Using Python to Find the Spectrum of Eigenvalues for 1D Half Filling of JCM-like Quantum Configurations

Jeremy Schiff

September 2020

Abstract

A Python program was designed to find and plot the spectrum of eigenvalues for 1D half filling of JCM-like quantum configurations. This is useful as it can be used as a visual representation of the level crossings between energy states, and as a simple exact test case for more general quantum simulators.

1 Introduction

Quantum simulation is a relatively new branch of quantum mechanics, seeking to create accurate simulations of the quantum world. For this to be done properly, it is theorised that a quantum computer is necessary, but for simple cases, classical computers will do. One such case is half filling of a 1D chain of particles. In such a case, using some mathematical tricks, it is in fact possible to ignore the quantum nature of the system, and instead analyse it using combinatorics and iteration. This is done by considering binary number representations of the states; each 1 corresponding to the presence of a fermion at that position, and each 0 representing an empty site. The purpose of this paper is to explain the logic behind, implementation of, and way to use, the Python code in question. Thus most of the explanations and proofs of these methods are beyond the scope of this paper, and thus are omitted. [1]

2 How to Use the Program

The program is quite straightforward to use, following the following steps;

1. Set the number of available sites, n, on line 7.
   (a) Half filling will automatically be calculated.

[1] For more information on this, and for many of the full proofs, see [1]

(b) If the number of sites is odd, there is no meaningful half filling, and thus the program will end with an error message.

2. Of the three variables, $g, Mand\epsilon$, two must be set to be constant, while the third may vary.

3. To do this, choose which of the three will vary by un-commenting 1 of lines 19-22 (to vary ϵ), or 25-28 (to vary M), or 31-34 (to vary g). The other two must be left commented out.

4. Use lines 11, 12 and 13 to define the range of variation of your chosen variable. Choose the minimum and maximum values it shall take, and the number of sample points in between to be calculated.

5. Run the program to obtain the span of the eigen values

3 The Program Functions

3.1 Generating the Allowed States

The first thing to do is find all the configurations of fermions which yield allowed states. By using binary as the ordering convention, we can generate *all* the possible states by simply counting in binary from 0 to 2^n. This range of possibilities can be limited more; the minimum half filling, as a binary number would be $n/2$ 0's followed by $n/2$ 1's, and the visa versa for the maximum. Converting these binary numbers to base 10, we find that the minimum and maximum states, respectively are $|2^{n/2}\rangle$ and $|2^n - 2^{n/2}\rangle$, and thus these are set as the range of our generated states. Then, the tuple 'binaryNumber' is created which consists of each state's number (in base 10) and the binary in list form. Clearly not all of these states are half fillings, so we filter out all the non-half-filling states by checking that the sum of the elements of the lists of binary digits add to $n/2$. The decimal numbers of the allowed states are recored in the 'allowedStates' list, and the dictionary 'stateLib' stores the conversion to binary.

3.2 Qfinder

The 'Qfinder' function take in a state, and uses the formula

$$\hat{Q_x} = \hat{\Psi}_x^\dagger \hat{\Psi}_x - t(x) \tag{1}$$

where

$$f(x) = \begin{cases} 0, & \text{if x is odd} \\ 1, & \text{if x is even} \end{cases} \tag{2}$$

to find the charge at each site [1].

Noting the fact that the LHS of (1) is the definition of the number operator, creating a library of the charges at each site for a given state is done.

3.3 Efinder

This function is constructed similarly to the previous. The electric field on each link can be defined as the sum of all the charges on sites to the left of a given link [1]. Thus, we itteративly add all the values found by 'Qfinder' and add them to a new dictionary of the values of E at each link, for a given state.

