# Quantum Many Body Physics – Final Project

1. Number of basis states:

$$N=3, M=2 \rightarrow |30\rangle, |03\rangle, |21\rangle, |12\rangle$$
$$N=2, M=3 \rightarrow |002\rangle, |020\rangle, |200\rangle, |110\rangle, |101\rangle, |011\rangle$$

In general, there are N+M-1 positions. We want to fill N of these positions, hence:

$$\binom{N+M-1}{N} = C(N+M-1, N) = \frac{(N+M-1)!}{N!(M-1)!}$$

2. Renyi Entropy:

$$|\psi(0)\rangle = |11\rangle = (0,1,0), |\psi(t)\rangle = \exp\left[\frac{-it\hat{H}}{\hbar}\right]|\psi(0)\rangle$$

$\hat{H}$ in matrix form from basis states:

$$\hat{H}|02\rangle = -J|11\rangle + U|02\rangle$$
$$\hat{H}|11\rangle = -J|02\rangle - J|20\rangle$$
$$\hat{H}|20\rangle = -J|11\rangle + U|20\rangle$$
$$\hat{H} = \begin{bmatrix} U & -J & 0 \\ -J & 0 & -J \\ 0 & -J & U \end{bmatrix}$$

Setting U = J = 1:

$$|\hat{H} - \mathbb{I}E| = \begin{vmatrix} 1-E & -1 & 0 \\ -1 & -E & -1 \\ 0 & -1 & 1-E \end{vmatrix} = 0$$
$$\rightarrow (1-E)(-E+E^2) + E - 1 + E - 1 = (1-E)(-E+E^2-2) = 0$$
$$\rightarrow E_1 = 1, \; E_2 = 2, \; E_3 = -1$$

This leads to eigenstates:

$$\bar{\psi_1} = \begin{pmatrix} \frac{-1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix}, \bar{\psi_2} = \begin{pmatrix} \frac{1}{\sqrt{6}} \\ \frac{2}{\sqrt{6}} \\ \frac{1}{\sqrt{6}} \end{pmatrix}, \bar{\psi_3} = \begin{pmatrix} \frac{1}{\sqrt{3}} \\ \frac{-1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{pmatrix}$$

And a time evolution:

$$|\psi(t)\rangle = \sum_i exp\left(\frac{-iE_i t}{\hbar}\right)|\bar{\psi_i}\rangle\langle\bar{\psi_i}|\psi(0)\rangle$$

$$= e^{\frac{-it}{\hbar}} \begin{pmatrix} \frac{-1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} \frac{-1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + e^{\frac{it}{\hbar}} \begin{pmatrix} \frac{1}{\sqrt{6}} \\ \frac{2}{\sqrt{6}} \\ \frac{1}{\sqrt{6}} \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{6}} & \frac{2}{\sqrt{6}} & \frac{1}{\sqrt{6}} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + e^{\frac{-2it}{\hbar}} \begin{pmatrix} \frac{1}{\sqrt{3}} \\ \frac{-1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{3}} & \frac{-1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

$$= \frac{1}{3} \begin{pmatrix} e^{\frac{it}{\hbar}} + e^{\frac{-2it}{\hbar}} \\ 2e^{\frac{it}{\hbar}} - e^{\frac{-2it}{\hbar}} \\ e^{\frac{it}{\hbar}} + e^{\frac{-2it}{\hbar}} \end{pmatrix}$$

This leads to the reduced density matrix:

$$\hat{\rho}_A = \frac{1}{9} \begin{pmatrix} (e^{it/\hbar} + e^{-2it/\hbar})(e^{-it/\hbar} + e^{2it/\hbar}) & 0 & 0 \\ 0 & (2e^{it/\hbar} - e^{-2it/\hbar})(2e^{-it/\hbar} - e^{2it/\hbar}) & 0 \\ 0 & 0 & (e^{it/\hbar} + e^{-2it/\hbar})(e^{-it/\hbar} + e^{2it/\hbar}) \end{pmatrix}$$

