1 Main Program

```
# -*- coding: utf-8 -*-
"""
Created on Sun Nov 14 10:32:42 2021
last edited on 14/12/2021
@Author: Liam W Bussey
Affiliation@ BT and University of Birmingham

This code will model the steady state and transient
evolution of a Rydberg reciever using ladder
configuration for atomic excitation.

The aim of this program is the give experimental
prediction based on user inputs from the state the user
wishes to study and the interacting field parameter to
make these tranistion.

when considering the time evolution of this system we
will study what happend when the RF/laser field is switch
on and off. the hope is to be able to effectively model a
realisitic receiver using the principle of quantum mechanics

In this code we ust the python module delevoped by durham
know as arc -Alkali-Rydberg-Calculator, which has a large
library of information relvent to the group 1 alkali metals
more information can be found at
http://pypi.org/project/ARC-Alkali-Rydberg-Calculator

To preform the steady state and transient analysis we use
the Qutip library. More information from this library can
be found at https://qutip.org/docs/latest/index.html
"""
from calendar import c
from cmath import pi
from re import I
import sys
from typing import Type
from numpy.lib.function_base import append
from qutip.qobj import Qobj
from brokenaxes import brokenaxes

sys.path.append('C:/Users/liamw/Desktop/Rydberg_Git/Github/rydberg_theory')
sys.path.append('C:/Users/liamw/Desktop/Rydberg_Git/Github/rydberg_theory/src')
sys.path.append('C:/Users/liamw/Desktop/Rydberg_Git/Github/rydberg_theory/src/
    python')

import numpy as np
    #pip install numpy
import pandas as pd
    #pip install pandas
import qutip as qp
    #pip install qutip
    #https://qutip.org/docs/latest/index.html
import scipy
from scipy import signal as sn
from scipy.interpolate import UnivariateSpline
from scipy.optimize import curve_fit, root_scalar

from scipy.constants import k as k_b
from scipy.constants import c as c_c ## speed of light
from scipy.constants import h as c_h # planck constant
from scipy.constants import hbar as hbar #reduced planks

import Transition_search as transistion
import atom_transition_wavelen as atom_wave
import EIT_to_main as EITSS  #EIT_to_main_doppler
```

```
65
66 import arc
67     #pip install ARC-Alkali-Rydberg-Calculator
68     #https://pypi.org/project/ARC-Alkali-Rydberg-Calculator/
69
70 from arc import Potassium41 as K41
71 from arc import Rubidium as Rb
72 from arc import Rubidium85 as Rb85
73 from arc import Rubidium87 as Rb87
74 from arc import Caesium as Cs
75 from arc import Strontium88 as Sr
76
77
78
79 import matplotlib.pyplot as plt
80 from mpl_toolkits.axes_grid1.inset_locator import mark_inset, zoomed_inset_axes,
       inset_axes
81 from matplotlib.transforms import IdentityTransform
82 #from matplotlib.gridspec import GridSpec
83     #https://matplotlib.org/stable/tutorials/introductory/pyplot.html
84     #conda install matplotlib
85 import liams_windows_config
86
87 #import fsv_data_extract_moku as expdata
88
89 from scipy.constants import k as k_b # boltzman constant
90 from scipy.constants import c as c_c ## speed of light
91 from scipy.constants import h as c_h # planck constant
92 from scipy.constants import hbar as hbar #reduced planks
93
94 from scipy.signal import chirp, find_peaks, peak_widths
95 ##### figure settings #########
96 plt.rcParams['font.family'] = 'serif'
97 plt.rcParams['font.size']= 20
98 plt.rcParams['figure.figsize'] = [14,12]
99 plt.rcParams['figure.autolayout']= True
100 plt.rcParams['axes.labelsize']= 18
101 plt.rcParams['axes.titlesize']= 18
102 plt.rcParams['xtick.labelsize'] = 18
103 plt.rcParams['ytick.labelsize'] = 18
104 plt.rcParams['figure.titlesize'] = 20
105
106 class laser_parameter():
107     def set_feild(self, f, OAM_list = ['L','R']):
108         while True:
109             if f != 'Unset':
110                 self.feild = str.upper(f)
111                 break
112             else:
113                 my_input = input('please enter L for laser or R for Radio field: ')
114                 try:
115                     my_input = str.upper(my_input)
116                     if (my_input not in OAM_list):
117                         print("That makes no sense... it is not in "
118                               + OAM_list)
119                         continue
120                     else:
121                         self.feild = my_input
122                         break
123                 except:
124                     print("That makes no sense...")
125                     continue
126
127     def set_temp(self, temp):
128         while True:
129             if temp != 'Unset':
130                 self.temp = float(temp)
131                 break
```

```
            else:
                my_input = input('please enter the temperature in K')

                self.temp = float(my_input)
                break

    def set_MW_feild(self,feild, vm):
        if (feild == 'R'):
            if vm != 'Unset':
                self.p_rf = float(vm)
                #print(self.p_rf)
            else:
                my_input = float(input('enter maxium field strength in V/m?: '))

                self.p_rf = my_input
        else:
            self.p_rf = np.nan

    def set_power(self, feild, laser_p):
        if (feild == 'L'):
            if laser_p != 'Unset':
                self.p = float(laser_p/1000)# laser power in W
                #print(self.p)
            else:
                my_input=float(input('Please enter laser power in mW: '))
                self.p = my_input/1000#laser power in W
        else:
            self.p = np.nan

    def set_waist(self, feild, waist):
        if (feild == 'L'):
            if waist != 'Unset':
                self.w = float(waist*10**(-6))# Micrometers
            else:
                my_input=float(input(r'Please enter laser waist in $\mu$ M: '))
                self.w  = my_input*10**(-6) #micrometers
        else:
            self.w = np.nan

    def set_q(self, q_set, OAM_list =['-1', '0', '1']):
        while True:
            if q_set != 'Unset':
                self.q = int(q_set)
                break
            else:
                my_input = input(r'please enter polaraisation q = -1,0,1(-$\sigma$
 ,$\pi$, +$\sigma$): ' )
                try:
                    if (my_input not in OAM_list):
                        print('not possible polaristion. enter value -1, 0 or 1')
                        continue
                    else:
                        self.q = int(my_input)
                        break
                except:
                    print("That makes no sense...")
                    continue

    def set_onress(self,ress, OAM_list =['Y','N']):
        while True:
            if ress != 'Unset':
                if (ress not in OAM_list):
                    print("That makes no sense... it is not in " + OAM_list)
                    break
                else:
                    self.onress = str.upper(ress)
                    break

```

```
            else:
                my_input = input('Is perfecttly on resonance [Y/N]: ')
                try:
                    my_input = str.upper(my_input)
                    if (my_input not in OAM_list):
                        print("That makes no sense... it is not in " + OAM_list)
                        continue
                    else:
                        self.onress = my_input
                        break
                except:
                    print("That makes no sense...")
                    continue

    def set_scanning_source(self, scan ,onress, OAM_list =['Y','N']):
        while True:
            if (onress == 'Y'):
                self.scanning_source = 'N'
                break
            else:
                if scan != 'Unset' and scan != 'N':
                    #print(scanning_source)
                    self.scanning_source = 'Y'
                    break
                elif scan != 'Unset' and scan != 'Y':
                    self.scanning_source = 'N'
                    break
                else:
                    my_input = input('Is this the source your scanning with? [Y/N]:
    ')
                    try:
                        my_input = str.upper(my_input)

                        if (my_input not in OAM_list):
                            print("That makes no sense... it is not in " +
                                  OAM_list)
                            continue
                        else:
                            self.scanning_source = my_input

                            break
                    except:
                        print("That makes no sense...")
