LASED

Release 0.3

Manish

May 18, 2022

CONTENTS:

CHAPTER

ONE

GETTING STARTED

1.1 Getting Started

1.1.1 What is LASED?

LASED stands for Laser-Atom interaction Simulator derived from quantum ElectroDynamics. LASED is a python library which can:

- Calculate the time evolution of an atomic system interacting with a laser.
- Generate the equations of motion of an atom-laser system.
- Rotate an atomic system to a different reference frame.
- Calculate the time evolution of the angular shape of an atomic state.

LASED can simulate any atomic system. The sub-states, angular momenta, spin, and energies of the system are provided by the user to simulate the atomic system.

Laser parameters also need to be specified to build the laser-atom system. These parameters include detuning from the transition frequency, polarisation, and laser intensity/power.

LASED can simulate a Gaussian beam profile and Doppler averaging over the atoms to provide a more accurate model of the atom-laser system. LASED can also simulate the angular shape of an atomic state over time as it is excited by a laser. This is useful in many experiments using atoms and lasers.

1.1.2 Installation

You can easily install LASED by opening up a terminal and running:

```
pip install LASED
```

The source code can be viewed here

1.1.3 Using LASED

Start by going to *Tutorials*.

The first tutorial is a guide on how to simulate one of the simplest excitations of an atom with a laser and shows how to use most of the functionality of LASED.

The second tutorial shows how to simulate decays to other states not in the laser-excitation manifold. It also introduces rotating the system to different reference frames using the Wigner rotation matrix and hence introduces simulating a polarisation angle.

The third tutorial simulates a metastable Helium atom with some real data on what the initial state looks like and how to initialise the system to this state.

Each tutorial after this simulates an increasingly more complex atomic system with hyperfine structure.

CHAPTER

TWO

TUTORIALS

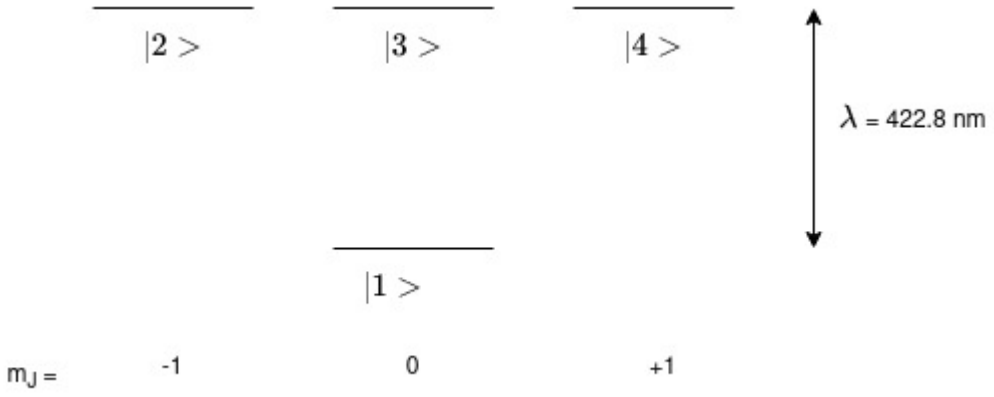
2.1 Tutorial 1: The Basics with Simple Calcium

The system of calcium's ground state excited to its first excited state with an on-resonance laser is one of the simplest systems to model as there is only a ground S-state and an excited P-state with no hyperfine structure. We can verify results already published by Murray 2003 by using the `LASED` package. In this paper the transition between the ground S-state and the first excited singlet P-state is modelled with the laser on-resonance and 200 MHz & 500 MHz detuned from resonance.

First, we'll set up the states. A diagram of the system we are going to model is seen below. The ground S-state is represented by one sub-state lebelled as $|1\rangle$ and the excited P-state is represented by three sub-states labelled as $|2\rangle$, $|3\rangle$, and $|4\rangle$.

The numbers indicate the labelling I have given them. It is convention to label the sub-states in your system by labelling the lowest in energy state first. The sub-state with the lowest m value is labelled as $|1\rangle$ and then the 2nd lowest m value as $|2\rangle$ etc. until you run out of sub-states. Then, the next highest in energy state is labelled.

The wavelength of the difference in energy between these two states is also shown as 422.8 nm.



Now, we'll import the `LASED` library and a plotting library. I use `Plotly` to plot all the figures here but any plotting library can be used.

[1]:
```
import LASED as las
from IPython.display import Image
import plotly.graph_objects as go  # For plotting
import time  # NOT NEEDED: For seeing how quickly the time evolution is calculated
import numpy as np
```

2.1.1 Setting up the System

With LASED you can declare atomic sub-states using the `State` object. These states are used to declare a `LaserAtomSystem` which can then be used to calculate the time evolution of the system.