3.4 DiagFinder

Recall that the Hamiltonian of a state $|\phi\rangle$ is given by

$$\langle\phi|\,\hat{H}\,|\phi\rangle \tag{3}$$

the equation for the Hamiltonian operator;

$$\hat{H} = \hat{H}_{bosons} + \hat{H}_{fermions} + \hat{H}_{interaction} \tag{4}$$

$$\hat{H}_{bosons} = \frac{1}{2}g^2 \sum_x \hat{E}_x^2 \tag{5}$$

$$\hat{H}_{fermions} = M \sum_x (-1)^x \hat{n}_x \tag{6}$$

where $\hat{U}$ is the raising operator for electric field [1].

$$\hat{H}_{interaction} = \epsilon \sum_x [\hat{\Psi}_x^\dagger \hat{U}_x \hat{\Psi}_{x+1} + \hat{\Psi}_{x+1}^\dagger \hat{U}_x^\dagger \hat{\Psi}_x] \tag{7}$$

Equations (4) and (5) both act on only one state, and thus the diaganal elements of the Hamiltonian matrix will only depend on them. Therefore, the 'DiagFinder' function computes $\hat{H}_{bosons}$ and $\hat{H}_{fermions}$ for each state, and adds them together to form the diagonal of the Hamiltonian matrix.

3.5 CoupleFinder

Clearly, the interaction term of the Hamiltonian depends on the interaction between pairs of states. This interaction term has the result of moving a single fermion one site to the left or right. Therefore, we can find all the states that are 'paired' together, by finding which of the 'allowedStates' can move a single 1, one space over to form another allowed state. Since we know that the Hamiltonian must be hermition [1], only one type of motion must be considered. Here, left motion has been chosen. Considering the nature of binary numbers, moving a single 1, one place to the left would increace the decimal form of the number by 2^p, where p is the number of places to the right of the 1 in question. Therefore, we can simply take each allowed state (in its base 10 number representation) and add 2^a [2]. If the resulting number is also the base 10 number representation of an allowed state, the two states must be paired by the interaction term. Once the pairs are found, all that is left is to record them in a list, aptly called 'pairs'.

[2] Here, range of a's checked has been taken to be the full range of allowed states.

3.6 OffDiagFinder

Once the paired states are found, the off diagonal elements of the Hamiltonian matrix can be set to ϵ. The row and column positions of these are the number of the allowed states they pair. We know that all the ϵ are positive due to proof 4.1.

3.7 The Start Functions

Each of the three start functions is slightly different, depending on which variable is changing, but use the same principal. First, run 'Qfinder' and 'Efinder' for all the allowed states. Then loop through whichever function depends on the viable, and find the eigenvalue of the Hamiltonian produced each time. Each time, sort the eigenvalues in order, then record them in the list 'eigValues'. Finally, plot a graph of the sorted eigenvalues against the changing variable. Which Start function to run is chosen by a simple if statement, depending on which variable the user has chosen.

4 Proofs

As stated in the introduction, most proofs have been omitted from this paper, however, there is one included in [1] and will thus be explained here.

4.1 Proof that all ϵ are Positive

1. Consider the interaction term acting to produce left-ward motion

$$\Psi_x^\dagger \Psi_{(x+1)} \tag{8}$$

on a generala fermionic state

$$\Psi_{y_1}^\dagger \Psi_{y_2}^\dagger \Psi_{y_3}^\dagger ... \Psi_{y_L}^\dagger \left|vac\right\rangle_f \tag{9}$$

Using the ordering convention

$$y_1 < y_2 < y_3 < ... < y_L \tag{10}$$

Where $L = N/2$ due to half filling.

2. The result of this action $\neq 0$ if and only if the destruction operator, $\Psi_{x+1}^\dagger$, annihilates one of the occupied sites. Thus;

$$x + 1 = y_m \tag{11}$$

for $1 \leq m \leq L$

3. This can be split into two separate cases, where m is even or m is odd

(a) Consider the even sites; $m = 2n$

$$x + 1 = y_{2n} \tag{12}$$

i. Applying the left motion to this case, we have

$$\Psi^\dagger_{y_{2n}-1}\Psi_{y_{2n}}\Psi^\dagger_{y_1}\Psi^\dagger_{y_2}\Psi^\dagger_{y_3}...\Psi^\dagger_{y_{2n}}...\Psi^\dagger_{y_L}\left|vac\right\rangle_f \tag{13}$$

ii. Now we must fix the ordering convention, so the first two terms must both be moved (together), until $\Psi_{y_{2n}}$ is immediately to the left of $\Psi^\dagger_{y_{2n}}$, to allow for annihilation.

