Physics 302 Final Project

Nathan Wolthuis and Shufan Xia

May 2021

We demonstrate and describe the steps required to make two interactive plots with *python* in order to visualize the two important conclusions from Section 4.4 Townsend: 1) time procession of S_z, the angular spin momentum in the z axis, and 2) the transition rate at different ω_0, ω_1 and ω. Our model explores the creation of numerical solutions for this phenomena, and provides interactive plots so that we can examine the effects of changing frequencies in real time.
Our interactive plots are made in *SEQ.py*[1]. To see the visualizations, run "python SEQ.py" in command line.

1 Magnetic Resonance

Magnetic resonance maintains a variety of applications ranging from investigating the local magnetic fields in atoms or molecules in nuclear magnetic resonance spectroscopy (MRS) to medical imaging in magnetic resonance imagining (MRI).

In magnetic resonance, a spin-1/2 particle is put in a constant magnetic field in the z direction, $B_0\hat{k}$, and an oscillating magnetic field in the x direction, $B_1 \cos\omega t$. This oscillating magnetic field induces the transition between $|+z\rangle$ to $|-z\rangle$. The Hamiltonian is given by:

$$\hat{H} = \frac{ge}{2mc}\hat{S} \cdot (B_1 \cos(\omega t)\hat{k} + B_0\hat{k}) = \omega_0\hat{S}_z + \omega_1(\cos\omega t)\hat{S}_x \tag{1}$$

Suppose the ket vector for the spin state of the particle in terms of S_z basis is $|\psi\rangle = \begin{pmatrix} a(t) \\ b(t) \end{pmatrix}$. The time development of this spin state is given by the Schrödinger equation, $\hat{H}|\psi\rangle = i\hbar\frac{d}{dt}|\psi\rangle$. When we plug in the Hamiltonian in Eq. 1, and apply the two Pauli spin matrices for $\hat{S}_z$ and $\hat{S}_x$, we have:

$$\frac{\hbar}{2}\begin{pmatrix} \omega_0 & \omega_1\cos\omega t \\ \omega_1\cos\omega t & -\omega_0 \end{pmatrix}\begin{pmatrix} a(t) \\ b(t) \end{pmatrix} = i\hbar\begin{pmatrix} \dot{a}(t) \\ \dot{b}(t) \end{pmatrix} \tag{2}$$

The goal is to solve this above matrix and find the time developments of a and b. The three most important parameters in magnetic resonance are: 1) $\omega_0 = \frac{qqB_0}{2mc}$. ω_0 is related to B_0, the strength of constant magnetic field B_0. 2) $\omega_1 = \frac{qqB_1}{2mc}$. Similarly, ω_0 is related to the strength of oscillating magnetic field in the z-direction, B_1. 3) ω is the frequency of the oscillating field.

2 Define and Solve SEQ in *python*

As we have seen in Townsend, solving this SEQ analytically is challenging. Townsend's proposed the approximated solution only when the oscillating field is weak and the system is at resonance. Our model will hopefully be able to provide a numerical solution that is more general. After considering the benefits of numerical integration methods for Rungee-Kutta, half-step, and ODEINT methods, we elected to use *scipy* ODIENT to handle the differential equation solving for our numerical approximation model.

- First, we need to define the derivatives of the desired variables as a function in *python*. To deal with complex numbers in ***python***, we write $a(t)$ as $a_R(t) + ia_I(t)$, where a_R is the real part of $a(t)$ and a_I is the imaginary part of $a(t)$. We maintain a similar practice for $b(t)$ and their respective derivatives $\dot{a}(t)$

[1] https://github.com/shufan1/PHYS302finalproject

and $\dot{b}(t)$. From here we describe any general spin state, ψ, as a combination of $a_R(t), a_I(t), b_R(t), \& \ b_I(t)$. Now, we re-write Eq. 2 as:

$$\dot{a}(t)_R = \frac{1}{2}[\omega_0 a_I(t) + \omega_1 \cos(\omega t) b_I(t)] \tag{3}$$

$$i\dot{a}(t)_I = -\frac{i}{2}[\omega_0 a_R(t) + \omega_1 \cos(\omega t) b_R(t)] \tag{4}$$

$$\dot{b}(t)_R = \frac{1}{2}[\omega_1 \cos(\omega t) a_I(t) - \omega b_I(t)] \tag{5}$$

$$i\dot{b}(t)_I = -\frac{i}{2}[\omega_1 \cos(\omega t) a_R(t) - \omega b_R(t)] \tag{6}$$