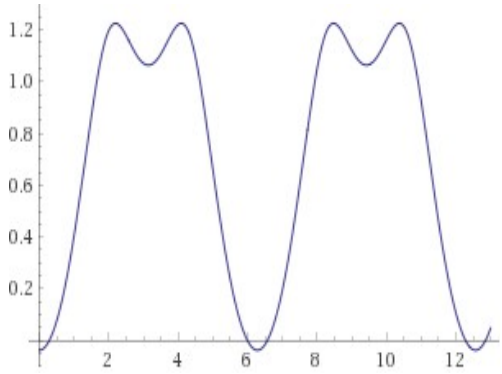
And a squared reduced density matrix:

$$\hat{\rho_A}^2 = \frac{1}{81} \begin{pmatrix} [2 + 2cosh(3it/\hbar)]^2 & 0 & 0 \\ 0 & [5 - 4cosh(3it/\hbar)]^2 & 0 \\ 0 & 0 & [2 + 2cosh(3it/\hbar)]^2 \end{pmatrix}$$

Calculating the Renyi entropy as $S(t) = -\ln(Tr(\rho_A^2(t)))$ :

$S(t) = -ln(\frac{1}{81}) - ln\left[24cos^2(3t/\hbar) + 28cos(3t/\hbar) + 32\right]$

Which looks like:




3. Simulation:

Simulation of the Bose-Hubbard model for a system of 6 bosons, equally bipartitioned.

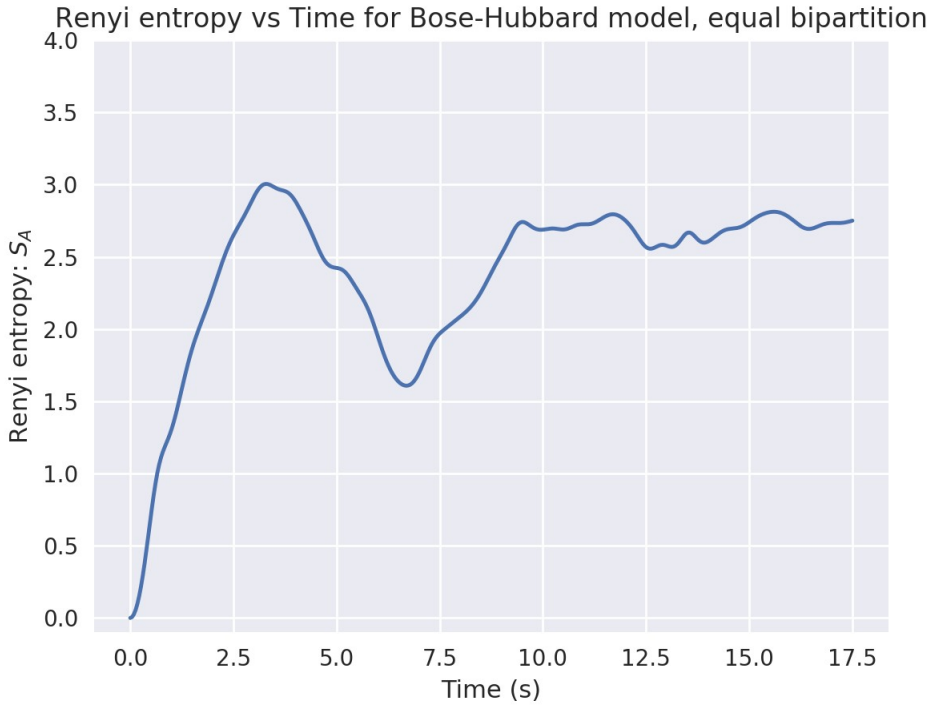
The parameters are J = 0.64, U = 1, N=M=6

The initial state is: $|111111\rangle$

For this system, the 4 lowest energies are:

| | | | |
|---|---|---|---|
| -4.91927368 | -4.39422147 | -4.00006934 | -3.76881561 |

The resultant plot for Renyi entropy over time is:




See below for code. Comments. Docstrings & multi-line text. Classes & functions. Builtins & keywords.

```
# -*- coding: utf-8 -*-
"""
```

```
author: Sam Spillard
python script to complete final project of Quantum Many-Body Physics module
Simulate experiment by Kaufman et al. 2016

Python version: 3.5.4
"""

from scipy.special import factorial as fact
import scipy as sp
import numpy as np
import itertools
from copy import deepcopy
import matplotlib.pyplot as plt
#import seaborn as sns
#sns.set()

class tColors:
    """
    Color class for debugging
    """
    WARN = '\x1b[93m'
    FAIL = '\x1b[91m'
    ENDC = '\x1b[0m'

class StateObj:
    """
    Class to keep track of state vector, prefactor and type
    idx is a parameter that keeps basis states in the correct order according
    to the integer representation of the vector.
    Includes creation, annihilation and number operators, deepcopy'd to prevent
    modifying basis states.
    """
    # Initialise attributes
    def __init__(self, init_vec, idx, _type):
        if(type(init_vec) != np.ndarray):
            raise TypeError('init_vec should be a numpy.ndarray')
        self.vector = init_vec
        self.idx = idx
        self.type = _type
        self.prefactor = 1
    # Creation operator
    def create(self, index, N):
        trans = deepcopy(state)
        if(trans.vector[index] == N):
            trans.prefactor = 0
        else:
            trans.vector[index] += 1
        return trans
    # Annihilation operator
    def destroy(self, index):
        trans = deepcopy(state)
        if(trans.vector[index] == 0):
            trans.prefactor = 0
        else:
            trans.vector[index] -= 1
        return trans
    # Number operator
    def num(self, index):
        if(self.vector[index] == 0):
            return -1
        return 1

def getBasisStates(N, M):
    """
    Function to get a list of basis states, ordered by the integer
    representation of the state vector.
    """
    x = itertools.product(range(N + 1), repeat=M)
    basis = [np.asarray(i, dtype=int) for i in x if np.sum(i)==N]
```

```
    states = np.asarray([StateObj(basis[i], i, 'boson') for i in
                         range(len(basis))])

    return states

def actHam(state, N, J, U):
    """
    Act hamiltonian on a state, as a series of Creation, Annihilation and
    Number operators. Copy states so that the original state is not effected.
    Multiply by appropriate parameters, J, U.
    Returns an array of new states with appropriate prefactors.
    """
    t1, t2 = [], []
    # First term
    for i in range(len(state.vector)-1):
        t1.append(state.create(i+1, N).destroy(i))
        t1.append(state.create(i, N).destroy(i+1))
    # Second term
    for i in range(len(state.vector)):
        PREFACTOR = (state.num(i)**2) - state.num(i)
        temp3 = deepcopy(state)
        temp3.prefactor *= (PREFACTOR*U/2)
        t2.append(temp3)

    for state in t1:
        state.prefactor *= (-1 * J)

    return np.r_[t1, t2]

def getHamMatrix(N, M, J, U):
    """
    Get hamiltonian matrix by acting hamiltonian on each basis state.
    """
    basis = getBasisStates(N, M)
    ham_matrix = np.zeros((len(basis), len(basis)))
    for state in basis:
        # Act ham on each basis state
        acted = actHam(state, N, J, U)
        for x in acted:
            for b in basis:
                # Find the matrix location of the 'acted' state and enter into
                # Hamiltonian matrix.
                if(np.all(x.vector == b.vector)):
                    ham_matrix[state.idx][b.idx] += x.prefactor
    # Return ham matrix and basis
    return ham_matrix, basis

def getInitialState(N, M, count=0, default=False):
    """
    Get initial state vector from user.
    """
    LENGTH = int((fact(N+M-1))/(fact(N)*fact(M-1)))
    PREAMBLE = """Enter initial state configuration (e.g. "1, 0, 0").
Note: This is the normalised vector representing the superposition of basis
states.
The basis states are ordered in ascending order of their integer representation
e.g. the state (0, 2) -> "2" will be [1, 0, 0], the state (1, 1) -> "11" will
be [0, 1, 0] and the state (2, 0) -> "20" will be [0, 0, 1].
Should have length: C(N+M-1, N) = """ + str(LENGTH) + ': '
    if(default):