(b) Now consider the odd site case; $m = 2n - 1$

i. Applying the left motion to this case, we have

$$\Psi^\dagger_{y_{2n-1}-1}\Psi_{y_{2n-1}}\Psi^\dagger_{y_1}\Psi^\dagger_{y_2}\Psi^\dagger_{y_3}...\Psi^\dagger_{y_{2n-1}}...\Psi^\dagger_{y_L}\left|vac\right\rangle_f \tag{14}$$

ii. Again, we move the first two terms are moved together so that $\Psi_{y_{2n-1}}$ can annihilate $\Psi^\dagger_{y_{2n-1}}$, while the first term is placed in the correct position for our ordering convention.

In both cases the motion of operators occurs in *pairs* of operators, so the total number of shifts is even so the sign does not change. Thus, in all cases of left motion, the expression is positive and so ϵ is positive.

4. This holds true for rightward motion, caused by

$$\Psi^\dagger_{x+1}\Psi_x \tag{15}$$

as well.

5 Conclusion

A python program was created to find the eigenvalue spanning for half filling of 1D chains of fermions, with the aim of finding level crossings. Further work could extend this to higher dimensions, as well as configurations other than half filling.

References

[1] M. Lewenstein F. Jendrzejewski E. Zohar V. Kasper, G. Juzeliunas. From the jaynes-cummings model to non-abelian gauge theories: a guided tour for the quantum engineer.

A Example Figures

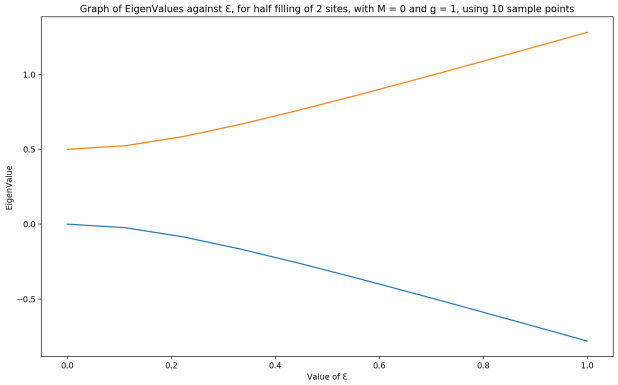

Figure 1: Example of eigen value span

.

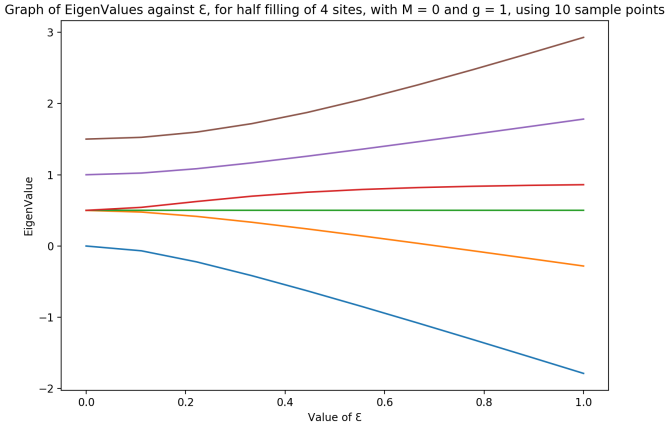

Figure 2: Example of eigen value span

.