In ***python***, we hold the derivatives of $a_R(t)$, $a_I(t)$, $b_R(t)$, and $b_I(t)$ in an array. In order to do so, we have defined a function to evaluate all of the derivatives for a given spin state:

```
import numpy as np

## evaluate the derivatives at a specific state at time t:
    #input: Psi, spin state, an array [aR,aI,bR,bI];
    #        t, the specific time;
    #       and params, [w0,w1,w], the scale for all three frequencies

def f(Psi,t,params):
    w0,w1,w = params
    w0 = w0*w00  #scale
    w1 = w1*w00
    w = w*w00
    aR = Psi[0]
    aI = Psi[1]
    bR = Psi[2]
    bI = Psi[3]
    daR = 1/2*(w0*aI+w1*np.cos(w*t)*bI) # Real part of the derivative a
    daI = -1/2*(w0*aR+w1*np.cos(w*t)*bR) # Imaginary part of the derivative a
    dbR =  1/2*(w1*np.cos(w*t)*aI-w0*bI)
    dbI =  -1/2*(w1*np.cos(w*t)*aR-w0*bR)

    return np.array([daR,daI,dbR,dbI])
```

- From here, we are going to solve the above differential equations by the previously mentioned built-in differential equation solver in the *scipy* package, ODEINT. ODEINT solves the differential method using the LSODA algorithm, which is capable of switching between stiff and nonstiff methods (i.e. for "well behaved" ODE's and "non-well behaved ODE's" respectively). ODEINT initiates the process through attempting a stiff method solution, and if the function seems poorly behaved, switches to a nonstiff method. ODEINT also adapts the step-size throughout the function, as it estimates the error through comparing the results of two simple integrators that are entwined within the function. Using these error estimations it shifts the step size in order to minimize the squared error.

  In summary, the **ODEINT** function evaluates the derivatives of the functions at many discrete time steps and use those derivatives to estimate the time behaviors of the functions. Our sample code is as follows:

```
from scipy.integrate import odeint

psoln = odeint(f, Psi0, t, args=(params,))
     #          f is the function to evaluate the derivative defined above,
     #          Psi0 is the initial spin state,
     #          t is an array time steps where odeint will evaluate the derivatives
```

```
#              and estimate the function
#          params, is the input to function f
```

- Now we use the above code to generate a solution, and calculate the probability of measuring spin-up $|+z\rangle$ and spin-down $|-z\rangle$ at a series of time t:

```
# find P_up, P_down and transition rate
#           input: omega_0, omega_1, omega, initial condition of the spin state r0
#           out put: t: a array of time steps from 0 to 2T1 microseconds
#           P_up: an array of the probability of getting +z when we measure S_z at
#                   each time step
#           P_down: an array of the probability of getting -z when we measure S_z
#                   at each time step
#           transition_rate: the proportion of particles that transition from +z to -z,
#                   defined as the maximum value of P_down.

def P_z(w0,w1,w,Psi0,N):
    t = np.linspace(0,2*T1,N) #time step from t=0 to t= 2T1
    params = [w0,w1,w]
    psoln = odeint(f, Psi0, t, args=(params,)) # solve our SEQ by odeint
    aR = psoln[:,0]
    aI = psoln[:,1]
    bR = psoln[:,2]
    bI = psoln[:,3]
    P_up = list(map(lambda r,i: abs(r+i*1j)**2,aR,aI))
    # probablitly of measuring the spin-up: the magnitude square of the coefficent on |z+>
    P_down = list(map(lambda r,i: abs(r+i*1j)**2,bR,bI))
    transition_rate = np.max(P_down) #transition rate is the
    return t,P_up,P_down,transition_rate
```

3 Visualizing the results

Once we have the numerical solution to Eq 2, we can plot them with *python.matplotlib* and add interactive sliders using the widgets from *matplotlib* package.

Each plot is interactive and allows the user to see the firsthand effects of changing the constant field strength, ω_0, the strength of the oscillating field, ω_1, and its frequency, ω. We can also vary the smoothness (and thus the computational time) for each plot. We chose to use ***python***'s *ax.plot* function, as that provides more user control for the nitty-gritty of the display functions. However should the user wish, much of the plotting should be doable in *matplotlib.pyplot* without too much trouble.