                        continue

    def set_dphase_l(self, onress, deph_l):
        if (onress == 'Y'):
            self.dephase_lower = 0
        else:
            if deph_l != 'Unset':
                self.dephase_lower = float(deph_l*(10**(6))) ## MHz
                #print(self.dephase_lower)
            else:
                myinput = float(input('please enter lower dephasing value in MHz:')
    )
                self.dephase_lower = myinput*(10**(6))## MHz

    def set_dphase_u(self, onress, deph_u):
        if (onress == 'Y'):
            self.dephase_upper = 0.0
        else:
            if deph_u != 'Unset':
                self.dephase_upper = float(deph_u*(10**(6)))## MHz
            else:
                myinput = float(input('please enter upper dephasing value in MHz:')
    )
                self.dephase_upper = myinput*(10**(6))

```

```
264     def set_laser_line(self, feild, laser_line):
265         if (feild == 'L'):
266             if laser_line != 'Unset':
267                 self.laser_line = float((laser_line*10**(3)))#kHz
268             else:
269                 my_input=float(input('Please enter laser linewidth in kHz: '))
270                 self.laser_line  = (my_input*10**(3))#kHz
271         else:
272             self.laser_line = 0.0
273 
274     def set_beam_prop(self, feild, beam_prop, OAM_list = ['1', '-1']):
275         if (feild == 'L'):
276             while True:
277                 if beam_prop !='Unset':
278                     self.beam_prop = int(beam_prop)
279                     break
280                 else:
281                     my_input = input(r'if beam travels in the postive direction
    please input 1 if beam travels in counter direction please put -1 ' )
282                     try:
283                         if (my_input not in OAM_list):
284                             print('popergation direction must be enter value -1 or
    1')
285                             continue
286                         else:
287                             self.beam_prop = int(my_input)
288                             break
289                     except:
290                         print("That makes no sense...")
291                         continue
292         else:
293             self.beam_prop = 0
294 
295     def __str__(self):
296 
297         print_string = str(self.q) + str(self.p) + str(self.w) +  str(self.feild) +
    str(self.p_rf) + str(self.ress)+str(self.scan)+str(self.deph_l)+str(self.
    deph_u)+ str(self.laser_line) + str(self.temp)
298 
299         return(print_string)
300 
301 
302     def __init__(self, f = 'Unset', vm ='Unset', laser_p ='Unset', waist = 'Unset',
     q_set= 'Unset', ress= 'Unset', scan = 'Unset', deph_l ='Unset', deph_u = '
    Unset', laser_line='unset', temp = 'Unset',beam_prop = 'Unset'):
303 
304         self.feild = "unset"
305         self.onress = "unset"
306         self.dephase_lower = np.nan
307         self.dephase_upper = np.nan
308         self.dephase_step =np.nan
309         self.p_rf = np.nan
310         self.q = np.nan
311         self.p = np.nan
312         self.w = np.nan
313         self.scan = "unset"
314         self.laser_line = np.nan
315         self.temp = np.nan
316         self.beam_prop= np.nan
317 
318         self.set_feild(f)
319         self.set_MW_feild(self.feild, vm)
320         self.set_power(self.feild, laser_p)
321         self.set_waist(self.feild, waist)
322         self.set_q(q_set)
323         self.set_onress(ress)
324         self.set_dphase_l(self.onress, deph_l)
325         self.set_dphase_u(self.onress, deph_u)
```

```
        self.set_scanning_source(scan, self.onress)
        self.set_laser_line(self.feild, laser_line)
        self.set_temp(temp)
        self.set_beam_prop(self.feild, beam_prop)

class state_describing_object():

    def set_N(self, N):
        while True:

            if (N != 'Unset'):
                self.energy_level = int(N)
                break
            else:
                my_input = input("Atomic Energy Level (N) [Integer value]:")

                try:

                    my_input = int(my_input)

                    if (type(my_input) != type(1) or my_input <=0 ):
                        print("That makes no sense...")
                        continue
                    else:
                        self.energy_level = my_input
                        break
                except:
                    print("That makes no sense...")
                    continue

    def get_L(self, l, OAM_list = ['S','P','D','F','G']):

        while True:
            if l !='Unset':
                self.L = str.upper(l)
                break
            else:
                my_input = input("What is L [String value]:")

                try:

                    my_input = str.upper(my_input)

                    if (my_input not in OAM_list):
                        print("That makes no sense... it is not in " +
                              OAM_list)
                        continue
                    else:
                        self.L = my_input
                    break
                except:
                    print("That makes no sense...")
                    continue

    def get_num_L(self, L):

        if L =="S":
            num_L = 0
        elif L == "P":
            num_L= 1
        elif L == "D":
            num_L= 2
        elif L == "F":
            num_L= 3
        elif L == "G":
            num_L= 4
        else:
            num_L = np.nan
```

```
            print("you screwed up")
        self.numerical_L = num_L

    def get_J(self, L, j):
        while True:
            if j != 'Unset':
                j = str.upper(j)
                if j == "Y":
                    self.J = L+0.5
                    break
                else:
                    self.J = L-0.5
                    break
            else:
                my_input = input("High Total Angular Momentum [Y/N]?:")

                try:

                    my_input = str.upper(my_input)

                    if ((my_input != "Y" and my_input != "N") or
                        (my_input == "N" and L == 0)):
                        print(L)
                        print(my_input)
                        print("That makes no sense...")