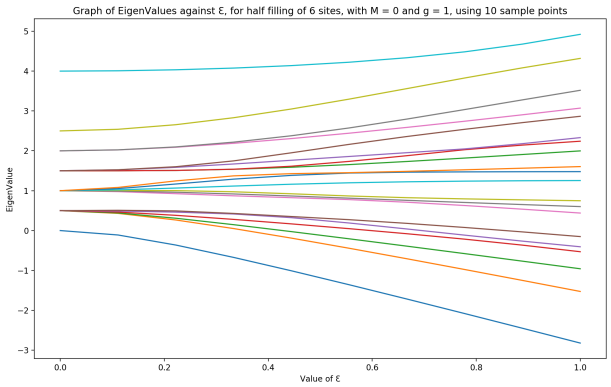

Figure 3: Example of eigen value span

.

B Code

Listing 1: SpanOfEigenValues.py the Core Functions

```
from numpy import zeros, linalg, linspace, array, shape
from sys import exit # Terminates the code, used if n is odd
import matplotlib.pyplot as plt
from time import time # Allows run time to be found
start_time = time() # records time at start

n = 4 #number of sights
if n%2 != 0:
    print('number of sites is not even, no meaningful half filling')
    exit()

MinValue = 0
MaxValue = 1
NumOfPoints = 10 # Number of sample points
variable = []

'''Choose a variable - only one!!! - leave the other two commented out'''

'''Vary   '''
variable.append('epsilon')
g = 1
M = 0
epsilonArray = linspace(MinValue, MaxValue, NumOfPoints)

'''Vary M'''
# variable.append('M')
# g = 1
# MArray = linspace(MinValue, MaxValue, NumOfPoints)
# epsilon = 1

'''Vary g'''
# variable.append('g')
# gArray = linspace(MinValue, MaxValue, NumOfPoints)
# M = 0
# epsilon = 1

eigValues = []
allowedStates = [] #list of allowed states
stateLib = {} #tells you what the binary representation of each state is
stateQlib = {} #tells you what the Q at each site is, for each state
stateElib = {} #tells you what the E at each link is, for each state
pairs = [] # Will be a list containing the connected pair states
```

```
#Generates binary numbers from 2^(n/2)-1 up to 2^n - 2^(n/2),
#(min and max half filling) all of length n
for i in range(2**(int(n/2))-1,2**n - int(2**(n/2)) +1):
    # Creates tuple, the first element is the number of the state,
    # the second is the binary representation in list form
    binaryNumber = i, [int(k) for k in "{0:0{1}b}".format(i, n)]
    if sum(binaryNumber[1]) == n/2: #Adds all 1/2 filling states to the list
                    # of allowedStates, and defines what they are in stateLib
        allowedStates.append(binaryNumber[0])
        newState = {binaryNumber[0]:binaryNumber[1]}
        stateLib.update(newState)

N = len(allowedStates) #number os states
H = zeros(shape=(N,N)) # Hamiltonian matrix, for now entirely made up of 0s
def Qfinder(state): #generates a list of the Q values at each site,
                    #for a given state, then attatches that
                    #list to the state in stateQlib
    Qlist = []
    for i in range(n):
        if i%2 == 0:
            Qlist.append(stateLib.get(state)[i])
        else:
            Qlist.append(stateLib.get(state)[i]-1)
    stateQlib.update({state:Qlist})

#generates a list of E values at each link, for a given state,
#then attatches that list to the state in stateElib
def Efinder(state):
    Elist = []
    for i in range(n-1):
        newE = stateQlib.get(state)[i]
        if i != 0:
            newE += Elist[i-1]
        Elist.append(newE)
    stateElib.update({state:Elist})

def DiagFinder(M,g): # Generates diagonal elements of H
    for i,state in zip(range(N), allowedStates):
        Hf = M * sum([(-1) ** i * stateLib.get(state)[i] \
            for i in range(len(stateLib.get(state)))])
        # Hb = 1/2 g * the sum of E^2 on each link for a given state
        Hb = g**2 * 1/2 * sum([stateElib.get(state)[i]**2 \
            for i in range(len(stateElib.get(state)))])
        H[i][i] = Hb + Hf
```

```
# Finds coupled states via left motion of fermion, adds them to the list 'pairs'
def CoupleFinder():
    for state in allowedStates:
        for a in range(2**n - 2*int(2**(n/2)) +1):
            if state + 2**a in allowedStates:
                pairs.append([state,state+2**a])