3.1 Time precession of S_z

- We examine the behavior when $\omega = \omega_0$, with the system at resonance:

```
# find the solution, and the probabilities of measuring +z and -z when w=w0
Psi0 = [1.0,0.,0.,0.] #initial condition, spin up: {{1+0j},{0+0j}}
w0,w1,w = 1.0,0.1,1.0 #
t,P_up,P_down,transition_rate = P_z(1.0,0.1,1.0,Psi0,600) #call P_z with 600 time steps
```

- Plot the result in matplotlib

```
import matplotlib.pyplot as plt

plt.rc('font',family ='serif') # use Latex in matplotlib
plt.rc('text',usetex =True)

fig= plt.figure(figsize=(15,13)) # initialize figure
ax = fig.add_subplot(111) # define the axies where we are going to plot P_zup and P_zdown
fig.subplots_adjust(bottom=0.3, top=0.8) # adjust the positions to make room for sliders later

# plot the default result, at resonance, and a weak oscilating field

P_zup_line, = ax.plot(t,P_up,
                    label='Probability of spin up $| \langle +z | \psi \\rangle |^2$')
                    # plot the probability of measuring spin up and save that line as an object
P_zdown_line, = ax.plot(t,P_down,
                    label='Probability of spin down $| \langle -z | \psi \\rangle |^2$')
```

- customize plot title, axis labels, x-axis with time in the unit of microseconds and add annotation to show the transition rate.

```
f0,f1,f_ac = np.array([w0,w1, w])*800 #kHz for plot title,
ax.set_title('$f_0= %0.2f kHz$, $f_1 = %0.2f kHz$(applied field strength),
                $f = %0.2f kHz$(applied field frequency)'%(f0,f1,f_ac), fontsize=25)
ax.set_xlim(0,2*T1)
ax.legend(loc='upper right', fontsize='x-large')
ax.set_xlabel('$t$ ($\mu$s)',fontsize=18)
ax.set_ylabel('$Probablity$ ', fontsize=19)
ax.xaxis.set_label_coords(1.08, -0.01)

#setting up the labels on the x axis to be the time in units of microseconds
xtic0, xtic1, xtic2, xtic3, xtic4 = 0, round(0.5*T1/(10**(-6)), 3), round(T1/(10**(-6)), 3),
                                    round(1.5*T1/(10**(-6)), 3),  round(2.0*T1/(10**(-6)), 3)
xtick_name = (xtic0, xtic1, xtic2, xtic3, xtic4) #convert the x-axis tick to units of microsecond
ax.set_xticks(np.arange(0,2.5 *T1,T1/2))
ax.set_xticklabels(xtick_name,fontsize=18)
ax.grid(True)

# annotate to show the transition rate
annotation = ax.annotate("Transition rate = %0.2f"%transition_rate, (0.66,0.685),
            xycoords = "figure fraction", fontsize =20)
```

- Add interactive slider adjustment using the Slider widget from matplotlib. Adjusting ω_0 and ω_1 essentially means changing the relative magnitude between the constant field and the oscillating field.

```
# Create axes for holding sliders
ax_w0 = fig.add_axes([0.3, 0.22, 0.4, 0.03]) :  w0, strength of the constant field
ax_w0.spines['top'].set_visible(True)
ax_w0.spines['right'].set_visible(True) E w1, strength of the alternating field
ax_w1 = fig.add_axes([0.3, 0.16, 0.4, 0.03])
ax_w1.spines['top'].set_visible(True)
ax_w1.spines['right'].set_visible(True)
ax_w = fig.add_axes([0.3, 0.10, 0.4, 0.03]) # w, frequencies of the oscillating field
ax_w.spines['top'].set_visible(True)
ax_w.spines['right'].set_visible(True)
ax_t = fig.add_axes([0.3, 0.05, 0.4, 0.03]) # create axes for the slider to adjust how many steps
```

```
ax_t.spines['top'].set_visible(True)
ax_t.spines['right'].set_visible(True)
# put four sliders at where we defined our axes
w0_slider = Slider(ax=ax_w0,label='$\omega_0$',valmin=0.,
                valmax=2.,valinit=1.0,facecolor='#cc7000',dragging= True)
w1_slider = Slider(ax=ax_w1,label='$\omega_1$',valmin=0.,
                valmax=0.5,valinit=0.1,facecolor='#cc7000',dragging= True)
w_slider = Slider(ax=ax_w,label='$\omega$',valmin=0.,
                valmax=2.,valinit=1.0,facecolor='#cc7000',dragging= True)
t_slider = Slider(ax=ax_t,label='N steps',valmin=100.,valmax=1000.,
                valfmt='%0.0f',valinit=600,facecolor='#cc7000',dragging= True)
```