                        continue
                    else:
                        if my_input == "Y":
                            self.J = L+0.5
                            break
                        else:
                            self.J = L-0.5
                            break
                except:
                    print("That makes no sense (crashed)...")
                    continue

    def get_mj(self, J, MJ):
        while True:
            if MJ != 'Unset':
                self.mj = float(MJ)
                break
            else:
                myinput = input(
                    'enter value for Secondary Angular Momentum, mj:')

                try:
                    myinput = float(myinput)

                    if ((myinput > J) or (myinput < -J) or
                        ((myinput % 0.5) != 0)):
                        print('that makes no sense. mj = -j, -j+1 ...j-1, j:')
                        print(J)
                    else:
                        self.mj = myinput
                        break
                except:
                    print("That makes no sense (crashed)...")
                    continue

    def __str__(self):

        print_string = str(self.energy_level) + str(self.L) + str(self.numerical_L)
      +  str(self.J) +str(self.mj)

        return(print_string)

    def get_name(self,n, l, j):
```

```
        self.name =str(n)+str(l)+str('_')+str('{')+str(j)+str('}')
        pass


    def __init__(self, n = 'Unset' , l = 'Unset', j ='Unset', MJ = 'Unset'):
        self.energy_level = "Unset"
        self.L = "unset"
        self.numerical_L = np.nan
        self.J = np.nan
        self.mj = np.nan
        self.name = "unset"

        self.set_N(n)
        self.get_L(l)
        self.get_num_L(self.L)
        self.get_J(self.numerical_L,j)
        self.get_mj(self.J, MJ)
        self.get_name(self.energy_level,self.L, self.J)
        pass

def create_subplot(x, y, title, ax=None, color=None, string_data=None, labels=None, linestyles=None):
    if ax is None:
        ax = plt.axes()

    ax.grid(color='black', linestyle='--', alpha=0.5)
    ax.grid(which='minor', color='grey', linestyle='--', alpha=0.5)
    ax.set_title(title)

    if labels is not None and string_data is not None:
        label_update = labels.format(*string_data)
        plots = ax.plot(x, y, color=color, label=label_update, linestyle=linestyles)
        ax.legend(loc='upper right', ncols=1, prop={'size': 18})
    else:
        plots = ax.plot(x, y, color=color)

    ax.set_xlim(-25, 25)

    return plots

def create_inset(ax, x, y, color=None, linestyles=None, x1=None, x2=None, y1=None, y2=None, Z=None):
    if x1 is not None:
        #axins = zoomed_inset_axes(ax, zoom=Z, loc=2)
        axins = inset_axes(ax, width="20%", height="80%",  loc='upper left', bbox_to_anchor=(0.04, 0, 1, 0.95), bbox_transform= ax.transAxes)#, loc='upper left'

        for i in range(len(y)):
            axins.plot(x[i], y[i], color=color[i], linestyle=linestyles[i])

        axins.set_xlim(x1, x2)
        axins.set_ylim(y1, y2)

        axins.yaxis.get_major_locator().set_params(nbins=3)
        axins.xaxis.get_major_locator().set_params(nbins=3)
        #axins.tick_params(labelleft=False, labelbottom=False)
        axins.grid(color='black', linestyle='--', alpha=0.5)
        axins.grid(which='minor', color='grey', linestyle='--', alpha=0.5)

        mark_inset(ax, axins, loc1=3, loc2=1, fc="none", ec="black", color = 'black')


if __name__ =='__main__':


```

```
   ########## data for main program theory#######

    config = liams_windows_config.liams_windows_config() # this is my windows save location

    atom = [Rb85() ] # this is the atom of interest Rb85(), Cs()

    #this asks the user how may state of the atom they wish to search
    natline = [[],[],[],[],[]]
    EITcon = [[],[],[],[],[]]
    laserlin =[]

    # ax1 ,ax2,ax1 ax2,
    fig, ( ax3) = plt.subplots(1,1, sharex =True, figsize= (14,12))


    title =['A) Probe', 'A) EIT Contrast','A) RF Sensing']
    Supertitle = '$^{133} Cs$ Absorption Spectra Changes for off Resonance RF'

    fig.supylabel('Transmission')
    fig.supxlabel('$\Delta$p (MHz)')
    fig.suptitle(Supertitle)

    test_chi_f_list_nat = [[],[],[],[],[]]
    test_chi_f_list_nat_lin = [[],[],[],[],[]]
    delta_p =[[],[],[],[],[]]
    trans_nat=[[],[],[],[],[]]


    Linewidth = [0.1, 1, 10, 100, 1000] # laser linewidths
    Rf_ress= [-20, -10, 0, 10, 20] # off resonances for RF
    couplingpower = [32, 64, 96, 128, 160 ]  #Cs coupling [17.3, 50, 100, 150, 200]  #Rb coupling [32, 64, 96, 128, 160 ]
    rfpower = [0.5, 1, 1.5, 2, 3.57]#rb RF poweer[0.5, 1, 1.5, 2, 3.57] #Cs RF power [0.05, 0.25, 0.5, 0.75, 1]
    probe_power = Rf_ress #rb probe powers [ 0.005, 0.01, 0.017, 0.025, 0.05 ]# Cs probe powers [0.0032, 0.0064, 0.0096,0.0128, 0.016]

###### variables ######
    blank = [] # Full half maxium store for probe only transition
    EITconl = []# Full half maxium store for probe only transition
    rfspilt =[]# RF splitting measurement

    for p in range(len(probe_power)): # this loops through the probe powers
        print(probe_power[p])
        for number_states in range (2, 5): ## this will display rf senisng when (2, 5)
        #number_states = 2 #int(input('please enter the number of state you are interested in: '))
            print(number_states)

            waist_p =160 #hree L2000  # Cs paper waist 708, Rb paper wait 160
            #print(waist_p )
            waist_c =244 #three L 2000    #Cs paper waist 708, Rb paper waist 244


            chi_list_p =[]
            chi_list_p_basic= []
            chi_list_c = []
            chi_list_rf =[]

            trans_doppler =[]

            chi_max =[]
            chi_p = []

            a = [] #used to store the probe power as a list

```

```
            temp = 298.15 #20.85 #298.15 # 25 C


            for i in range(0,   1):

                state_array = [] # this is a list generated by the user for state information
                laser_trans = [] # this is a list to store laser information
                #a.append(probe_p)
               # print('temp= '+ str(temp[i]))

                for j in range(number_states):
                    #this is used to build the state information array from user input
                    #print('please enter information for ' + str(i) + 'th tranistion-----')
                    if j == 0:
                        this_state = state_describing_object(5 , 's',  'y', 0.5) # this is the object built from user input
                    elif j == 1:
                        this_state = state_describing_object(5 , 'p', 'y', 1.5) # this is the object built from user input
                    elif j == 2:
                        this_state = state_describing_object(26, 'd', 'Y', 2.5)#83, 'D',  'y', 2.5) #this is the object built from user input
                    elif j == 3:
                        this_state = state_describing_object(27, 'p', 'y', 1.5)
                    #else:
                    #    this_state = state_describing_object(44,'g','y', 4.5)# 84, 'd',  'y', 2.5) #this is the object built from user input

                    state_array.append(this_state) # this is n number of state informatin the user asked for

                for b in range(number_states-1):
                    #print('please enter laser information for ' + str(i) + 'th tranistion-----')
                    if b == 0:  # 0.000589, 0.00101, 0.001313, 0.002095,0.002589, 0.00334, 0.00354
                        laser_param = laser_parameter('L','Unset',0.0175 , waist_p, 1,'N','Y',-250, 250,  1, temp, 1) #112 Isat = 0.0005208probe beam has 100 micron beam waist
                    elif b == 1: #44.15
                        laser_param = laser_parameter('L','Unset', 32, waist_c, 1,'Y', 'n','Unset','Unset', 1, temp, 1) #0.00512 3 laser system

                    elif b == 2: #65
                        laser_param = laser_parameter('r', 0.5, 'Unset', 'Unset', -1,'N','N',probe_power[p],probe_power[p], 100, temp, -1)
                    #else:
                    #    laser_param= laser_parameter('R',5.9, 'Unset', 'Unset', 1, 'Y', 'N', 'Unset','Unset', 'Unset', temp)
                    #laser_parameter()
                    laser_trans.append(laser_param)

                dephasing_step = 9003#1000001 #int(input('please enter number of steps in the dephasing:'))
                #print(laser_trans)


                # This will search and merge the state of interest with the data frame of state generated in CSV
                # Rb_state_lifetimes_test.csv (this file is generated using the Rb_lifetimes.py program)
                intrested_states = transistion.transition_picker(state_array, atom)

                config = liams_windows_config.liams_windows_config() # This is the config route and file location on my work laptop
```

```
634
635                 intrested_states.to_csv(config.project_loc+"/Data/CSV/extracted.CSV
    ") # test CSV for pulled states
636
637
638
639                 Trans_wave = atom_wave.transition_generator_excitation(atom,
    state_array, laser_trans) # This calls the function to get the tranistion rabi
     frequency
640                 decay_wave = atom_wave.transition_generator_decay(atom, state_array
    , laser_trans)# this calls the function to get the decay rates
641
642
643                 Trans_wave.to_csv(config.project_loc+"/Data/CSV/trans_wave.CSV") #
    test CSV for pulled states
644                 decay_wave.to_csv(config.project_loc+"/Data/CSV/decay_wave.CSV") #
    test CSV for pulled states
645
646
647                 ### steady state builder for EIT for an nxn steady state system
648                 chi, delta_p_1 = EITSS.EIT_builder(number_states,Trans_wave,
    decay_wave, laser_trans, dephasing_step)
649
650
651
652
653
654                 chi_list_p_basic.append(chi[:,0]) ## non doppler profile
655                 delta_p[p].append(delta_p_1)
656
657
658                 #### Calculate the real Transmission Units chek
659                 ####constanst values
660
661                 e_0 = np.longdouble(8.85418782*(10**(-12)))
662                 Bolt = np.longdouble(1.3806*10**(-23))## m^2 kg s^-2 K^-1
663                 a0 = np.longdouble(5.29*10**(-11)) # bohr radius m
664                 e = np.longdouble(1.602*(10**(-19)))# charge of an electron C
665                 wave_num = np.longdouble(2*np.pi/(780*10**(-9)))
666
667                 T = laser_param.temp #
668                 pres2 = np.longdouble(atom[0].getPressure(T)) #Pa
669                 n2 = np.longdouble(atom[0].getNumberDensity(temp))*((np.pi*((
    waist_p*10**(-6))**2)*0.072)/(np.pi*((0.0125)**2)*0.072))
670
671
672
673
674                 rabi_p = np.longdouble((Trans_wave.iloc[0]['rabi Freq rads/s']))
675
676                 dipole_12 = np.longdouble(Trans_wave.iloc[0]['Dipole_moment']*(a0*e
    ))
677
678                 E = np.longdouble((hbar*rabi_p)/dipole_12)
679
680                 L = 0.0718
681
682                 Alpha = ((2*dipole_12)/(E*e_0))
683
684                 ## non doppler Alpha*n* Alpha*n2*
685                 Chi_val_nat = Alpha*n2*(np.imag(chi_list_p_basic[0]))
686
687                 test_chi_f_list_nat[p].append(Chi_val_nat)
688               ### peak and splitting for spectra.
689                 if number_states == 2:
690                     peaks, _ = sn.find_peaks(Chi_val_nat) #Chi_val_nat peak finder
691
```

```
                    result_half_n, _, _,_ = sn.peak_widths((Chi_val_nat), peaks,
    rel_height = 0.5)


                    coarse_upper =(laser_trans[0].dephase_upper - 25*10**6)/(
    dephasing_step/3)
                    coarse_lower = (-25*10**6-laser_trans[0].dephase_lower)/(
    dephasing_step/3)
                    high_res_mid =(25*10**6 + 25*10**6)/(dephasing_step/3)


                    natline[p].extend(result_half_n*-1*(coarse_upper-high_res_mid-
    coarse_lower)/(10**6))
                    blank.append(natline[p][0]/(2*np.pi))

                if number_states == 3:
                    peaks, _ = sn.find_peaks(test_chi_f_list_nat[p][0]-
    test_chi_f_list_nat[p][1]) #Chi_val_nat
                    result_half_n_EIT, _, _,_ = sn.peak_widths((test_chi_f_list_nat
    [p][0]-test_chi_f_list_nat[p][1]), peaks, rel_height = 0.5)

                    coarse_upper =(laser_trans[0].dephase_upper - 25*10**6)/(
    dephasing_step/3)
                    coarse_lower = (-25*10**6-laser_trans[0].dephase_lower)/(
    dephasing_step/3)
                    high_res_mid =(25*10**6 + 25*10**6)/(dephasing_step/3)


                    EITcon[p].extend(result_half_n_EIT*-1*(coarse_upper-
    high_res_mid-coarse_lower)/(10**6))

                    EITcon1.append(EITcon[p][0]/(2*np.pi))

                if number_states == 4:
                    peaks, _ = sn.find_peaks(test_chi_f_list_nat[p][0]-
    test_chi_f_list_nat[p][2]) #Chi_val_nat
                    if len(peaks) > 1:
                        result_half_n_rf, _, _,_ = sn.peak_widths((
    test_chi_f_list_nat[p][0]-test_chi_f_list_nat[p][2]), peaks, rel_height = 0.5)

                        peak_0 = int(peaks[0])
                        peak_1 = int(peaks[1])