        idx = int(LENGTH/2)
        initialStateVec = np.zeros(LENGTH)
        initialStateVec[idx] = 1
        state = StateObj(initialStateVec, None, 'state')

    else:
        initialStateVec = np.asarray([float(x) for x in input(PREAMBLE)
            .split(', ')])
        if(len(initialStateVec) != LENGTH or np.sum(np.square(
```

```
                initialStateVec)) != 1.):
            print(tColors.FAIL + '\nLength of initial state should be C(N+M-1,'
                               + 'N)\nState should be normalised.'
                               + tColors.ENDC)
            input('Press Enter to continue...')
            count += 1
            state = getInitialState(N,M,count=count)
        else:
            state = StateObj(initialStateVec, None, 'state')
    return state

def timeEvolve(initialState, hamMatrix, t):
    """
    Evolve a state vector to time t, according to the hamiltonian matrix.
    """
    if(initialState.type == 'boson'):
        raise TypeError('State should not be in boson format.')
    expMat = -1j * hamMatrix * t
    expMat = sp.linalg.expm(expMat)
    vNewState = np.dot(expMat, initialState.vector)
    newState = StateObj(vNewState, None, 'state')
    return newState

def getRDM(state, N, M, basis, ASIZE=1):
    """
    Get reduced density matrix of a state.
    Input state in basis vector representation e.g. [1/sqrt(2), 1/sqrt(2), 0]
    """
    # Every available a state
    a_basis = np.asarray([np.asarray(i) for i in itertools.product(range(N+1),
                          repeat=ASIZE)])
    # Every available b state
    b_basis = np.asarray([np.asarray(i) for i in itertools.product(range(N+1),
                          repeat=(M-ASIZE))])
    # Initialise c_matrix as zeros
    ALEN, BLEN = len(a_basis), len(b_basis)
    c_matrix = np.zeros((ALEN, BLEN), dtype=complex)
    for i in range(len(state.vector)):
        a_vec = basis[i].vector[:ASIZE]
        for j in range(ALEN):
            if(np.all(a_basis[j] == a_vec)):
                a_idx = j
                break
        b_vec = basis[i].vector[ASIZE:]
        for j in range(BLEN):
            if(np.all(b_basis[j] == b_vec)):
                b_idx = j
                break

        c_matrix[a_idx, b_idx] = state.vector[i]

    rdm = np.dot(c_matrix, c_matrix.conj().T)

    return c_matrix, rdm

def entropy(rdm):
    """
    Given a reduced density matrix, return the Renyi entropy
    """
    rdm2 = np.dot(rdm, rdm)
    vals, vecs = sp.linalg.eigh(rdm2)
    _entropy = -np.log(np.sum(vals))
    return _entropy

def getPlot(initDict, tArr):
    """
    Given dictionary of initial parameters and an array of times to evaluate
    at, calculate the Renyi entropy and plot.
    """
```

```
    entropy_arr = []
    for t in tArr:
        print('Time: ' + str(t))
        tState = timeEvolve(initDict['initState'], initDict['ham'], t)
        cmat, rdm = getRDM(tState, initDict['N'], initDict['M'],
                           initDict['basis'], initDict['ASIZE'])
        tEntropy = entropy(rdm)
        entropy_arr.append(tEntropy)
    plt.plot(tArr, entropy_arr)
    plt.xlabel('Time (s)')
    plt.ylabel('Renyi entropy: $S_{A}$')
    plt.title('Renyi entropy vs Time for Bose-Hubbard model, equal bipartition')
    #plt.savefig('plot.png', format='png', dpi=100)
    plt.show()
    return

def init():
    """
    Initialisation. Gets parameters from user and construct Hamiltonian matrix.
    Returns dict of params.
    """
    N, M, J, U = [float(x) for x in input(
            'Enter params (comma separated: "N, M, J, U"): ').split(', ')]
    N, M = int(N), int(M)
    default = bool(int(input('Use default initial state? (1, 1, 1...)'
                             + '(yes:1/no:0): ')))
    initialState = getInitialState(N, M, default=default)
    hamMatrix, basis = getHamMatrix(N, M, J, U)
    if(bool(int(input('Print 4 lowest energies? (yes:1, no:0): ')))):
        vals, vecs = sp.linalg.eigh(hamMatrix)
        print(vals[:4])
    ASIZE = int(input('Enter size of A subsystem: '))
    return {'N': N, 'M': M, 'J': J, 'U': U, 'initState': initialState,
            'ham': hamMatrix, 'basis': basis, 'ASIZE': ASIZE}

if(__name__ == '__main__'):
    print('In module.')
    initDict = init()
    getPlot(initDict, np.linspace(0, 18, 801))
```