- We now define an update function to have ***matplotlib*** re-plot the time precession of the spin in z axis under different values of ω_0, ω_1 and ω from the sliders.

```
# Update values: this function is called whenever the value on the slider is changed
def update(val):
    # retrive the values of w0, w1, w from the sliders
    w0 = w0_slider.val
    w1 = w1_slider.val
    w = w_slider.val
    N = int(t_slider.val) # number of time step must be integer
    t,zup,zdown,transition_rate = P_z(w0,w1,w,Psi0,N)
            #calculate the probability of spin-up or spin-down with the new w0,w1,w values
    P_zup_line.set_data(t, zup) # update the new value on the old line
    P_zdown_line.set_data(t, zdown)
    ax.set_title('$f_0= %0.2f kHz$, $f_1 = %0.2f kHz$(applied field strength),
                $f = %0.2f kHz$(applied field frequency)'%(f0,f1,f_ac), fontsize=25)
    annotation.set_text("Transition rate = %0.2f"%transition_rate) #update annotation
    fig.canvas.draw_idle() #update a figure that has been altered, but not automatically re-drawn

w0_slider.on_changed(update) #when the slider change, call the update function
w1_slider.on_changed(update)
w_slider.on_changed(update)
t_slider.on_changed(update)

plt.show()
```

- Fig 1 shows two of the time processions of the angular spin in the $z-$axis under the conditions of ω_0=1.0, $\omega_1 = 0.1$ and $\omega = 1.0$ (at resonance), as well as under non resonant conditions with ω_0=1.0, $\omega_1 = 0.1$ and $\omega = 0.46$. We also observe that the Period T is dependent on the strength of the oscillating field ω_1.

3.2 Transition rate as a function of ω

The second component of the model is to visualize how the transition rate from $|+z\rangle$ to $|-z\rangle$ depends on the frequency of the oscillating field, ω under different ω and ω_1.

- We make a list of values for the applied oscillating field frequency, w_list. And use the P_z() function defined above to find the transition rate. You could use a loop, iterate over the different values in w_list and apply the function P_z(). We used the lambda function in python to make this process a little bit faster. We also ensure that the time series in P_z. covers at least one period at resonance, $T = \frac{2\pi}{\omega_1}$.

```
Psi0 = [1.,0.,0.,0.] # at t=0, Psi = |+z>
w0,w1 = 1.,0.1 # default value
f0,f1= np.array([w0,w1])*800 #kHZ for plot title
```

```
w_list = np.arange(0.4,1.4,0.005) # a list of applied frequency values
rate = np.array(list(map(lambda w: P_z(w0,w1,w,r0,600), w_list)))[:,3]
    # find the transition rate at each applied freq
```