First, we must declare the system's variables: wavelength of the transition, lifetime of the excited state, isospin, etc.

Note: LASED has a timescale of nanoseconds so all times will be input in nanoseconds. If I want a lifetime of 4.6e-9 s then I have to input 4.6 into my `LaserAtomSystem` object.

[2]:
```
wavelength_ca = 422.8e-9  # wavelength of Ca transition in metres
tau_ca = 4.6 # lifetime in nanoseconds
I_ca = 0  # Isospin of calcium
```

Create the `State` objects by providing the `label` of the sub-state with the convention as above. **The system may not be modelled correctly if you do not stick to this labelling convention**.

Each sub-state must have a relative angular frequency `w` associated with it. This angular frequency is related to the energy as usual $E = \hbar w$. All frequencies should be in gigaradians per second. The energies are *relative* so you just have to set a zero point and then all other sub-states have energies relative to this point. I have set the zero-point as the energy of the ground sub-state.

Each sub-state must be labelled with its corresponding quantum numbers: orbital angular momentum `L`, spin `S`, projection of total angular momentum `m`. The isospin `I` is assumed to be zero if not specified. The total angular momentum (without isospin) can be specified with keyword `J` and is calculated as `J=L+S` if not specified. The total angular momentum with isospin can be specified with keyword `F` and if not specifed is calculated as `F=J+I`.

To create the ground or excited state you must insert each sub-state in an ordered list starting with the smallest labelled state to the highest labelled state.

[3]:
```
# Calculate angular frequency of the transition
w_e = las.angularFreq(wavelength_ca)  # Converted to angular frequency in Grad/s

# Create states
s1 = las.State(label = 1, w = 0, m = 0, L = 0, S = 0)
s2 = las.State(label = 2, w = w_e, m = -1, L = 1, S = 0)
s3 = las.State(label = 3, w = w_e, m = 0, L = 1, S = 0)
s4 = las.State(label = 4, w = w_e, m = 1, L = 1, S = 0)
print(s3)

# Create ground and excited states list
G_ca = [s1]
E_ca = [s2, s3, s4]
```

```
State(label = 3, w = 4455183.460995396, m = 0, L = 1, J = 1, I = 0, F = 1)
```

Declare the laser parameters. The intensity of the laser `laser_intensity` must be in units of mW/mm^2.

The polarisation of the laser is defined by keyword `Q` and is either right-hand circular (σ^+) with a +1, left-hand circular (σ^-) with a -1, and linear (π) polarisation with the polarisation axis defined along the axis with a 0 (angle of polarisation of zero degrees). `Q` is defined as a list of any of these three values. If `Q` is 0 then it is defined that the linear polarisation is aligned with the x-axis (if the z-axis is the quantisation axis). This is known as the collision frame in scattering experiments.

The `detuning` of the laser away from resonance can be specified as well. If no detuning is given then the laser is assumed to be on-resonance i.e. `detuning = 0`

Note on using more than one polarisation: If more than one value is in the list then the laser is defined as having simulataneous polarisations of the same laser acting upon the atom. This can be possible if working in the natural frame

when the laser is travelling along the direction of the quantisation axis and the polarisation is linear in the collision frame. When working in this natural frame `Q = [-1, 1]`. You must normalise the Rabi frequencies if you do this by a normalisation constant by giving the `LaserAtomSystem` an attribute called `rabi_scaling`. In this case of two simultaneous Rabi frequencies, the correct time evolution would be scaled by `1/np.sqrt(2)`. If `n` simultaneous polarisations are used then it must be scaled by `1/np.sqrt(n)`. You must also apply the `rabi_factors` attribute as the simultaneous combination of LHC and RHC Rabi frequencies is only equal to a linear exciation if the total Rabi frequency is $1/\sqrt{(2)}(\Omega^{-1} - \Omega^{+1})$ so the RHC Rabi frequency must be multiplied by -1. This is achieved by setting the `rabi_factors` attribute to `rabi_factors = [1, -1]` if using a `Q` as stated above. Each element of `Q` is multiplied by the corresponding element in `rabi_factors`.

[4]:
```
intensity_ca = 100 # mW/mm^-2
Q_ca = [0]
detuning = 0.2*2*np.pi  # detuning here is 200 MHz in Grad/s
detuning2 = 0.5*2*np.pi  # detuning here is 500 MHz in Grad/s
```

The `time` over which the simulation run must be sepcified. It must be specified with a list with every discrete time step (in nanoseconds) in it. Numpy's `linspace` is handy for this task.

