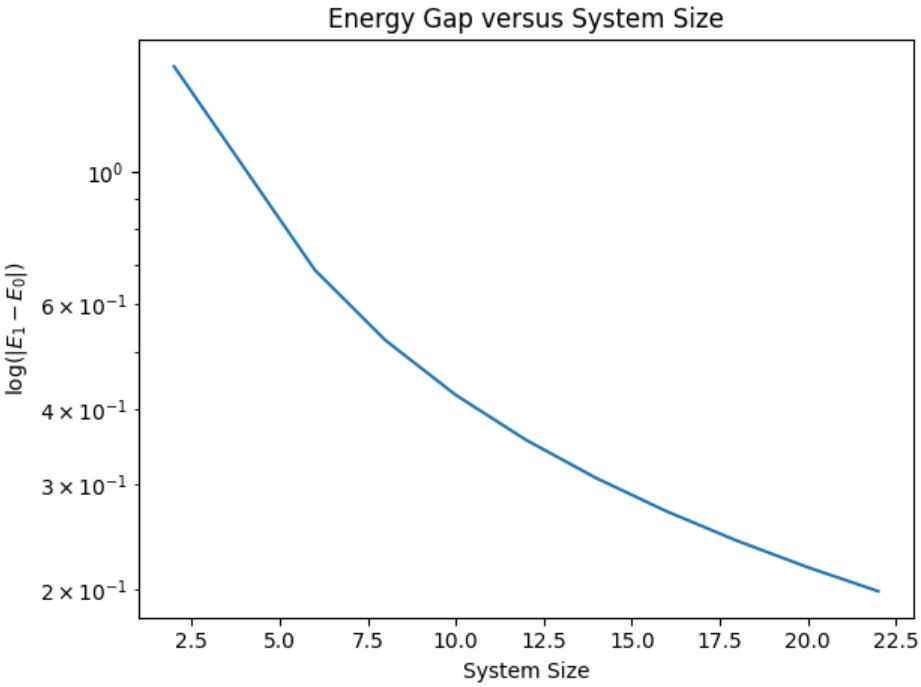


We can see steady convergence to the actual value quite quickly as M increases. By $M = 10$ the functions are essentially flat. For $L = 14$ we see similar things


Ground State Energy versus Lanczos Matrix Size for $L = 14$
E_0
−4.6
−4.8
−5.0
−5.2
−5.4
−5.6
−5.8
−6.0
−6.2
10
20
30
40
50
Matrix Size


E1 versus Lanczos Matrix Size for $L = 14$
E_1
−2.5
−3.0
−3.5
−4.0
−4.5
−5.0
−5.5
−6.0
10
20
30
40
50
Matrix Size

Again we see quick convergence of the energy values, although not as fast as $L = 10$, indicating that larger systems need larger matrices. We now look at the energy gap $|E_1 - E_0|$ as a function of system size. Plotting the values logarithmically we obtain the following graph.



Notice that a negative exponential is a straight line of negative slope in a log plot. Since our graph is curved that means its decaying faster than an exponential, which would indicate it will converge to some small but positive value as an exponential is one of the fastest ways to converge to 0. Thus if we converge faster than an exponential it suggests it converges too fast to even reach 0.