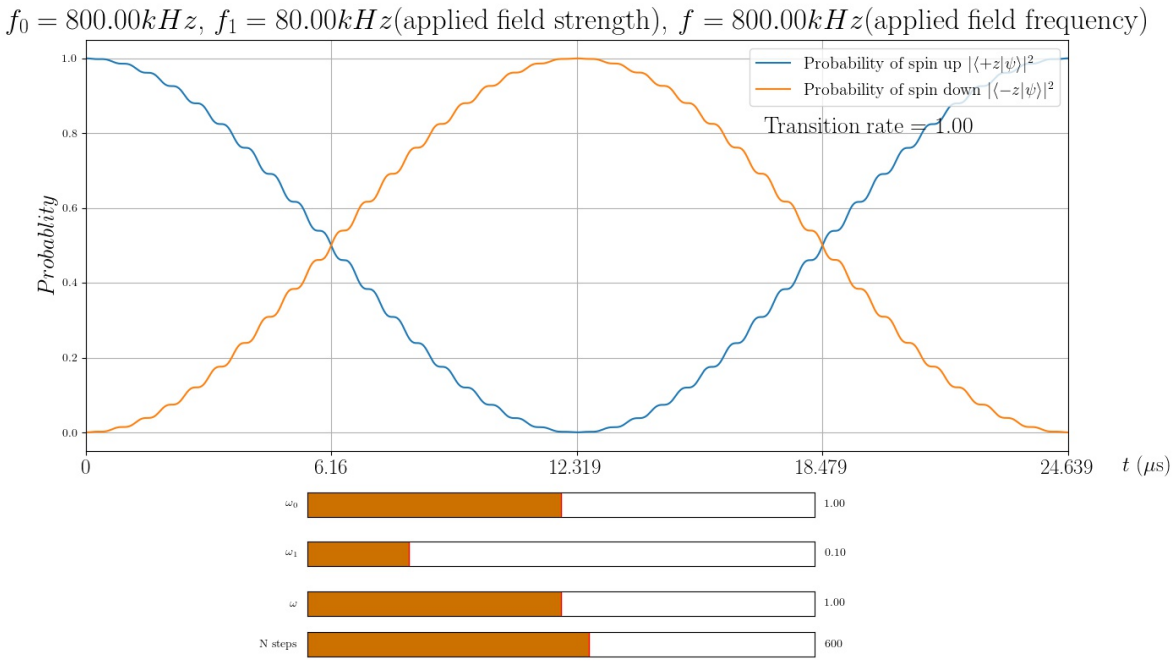
- The code to plot the transition rate as a function of the applied oscillating field frequency is quite similar to the code for plotting the time precession. In order to provide some basis of comparison for our model we also define and plot the expected function of transition rate vs ω based on the formula provided in Figure 4.6 of Townsend.

```
### expected transition_rate vs applied AC field frequency from Townsend Figure 4.6
def trans_rate_exp(w0,w1,w):
w0 = w0*w00
w1 = w1*w00
w = w*w00
y = w1**2/4/((w0-w)**2+w1**2/4)
return y

fig= plt.figure(figsize=(15,13))
ax = fig.add_subplot(111)
fig.subplots_adjust(bottom=0.3, top=0.8)
transition_rate_line, = ax.plot(w_list,rate, label='Transition Probability from
    $|\langle +z|\psi \\rangle|^2$ to $|\langle -z |\psi\\rangle |^2$')
exp_line, = ax.plot(w_list, trans_rate_exp(w0,w1,w_list),label="expectation")

ax.legend(loc='upper right', fontsize='large')
ax.set_xlabel('$\\frac{\omega}{\omega_0}$', fontsize=20)
ax.xaxis.set_label_coords(1.01, -0.01)
ax.set_title('$f_0= %0.2f kHz$, $f_1 = %0.2f kHz$(applied field strength)'%(f0,f1), fontsize=20)
ax.grid(True)
```

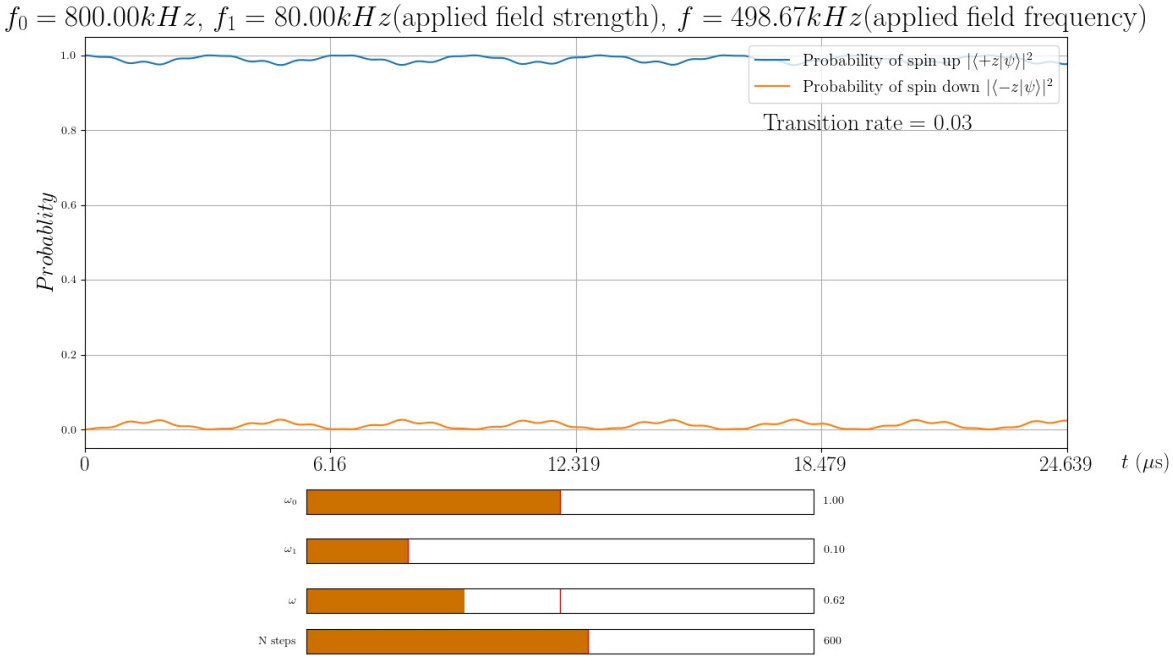
- Add three sliders to make interactive adjustments on ω_0, ω_1 and N, with N as the number of time steps used in solving the differential equation. Next we define an update function to have the plot change under different conditions of ω_0, ω_1 and N.
- In Fig 2 we show the transition rate vs ω function at three different conditions: a) ω_0=1.0, ω_1=0.1; b) ω_0=1.0, ω_1=0.04; c) ω_0=1.0, ω_1=0.42.