                        peaks_sp = peak_1-peak_0

                        coarse_upper =(laser_trans[0].dephase_upper - 25*10**6)/(
    dephasing_step/3)
                        coarse_lower = (-25*10**6-laser_trans[0].dephase_lower)/(
    dephasing_step/3)
                        high_res_mid =(25*10**6 + 25*10**6)/(dephasing_step/3)


                        x = ((peaks_sp)*-1*(coarse_upper-high_res_mid-coarse_lower)
    /(10**6))

                        rfspilt.append(x)
                    else:
                        rfspilt.append(0)




                exp_dat_1 = (-wave_num*L*(Chi_val_nat))
                trans_nat[p].append(np.exp(exp_dat_1)) ### Transmission data

    #Functionalise plot data for multiple runs
    color = ['black', 'blue', 'red', 'purple', 'green'] #list colors
    linestyles  =['solid', 'dashed','dotted', 'dashdot', (0,(1,1))]
    labelone = r'2$\pi\cdot${1:.3f} MHz' #2$\pi$ $\mu$W,
```

```
    labeltwo = r'2$\pi\cdot${1:.3f} MHz' #2$\pi$ {0:.2f} mW,{0:.2f} mW,
    labelthree = r'{0:.0f} MHz, {1:.3f} MHz' #{0:.2f} V/m,

    # Loop over probe_power to create plots
    for p in range(len(probe_power)):
        #create_subplot(delta_p[p][0]/(10**6), trans_nat[p][0], title[0], ax1, color[p], (probe_power[p], blank[p],), labelone, linestyles[p])
        #create_subplot(delta_p[p][1]/(10**6), trans_nat[p][1], title[1], ax2, color[p], (probe_power[p], EITconl[p],), labeltwo, linestyles[p])
        create_subplot(delta_p[p][2]/(10**6), trans_nat[p][2], title[2], ax3, color[p], (probe_power[p], rfspilt[p],), labelthree, linestyles[p])

    #Create inset
    #create_inset(ax1, [delta_p[i][0]/(10**6) for i in range(len(probe_power))], [trans_nat[i][0] for i in range(len(probe_power))], color, linestyles, -1, 1, 0.505, 0.525, 10)

    #create_inset(ax2, [delta_p[i][1]/(10**6) for i in range(len(probe_power))], [trans_nat[i][1] for i in range(len(probe_power))], color, linestyles, -4, 4, 0.5, 0.9, 1.1)

    #create_inset(ax3, [delta_p[i][2]/(10**6) for i in range(len(probe_power))], [trans_nat[i][2] for i in range(len(probe_power))], color, linestyles, -6, -3, 0.5, 0.9, 1.1)

    plt.show()
    plt.savefig(config.project_loc + '/Data/Graphs/'+ Supertitle +'.png')
    plt.close()
    print('computation complete')
    pass
```

2 Directory Configuration

```
### this file determines the location of files in my local system
class liams_windows_config():

    def file_system_params(self):
        self.project_loc = "C:/Users/liamw/Desktop/RYDBERG_GIT-1/src/PhD_Rydberg_code_version_1/Springer_code"

        pass

    def __str__(self):
        return("Project Loc: " + self.project_loc)

    def __init__(self):
        self.file_system_params()
        pass
```

3 Transition Search subprogram

```
# -*- coding: utf-8 -*-
"""
   '''
    @author: Liam Bussey
    @return: Panda Dataframe, table of....
    This program searches the data frame created by Rydberg lifetime to pull two indivdual tranisition
    @param: n is the lower state, principle qm
    @param: l is the orbital angular momentum l = 0,1,...n-1
    @param: j is the total angular momentum j = S+l
```

```
    @param: mj: secondary total angular momentum quantum number mj = -j, -j+1...j
    -1, j
    @param: verbose = false wont print, if true willl print

    '''

"""
import sys
sys.path.append('C:/Users/liamw/Desktop/RYDBERG_GIT-1')
sys.path.append('C:/Users/liamw/Desktop/Rydberg_Git/src')
sys.path.append('C:/Users/liamw/Desktop/Rydberg_Git/src/PhD_Rydberg_code_version_1')

import pandas as pd
import numpy as np
import liams_windows_config
import csv

def state_array_dataframe_generator(state_array):

    state_df = pd.DataFrame(columns = ['name','n','l','j','mj'])

    for i in range(len(state_array)):
        single_record = pd.DataFrame({"name": [state_array[i].name],
                         "n": [state_array[i].energy_level],
                         "l": [state_array[i].numerical_L],
                         "j": [state_array[i].J],
                         "mj": [state_array[i].mj],
                         })

        state_df = pd.concat([state_df,single_record])

    return(state_df)

def get_state_lifetime(atom,N,L,J,T=300):
    #atom = atomic species and natrually occuring or isotopic
    #N = Princple QN, L = Orbital Angular Momentum QN,
    #J= Total angular momentum,
    #T = Temperature in Kelvin

    val = [atom[i].getStateLifetime(int(N),
                                int(L),
                                J,
                                temperature = T,
                                includeLevelsUpTo=int(N+10)
                                )for i in range(len(atom))]
    return(val)

def transition_picker(state_array, atom, vbose = False):

    config = liams_windows_config.liams_windows_config()

    state_df = state_array_dataframe_generator(state_array)

    name_list = [atom[i].__str__().split(".")[2].split(" ")[0]+(' in s')
                            for i in range(len(atom))]

    state_df1 = pd.DataFrame(columns=['name','n','l','j','mj'])

    state_df1=pd.concat([state_df1, state_df])

    state_df1[name_list] = state_df.apply(lambda x: get_state_lifetime(atom, x.n, x
    .l, x.j), axis= 1, result_type = 'expand')


    return(state_df1)

if __name__ =='__main__':
```

```
75
76      print('this should be inported into main file')
77      pass
```

4 Atom transition subprogram

```
1  # -*- coding: utf-8 -*-
2  """
3
4   test file wavelength search
5
6
7  """
8  import sys
9  sys.path.append('C:/Users/liamw/Desktop/Rydberg_Git/Github/rydberg_theory')
10 sys.path.append('C:/Users/liamw/Desktop/Rydberg_Git/Github/rydberg_theory/src')
11 sys.path.append('C:/Users/liamw/Desktop/Rydberg_Git/Github/rydberg_theory/src/
       python')
12
13 import pandas as pd
14 import numpy as np
15 import liams_windows_config
16
17 import arc
18
19 def state_array_dataframe_generator(state_array, laser):
20   #this created the data frame for user input data from the main program which is
        then ready to be amended
21     state_df = pd.DataFrame(columns = ['transition','feild','n1','l1','j1',
22                                        'mj1','n2','l2','j2','mj2','q','p','w',
23                                        'p_rf', 'linewidth'])
24     #print(len(state_array)-1)
25     for i in range(len(state_array)-1):
26         single_record = pd.DataFrame({"transition": state_array[i].name +str(' -> '
       )+state_array[i+1].name,
27                           "n1": [state_array[i].energy_level],
28                           "l1": [state_array[i].numerical_L],
29                           "j1": [state_array[i].J],