def OffDiagFinder(epsilon): # Finds off diagonal elemenst
    # Goes through the list pairs, finds the number of the paired states,
    # and sets that element of H to e
    for p in pairs:
        H[allowedStates.index(p[0])][allowedStates.index(p[1])] = epsilon
        H[allowedStates.index(p[1])][allowedStates.index(p[0])] = epsilon

def Start1(): # Runs the program for variable epsilon
    for state in allowedStates: # Runs Qfinder and Efinder for all states
        Qfinder(state)
        Efinder(state)

    DiagFinder(M,g)
    CoupleFinder()
    for epsilon in epsilonArray: # Runs OffDiagFinder for diferent values of
                    #epsilon and finds the eigenvalues of each H that creates
        OffDiagFinder(epsilon)
        eigValue = linalg.eig(H)[0] # The eigenvalues of H
        # eigVectors  = linalg.eig(H)[1] # The eigenvectors of H
        # List of eigenvalues of H, 1 sublist per value of epsilon
        eigValues.append(sorted(eigValue))

    for i in range(len(eigValues[0])):
        plt.plot(epsilonArray, array(eigValues)[:,i], label=f'  {i+1}')
    plt.title(f'Graph of EigenValues against   , for half filling of {n} sites,
        with M = {M} and g = {g}, using {NumOfPoints} sample points')
    plt.xlabel('Value of   ')
    plt.ylabel('EigenValue')
    # plt.legend()
    # plt.xlim(0.0233333,0.0233334)
    plt.show()

def Start2(): # Runs the program for variable M
    for state in allowedStates: # Runs Qfinder and Efinder for all states
        Qfinder(state)
        Efinder(state)

    CoupleFinder()
```

```
    OffDiagFinder(epsilon)
    for M in MArray:
        DiagFinder(M,g)
        eigValue = linalg.eig(H)[0] # The eigenvalues of H
        # eigVectors  = linalg.eig(H)[1] # The eigenvectors of H
        eigValues.append(sorted(eigValue)) # List of eigenvalues of H,
                                           #1 sublist per value of epsilon

    for i in range(len(eigValues[0])):
        plt.plot(MArray, array(eigValues)[:,i], label=f'  {i+1}')
    plt.title(f'Graph of EigenValues against M, for half filling of {n} sites, \
        with    = {epsilon} and g = {g}, using {NumOfPoints} sample points')
    plt.xlabel('Value of M')
    plt.ylabel('EigenValue')
    # plt.legend()
    # plt.xlim(0.0233333,0.0233334)
    plt.show()

def Start3(): # Runs the program for variable g
    for state in allowedStates: # Runs Qfinder and Efinder for all states
        Qfinder(state)
        Efinder(state)

    CoupleFinder()
    OffDiagFinder(epsilon)
    for g in gArray:
        DiagFinder(M,g)
        eigValue = linalg.eig(H)[0] # The eigenvalues of H
        # eigVectors  = linalg.eig(H)[1] # The eigenvectors of H
        eigValues.append(sorted(eigValue)) # List of eigenvalues of H,
                                           #1 sublist per value of epsilon

    for i in range(len(eigValues[0])):
        plt.plot(gArray, array(eigValues)[:,i], label=f'  {i+1}')
    plt.title(f'Graph of EigenValues against g, for half filling of {n} sites, \
        with    = {epsilon} and M = {M}, using {NumOfPoints} sample points')
    plt.xlabel('Value of g')
    plt.ylabel('EigenValue')
    # plt.legend()
    # plt.xlim(0.0233333,0.0233334)
    plt.show()


if len(variable) == 1:
    if variable == ['epsilon']:
        Start1()
```

```
    elif variable == ['M']:
        Start2()
    elif variable == ['g']:
        Start3()

elif len(variable) >1:
    print('Error: Too many variables, comment out all but 1, see line 15')
else:
    print('Error: No variables set, comment out all but 1, see line 15')
```