4 Model Limitations of transition rate vs ω

We also have found that for $\omega_1 > 0.3$ when $\omega_0 = 1.0$, the calculated probability varies significantly from the expected value provided from Townsend. Further, there appears to be anomalies in our calculated probabilities that show up as sharp dips in the data (see Figure 2.(c)). We suspect that these anomalies come from the errors in solving the differential equations by numerical approximations. However, we have found that increasing number of time steps does not seem to resolve this issue. This leads to the anomalies being rather interesting, and worth investigating in future iterations of this project. One possible solution is first numerically approximating the solutions, then interpolating the scattered data points as a smooth function of time. (See Figure 3).

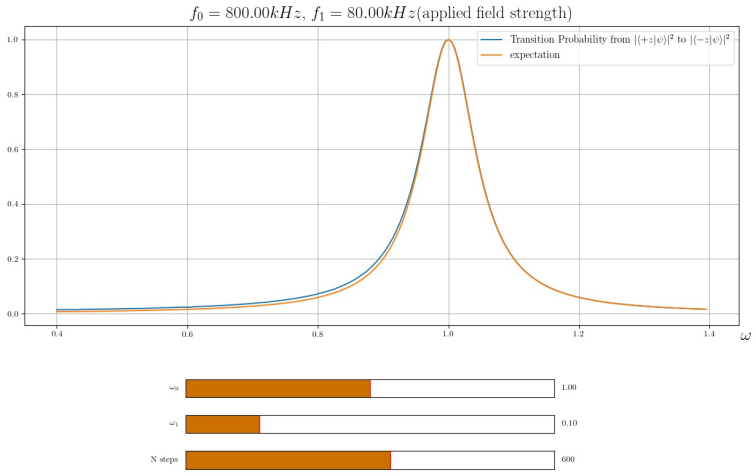


(a) The time procession of angular spin in z axis when ω_0=1.0, $\omega_1 = 0.1$ and $\omega = 1.0$, which correspond to $f_0 = 800$kHz, f_1=80kHz, and f= 800kHz.

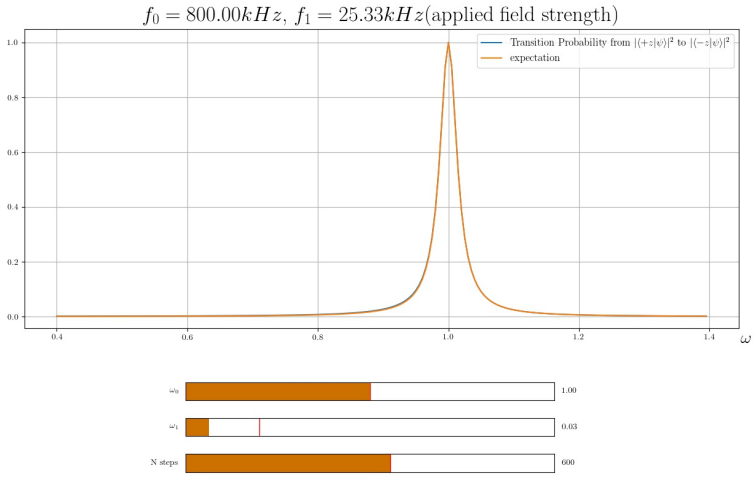


(b) The time procession of angular spin in z axis when ω_0=1.0, $\omega_1 = 0.1$ and $\omega = 0.62$, which correspond to $f_0 = 800$kHz, f_1=80kHz, and f= 498.67kHz.