30                           "mj1": [state_array[i].mj],
31                           "n2": [state_array[i+1].energy_level],
32                           "l2": [state_array[i+1].numerical_L],
33                           "j2": [state_array[i+1].J],
34                           "mj2": [state_array[i+1].mj],
35                           "feild": [laser[i].feild],
36                           "q": [laser[i].q],
37                           "p": [laser[i].p],
38                           "w": [laser[i].w],
39                           "p_rf": [laser[i].p_rf],
40                           'temp': [laser[i].temp],
41                           'linewidth': [laser[i].laser_line]})
42         state_df = pd.concat([state_df, single_record])
43     return(state_df)
44
45 def state_array_dataframe_generator_decay(state_array, laser):
46     #this created the data frame for user input data from the main program which is
         then ready to be amended
47     state_df = pd.DataFrame(columns = ['transition','n2','l2','j2',
48                                        'mj2','n1','l1','j1','mj1'])
49
50     for i in range(len(state_array)-1):
51         single_record = pd.DataFrame({"transition": state_array[i+1].name +str(' ->
       ')+state_array[i].name,
52                           "n2": [state_array[i].energy_level],
53                           "l2": [state_array[i].numerical_L],
54                           "j2": [state_array[i].J],
```

```
                            "mj2": [state_array[i].mj],
                            "n1": [state_array[i+1].energy_level],
                            "l1": [state_array[i+1].numerical_L],
                            "j1": [state_array[i+1].J],
                            "mj1": [state_array[i+1].mj],
                            'temp': [laser[i].temp]})
        state_df = pd.concat([state_df, single_record])
    return(state_df)

def getwavelength(atom,n1,l1,j1,n2,l2,j2):
    # this returns the wavelength associated with the transition
    val_list =[(atom[i].getTransitionWavelength(n1,l1,j1,n2,l2,j2)*1e9) for i in range(len(atom))]
    return(val_list)

def getfrequency(atom,n1,l1,j1,n2,l2,j2):
    # this returns the frequency associated with the transition
    val_list =[(atom[i].getTransitionFrequency(n1,l1,j1,n2,l2,j2)) for i in range(len(atom))]
    return(val_list)

def getdecay(atom, n1, l1, j1, n2, l2, j2, T):
    ### this finds the spontanous decay from state 1 to state 2
    val_list = [(atom[i].getTransitionRate(n1, l1, j1, n2, l2, j2, temperature=T)) for i in range(len(atom))]
    ### return tranisition rante in s-1 (Hz)
    return(val_list)

def getmass(atom):
    val_list =[atom[i].mass for i in range(len(atom))]
    return(val_list)

def getdens(atom, temp):
    val_list =[atom[i].getNumberDensity(temp) for i in range(len(atom))]
    return(val_list)

def transition_generator_excitation(atom_list, interested_state,laser):
    #atom = atomic species and natrually occuring or isotopic
    #interested states provided the user defined states
    #This will find the relevant tranistion wavelength between the given states
    #that are of interest.
    # i.e Rb 5S_{1/2} -> 5P_{3/2} == 780nm
    state_df = state_array_dataframe_generator(interested_state, laser)
    name_list = ['wavelength in nm for Rb']
    name_list2 = ['freq in Hz for Rb']
    name_list3 = ['Transition Rate(Hz) with BBR for Rb']
    name_list4 = ['Transition Rate(Hz) at 0.1K for Rb']
    name_list5= ['rabi Freq rads/s']
    name_list6 = ['Dipole_moment']
    name_list7 = ['mean_speed']
    name_list8 =['mass']
    name_list9 = ['Atomic_number_dens']
    trans_wave = pd.DataFrame (columns = ['transition','feild','n1','l1','j1',
                               'mj1','n2','l2','j2','mj2', 'q','p','w','p_rf', 'temp'])

    trans_wave = pd.concat([trans_wave, state_df])

    trans_wave[name_list] = trans_wave.apply(lambda x: getwavelength(atom_list,
                                                x.n1, x.l1, x.j1, x.n2, x.l2, x.j2),
                                                axis=1, result_type='expand')
    trans_wave[name_list2] = trans_wave.apply(lambda x: getfrequency(atom_list,
                                                x.n1, x.l1, x.j1, x.n2, x.l2, x.j2),
                                                axis=1, result_type='expand')
    trans_wave[name_list3] = trans_wave.apply(lambda x: getdecay(atom_list,
```

```
                                                    x.n1, x.l1, x.j1, x.n2, x.
    l2, x.j2, T=x.temp),
                                                    axis=1, result_type='expand
    ')
    trans_wave[name_list4] = trans_wave.apply(lambda x: getdecay(atom_list,
                                                    x.n1, x.l1, x.j1, x.n2, x.
    l2, x.j2, T=0.1),
                                                    axis=1, result_type='expand
    ')
    trans_wave[name_list5] = trans_wave.apply(lambda x: get_rabi(atom_list, x.feild
    ,
                                                   x.n1, x.l1, x.j1, x.mj1, x.
    n2, x.l2, x.j2,
                                                    x.q , x.p, x.w, x.p_rf, x.
    linewidth),
                                                    axis=1, result_type='expand
    ')

    trans_wave[name_list6] = trans_wave.apply(lambda x: get_dipole(atom_list, x.
    feild,
                                                   x.n1, x.l1, x.j1, x.mj1, x.
    n2, x.l2, x.j2, x.mj2,
                                                    x.q , x.p, x.w, x.p_rf),
                                                    axis=1, result_type='expand
    ')

    trans_wave[name_list7] = trans_wave.apply(lambda x: get_speed(atom_list, x.
    feild,
                                                   x.n1, x.l1, x.j1, x.mj1, x.
    n2, x.l2, x.j2, x.mj2,
                                                    x.q , x.p, x.w, x.p_rf, x.
    temp),
                                                    axis=1, result_type='expand
    ')
    trans_wave[name_list8] = trans_wave.apply(lambda x: getmass(atom_list),
                                                    axis=1, result_type='expand
    ')
    trans_wave[name_list9] = trans_wave.apply(lambda x: getdens(atom_list, x.temp),
                                                    axis=1, result_type='expand
    ')
    return (trans_wave)

def transition_generator_decay(atom_list, interested_state, laser):
    #atom = atomic species and natrually occuring or isotopic
    #interested states provided the user defined states
    #This will find the relevant tranistion wavelength between the given states
    #that are of interest.
    # i.e Rb 5S_{1/2} -> 5P_{3/2} == 780nm
    state_df = state_array_dataframe_generator_decay(interested_state, laser)
    name_list = ['wavelength in nm for Rb']
    name_list2 = ['freq in Hz for Rb']
    name_list3 = ['Transition Rate (Hz) with BBR for Rb']
    name_list4 = ['Transition Rate at (Hz) 0.1K for Rb']

    trans_wave = pd.DataFrame (columns = ['transition','n1','l1','j1',
                                'mj1','n2','l2','j2','mj2', 'temp'])

    trans_wave = pd.concat([trans_wave, state_df])

    trans_wave[name_list] = trans_wave.apply(lambda x: getwavelength(atom_list,
                                                   x.n1, x.l1, x.j1, x.n2, x.l2, x
    .j2),
                                                   axis=1, result_type='expand')

    trans_wave[name_list2] = trans_wave.apply(lambda x: getfrequency(atom_list,
                                                   x.n1, x.l1, x.j1, x.n2, x.l2, x
    .j2),
                                                   axis=1, result_type='expand')
```

```
165
166     trans_wave[name_list3] = trans_wave.apply(lambda x: getdecay(atom_list,
167                                                         x.n1, x.l1, x.j1, x.n2, x.
        l2, x.j2, x.temp),
168                                                         axis=1, result_type='expand
        ')
169
170     trans_wave[name_list4] = trans_wave.apply(lambda x: getdecay(atom_list,
171                                                         x.n1, x.l1, x.j1, x.n2, x.
        l2, x.j2, 0.0),
172                                                         axis=1, result_type='expand
        ')
173     return (trans_wave)
174 #### Using the FWHM to calculate laser waist (must check units as linewidth is
        giveing in KHz) can be changed back if needed using   w
175
176 def get_rabi(atom_list, feild ,n1, l1, j1, mj1, n2, l2, j2, q, p, w, p_rf,
        linewidth):
177
178     if (feild == 'L'):
179
180         rabiFreq = [(atom_list[i].getRabiFrequency(n1, l1, j1, mj1,
181                                                    n2, l2, j2,
182                                                    q, p, w)) for i in range(len(
        atom_list))]
183         rabiFreq = rabiFreq ### return in rad-1 divide by 2Pi for Hz
184         return(rabiFreq)
185
186     else:
187
188         rabiFreq = [(atom_list[i].getRabiFrequency2(n1, l1, j1, mj1,
189                                                    n2, l2, j2,q,
190                                                    p_rf)) for i in range(len(
        atom_list))]
191         rabiFreq =rabiFreq  ### return in rad-1 divide by 2Pi for Hz
192         return(rabiFreq)
193
194 def get_dipole(atom_list, feild ,n1, l1, j1, mj1, n2, l2, j2, mj2, q, p, w, p_rf):
195
196     dipolemoment = [(atom_list[i].getDipoleMatrixElement(n1, l1, j1, mj1,
197                                                          n2, l2, j2, mj2, q))
198                                                         for i in range(len(
        atom_list))]
199     return(dipolemoment)
200
201 def get_speed(atom_list, feild ,n1, l1, j1, mj1, n2, l2, j2, mj2, q, p, w, p_rf,
        temp):
202
203     meanspeed = [atom_list[i].getAverageSpeed(temp) #returns ms-1
204                  for i in range(len(atom_list))]
205     return(meanspeed)
206
207 if __name__ =='__main__':
208     print('this should be inported into main file')
209
210     pass
```

5 EIT to Main subprogram

```
1 import qutip as qp
2 import numpy as np
3 import pandas as pd
4 import matplotlib.pyplot as plt
5
6 from scipy.constants import k as k_b
```

```
from scipy.constants import c as c_c ## speed of light
from scipy.constants import h as c_h # planck constant
from scipy.constants import hbar as hbar #reduced planks

def basis_generator(number_states):

    #### basis state for a N system
    basis_list =[]

    for i in range(number_states):
        basis = qp.basis(number_states, i)
        basis_list.append(basis)

    return(basis_list)

def atomic_operators(basis, number_states):

    #### sigma operator or atomic jump operator
    sigma_operator = []#np.zeros(shape = ((number_state),(number_state)))

    for i in range(number_states):
        for b in range(number_states):
            sigma = basis[i]*basis[b].dag()

            sigma_operator.append(sigma)

    return(sigma_operator)

def Hamil_builder(number_states, Trans_wave, dephasing):

    Hamiltonian = np.zeros(shape=((number_states),(number_states)))
          ### NxN matrix of zeros to form the base Hamiltonian


 #### how to build this a function builder
    for i in range (number_states- 1):

        Hamiltonian[i][i+1] = -(1/(4*np.pi))*(Trans_wave.iloc[i]['rabi Freq rads/s'
    ])   ### fills the upper off diagonal element for an NXN matrix
        Hamiltonian[i+1][i] = -(1/(4*np.pi))*(Trans_wave.iloc[i]['rabi Freq rads/s'
    ])   ### fills the lower off diagonal element for an NXN matrix

    for i in range (number_states):
        if i == 0:
            Hamiltonian[i][i] = 0.0
        else:
            Hamiltonian[i][i] = dephasing[i-1]
           # forms the diagonal element of the system

    return(Hamiltonian)

def collapse_op(number_state, basis, decay_wave):

    natural_line =[]
    for i in range(number_state - 1):
        #spontaneous decay in Hz
        natural_line.append((decay_wave.iloc[i]['Transition Rate (Hz) with BBR for
    Rb']))

    ls = []
    dim = number_state
    n_l = []

    for j in range(1, dim):
        op = np.zeros((dim, dim))
        natural_line_value = natural_line[j-1]

        op[j-1,j] = natural_line_value
```

```
        ls.append(op)

    ls = np.array(ls)

    return(ls)


def collapse_op_laser(number_state, laser_trans, basis, Trans_wave):
 #laser linewidth in Hz
    laser_line =[]
    for i in range(number_state - 1):
        if  Trans_wave.iloc[i]['feild'] == 'R':
            laser_line.append(0)
        else:
            laser_line.append((Trans_wave.iloc[i]['linewidth']))
    ls = []
    dim = number_state
    q_l = []
    op = np.zeros((dim, dim))

    for j in range(1, dim):
        op = np.zeros((dim, dim))
        laser_line_value = laser_line[:j]
        laser_line_value.reverse()

        for i in range(0, j):
            op[i, j] = (np.sum(laser_line_value[:(j-i)]))

        ls.append(op)
    return(ls)

def dephase_dataframe_generator(laser_trans, Trans_wave):