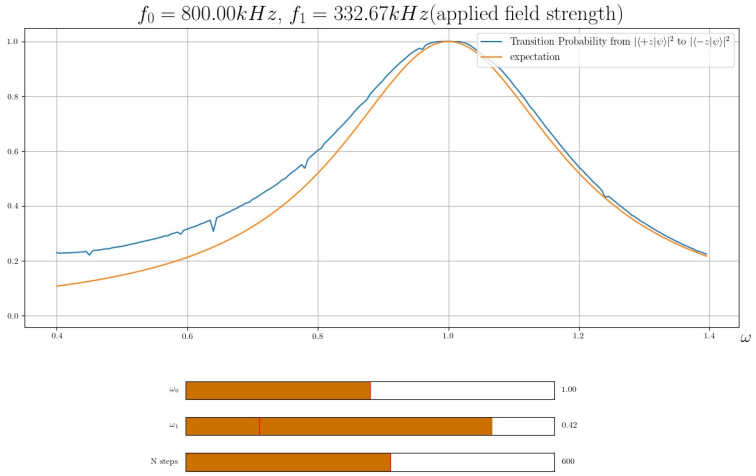
Figure 1: The probabilities measuring $|+z\rangle$ and $|-z\rangle$ for a spin-1/2 particle that is in $|+z\rangle$ at at t = 0 when the oscillating field in the x direction is a) at resonant frequency b) not at resonance. In b), the spin state of the particle does not make a complete transition from $|z\rangle$ to $|-z\rangle$.



(a) The transition rate as a function of ω when ω_0=1.0, ω_1 = 0.1, that is the oscillating field is one-tenth of the constant field, fairly weak.



(b) The transition rate as a function of ω when ω_0=1.0, ω_1 = 0.03, that is the oscillating field is very weak compared to the constant field.



(c) The transition rate as a function of ω when ω_0=1.0, ω_1 = 0.42, that is the oscillating field is significant compared to the constant field.

Figure 2: Transition rate as a function of ω at different conditions of ω_0 and ω_1. a) ω_0=1.0, ω_1=0.1; b) ω_0=1.0, ω_1=0.04; c) ω_0=1.0, ω_1=0.42.

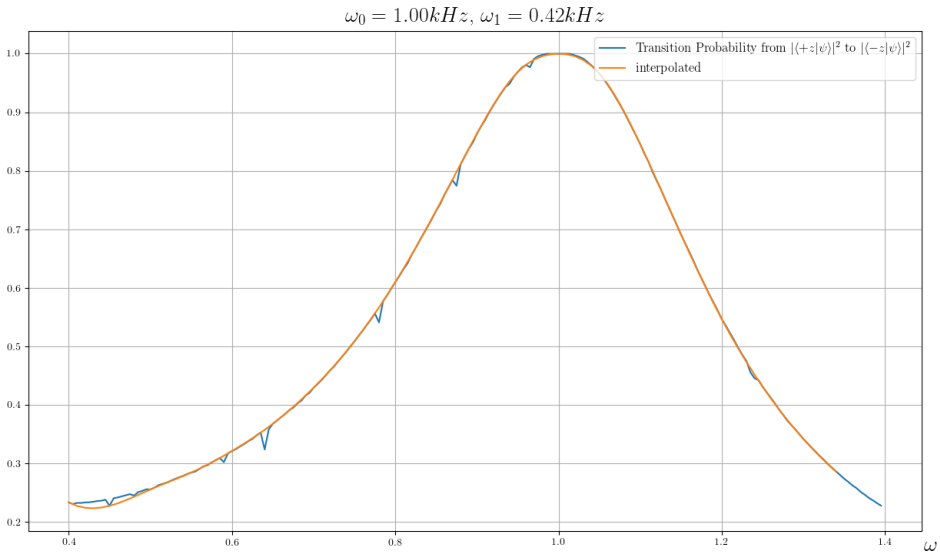


Figure 3: Smoothed function of Transition rate vs ω when ω_0=1.0, ω_1=0.42. This done in interpolate.py