    dephase_df = pd.DataFrame(columns=['onress', 'dephase_lower',
                                       'dephase_upper',"scanning_source",
                                       'wavelength',"mean_speed",'beam_prop'])

    for i in range(len(laser_trans)):
        single_record = pd.DataFrame({"onress": [laser_trans[i].onress],
                          "dephase_lower": [laser_trans[i].dephase_lower],
                          "dephase_upper": [laser_trans[i].dephase_upper],
                          "scanning_source": [laser_trans[i].scanning_source],
                          "wavelength": [Trans_wave.iloc[i]['wavelength in nm for Rb
    ']/(1*10**9)],
                          "mean_speed": [Trans_wave.iloc[i]['mean_speed']],
                          'transf': [Trans_wave.iloc[i]['freq in Hz for Rb']],
                          'mass': [Trans_wave.iloc[i]['mass']],
                          'beam_prop': [laser_trans[i].beam_prop]})

        dephase_df = pd.concat([dephase_df,single_record])

    return(dephase_df)

def dephase(laser_trans, number_states, dephasing_step, Trans_wave):

    dephase_df = dephase_dataframe_generator(laser_trans, Trans_wave)
    #print(dephase_df.scanning_source)
    dephasing_range = np.zeros(shape=(dephasing_step, (number_states-1)))
    dephasing = np.zeros(shape=(dephasing_step, (number_states-1)))
    dephasing_range_concat =[]

    for i in range(len(dephase_df)):
        (dephasing_step/3)
        upper= dephase_df.iloc[i]['dephase_upper']
        lower = dephase_df.iloc[i]['dephase_lower']

        if upper == 0 and lower == 0:
            step = dephasing_step
            Dephasing_range = np.zeros(shape=(step))
```

```
            dephasing_range[:, i] = Dephasing_range
        if dephase_df.iloc[i]['scanning_source'] == 'N' and dephase_df.iloc[i]['
    onress'] == 'N':
            step = dephasing_step
            Dephasing_range = [upper]*step
            dephasing_range[:, i] = Dephasing_range

        elif dephase_df.iloc[i]['scanning_source'] == 'Y' and dephase_df.iloc[i]['
    onress'] == 'N':
            step = int(dephasing_step/3)
            Dephasing_range = np.linspace([lower, -25*10**6,  25*10**6],
    [-25*10**6, 25*10**6, upper],step, axis = 1)
            Dephasing_range.reshape(dephasing_step)
            dephasing_range_concat = np.ravel(Dephasing_range)
            dephasing_range[:, i] = dephasing_range_concat



    #print(dephasing_range)

    for j in range(number_states - 1):
        if j == 0:
            #dephasing[:,j] = Dephasing_range
            dephasing[:,j] = dephasing_range[:,j]
        else:
            dephasing[:,j] = (dephasing[:,j-1]+dephasing_range[:,j])#
    dephasing_range[:,j])

    dephasing = dephasing
    return(dephasing)

def EIT_builder(number_state, Trans_wave, decay_wave, laser_trans,dephasing_step):

    ### bais is the ket vector for the number of states in the atom
    basis_list = basis_generator(number_state)

    #### atomic operator is the jump operator ie basis[0]*basis[1].dag() this will
    from an nxn matrix
    atomic_operator = atomic_operators(basis_list, number_state)

    ## this calls the collapse function to form the collapse operator for the
    system
    collapse_2 = collapse_op(number_state, basis_list, decay_wave) # Natural decay

    collapse_l = collapse_op_laser(number_state, laser_trans, basis_list,
    Trans_wave) # laser linewidth

    collapse_n_l_l =[]
    collapse_n_list =[]

    for i in range(number_state-1):

        collapse_n = collapse_2[i]

        collapse_n_list.append(-np.sqrt(collapse_n))#

        collapse_n_l =  -((np.sqrt(collapse_l[i]+collapse_2[i]))) #
        collapse_n_l_l.append((collapse_n_l))

    collapse_n_l_q = []
    collapse_n_q = []


    for i in range(number_state-1):
        collapse_n_q.append(qp.Qobj(collapse_n_list[i]))
        collapse_n_l_q.append(qp.Qobj(collapse_n_l_l[i]))

```

```
198
199
200     chi_1 = pd.DataFrame()### first column electric susceptability for the |0> --->
          |1>
201                         ### second column is chi for the |1> ---> |2>
202                         ### third column is chi for the |2>--> |3> and so on
203
204     chi_2 = pd.DataFrame()
205
206
207     chi_value = np.zeros(shape=(dephasing_step,(number_state-1)), dtype=complex)
208     chi_value_2 = np.zeros(shape=(dephasing_step,(number_state-1)), dtype=complex)
209
210     chi_test_1 = np.zeros(shape=(dephasing_step,(number_state-1)), dtype=complex)
211     chi_test_2 = np.zeros(shape=(dephasing_step,(number_state-1)), dtype=complex)
212
213     dephasing_2 = dephase(laser_trans,number_state, dephasing_step, Trans_wave)
214
215     for i in range(dephasing_step):
216
217
218         ##non doppler Hamiltonian
219         Hamiltonian = (Hamil_builder(number_state, Trans_wave, dephasing_2[i, :]))
          # this calls the hamiltonian builder function
220
221         quantum_obj_2 = qp.Qobj(Hamiltonian)
222
223         ##### steady state solution to non doppler hamiltonian
224         rho_ss_1 = qp.steadystate(quantum_obj_2, collapse_n_q) #natural
225         #rho_ss_2 = qp.steadystate(quantum_obj_2, collapse_n_l_q) # natrual +laser
226
227         ## expectation value of systemm
228         for b in range(1, number_state):
229
230             a = b + (number_state*(b-1))
231            ## non doppler expectation value
232
233             chi_test_1[i, b-1] = qp.expect(atomic_operator[a], rho_ss_1) #natural
234             #chi_test_2[i, b-1] = qp.expect(atomic_operator[a], rho_ss_2) # natrual
           +laser
235
236
237     # electric susceptability for the off diagonal elements
238
239     chi_1 = chi_test_1 # natrual
240     #chi_2 =chi_test_2 # natrual +laser
241
242     dephasing_graph = [] # transferes the probe dephasing
243
244     laser_state_df = dephase_dataframe_generator(laser_trans, Trans_wave)
245
246     # dephasing list not 100% needed
247     for i in range(number_state-1):
248         if ((laser_state_df.iloc[i]['onress'] == 'N') & (laser_state_df.iloc[i]['
          scanning_source'] == 'Y')):
249             dephasing_graph = dephasing_2[:,i]#*2*np.pi  # transferes the probe
          dephasing (need to work out how to transfere the relevent dephasing)
250
251     return(chi_1, dephasing_graph)
252
253 if __name__ == '__main__':
254     print('this should be inported into main file')
```