[5]:
```
# Simulation parameters
start_time = 0
stop_time = 50 # in ns
time_steps = 501
time_ca = np.linspace(start_time, stop_time, time_steps)
```

Create a `LaserAtomSystem` object by using the variables stated above. Three system's are created here for different detunings.

[6]:
```
calcium_system = las.LaserAtomSystem(E_ca, G_ca, tau_ca, Q_ca, wavelength_ca,
                                     laser_intensity = intensity_ca)
calcium_system200MHzdetuned = las.LaserAtomSystem(E_ca, G_ca, tau_ca, Q_ca, wavelength_
↪ca,
                                     laser_intensity = intensity_ca)
calcium_system500MHzdetuned = las.LaserAtomSystem(E_ca, G_ca, tau_ca, Q_ca, wavelength_
↪ca,
                                     laser_intensity = intensity_ca)
```

2.1.2 Time Evolution of the System

Perform a `timeEvolution` of each system. The `pretty_print_eq` keyword is used here to print out the system's equations of motion using `Sympy`. There are many other keywords which can be used with `timeEvolution` including averaging over the doppler profile of the atoms, averaging the Gaussian laser beam profile, and numerically printing the equations.

I have just timed this piece of code to see how long it takes.

Note: An initial condition density matrix can be specified at t = 0 to evolve. If none is stated then **all ground sub-states are populated equally with no coherence between sub-states**.

[7]:
```
tic = time.perf_counter()
calcium_system.timeEvolution(time_ca,
                             pretty_print_eq = True)
calcium_system200MHzdetuned.timeEvolution(time_ca,
                             detuning = detuning)
calcium_system500MHzdetuned.timeEvolution(time_ca,
```

(continues on next page)

(continued from previous page)

```
                                detuning = detuning2)
toc = time.perf_counter()
print(f"The code finished in {toc-tic:0.4f} seconds")
```

```
Populating ground states equally as the initial condition.
```

$$\dot{\rho}_{11} = i\rho_{13}\Omega(3,1,0) + \frac{1.0\rho_{22}}{\tau} - i\rho_{31}\Omega(3,1,0) + \frac{1.0\rho_{33}}{\tau} + \frac{1.0\rho_{44}}{\tau}$$

$$\dot{\rho}_{22} = -\frac{\rho_{22}}{\tau}$$

$$\dot{\rho}_{23} = i\rho_{21}\Omega(3,1,0) - \frac{\rho_{23}}{\tau}$$

$$\dot{\rho}_{24} = -\frac{\rho_{24}}{\tau}$$

$$\dot{\rho}_{32} = -i\rho_{12}\Omega(3,1,0) - \frac{\rho_{32}}{\tau}$$

$$\dot{\rho}_{33} = -i\rho_{13}\Omega(3,1,0) + i\rho_{31}\Omega(3,1,0) - \frac{\rho_{33}}{\tau}$$

$$\dot{\rho}_{34} = -i\rho_{14}\Omega(3,1,0) - \frac{\rho_{34}}{\tau}$$

$$\dot{\rho}_{42} = -\frac{\rho_{42}}{\tau}$$

$$\dot{\rho}_{43} = i\rho_{41}\Omega(3,1,0) - \frac{\rho_{43}}{\tau}$$

$$\dot{\rho}_{44} = -\frac{\rho_{44}}{\tau}$$

$$\dot{\rho}_{12} = -\frac{\rho_{12}}{2\tau} - i\rho_{32}\Omega(3,1,0)$$

$$\dot{\rho}_{13} = -i\rho_{11}\Omega(3,1,0) - \frac{\rho_{13}}{2\tau} - i\rho_{33}\Omega(3,1,0)$$

$$\dot{\rho}_{14} = -\frac{\rho_{14}}{2\tau} - i\rho_{34}\Omega(3,1,0)$$

$$\dot{\rho}_{21} = -\frac{\rho_{21}}{2\tau} + i\rho_{23}\Omega(3,1,0)$$

$$\dot{\rho}_{31} = i\rho_{11}\Omega(3,1,0) - \frac{\rho_{31}}{2\tau} + i\rho_{33}\Omega(3,1,0)$$

$$\dot{\rho}_{41} = -\frac{\rho_{41}}{2\tau} + i\rho_{43}\Omega(3,1,0)$$

```
Populating ground states equally as the initial condition.
Populating ground states equally as the initial condition.
The code finished in 4.8326 seconds
```

2.1.3 Saving and Plotting

We can save the data to a .csv file and now plot the data generated to see the time evolution.

To save to csv use the `saveToCSV("filename")` function on your `LaserAtomSystem` object.

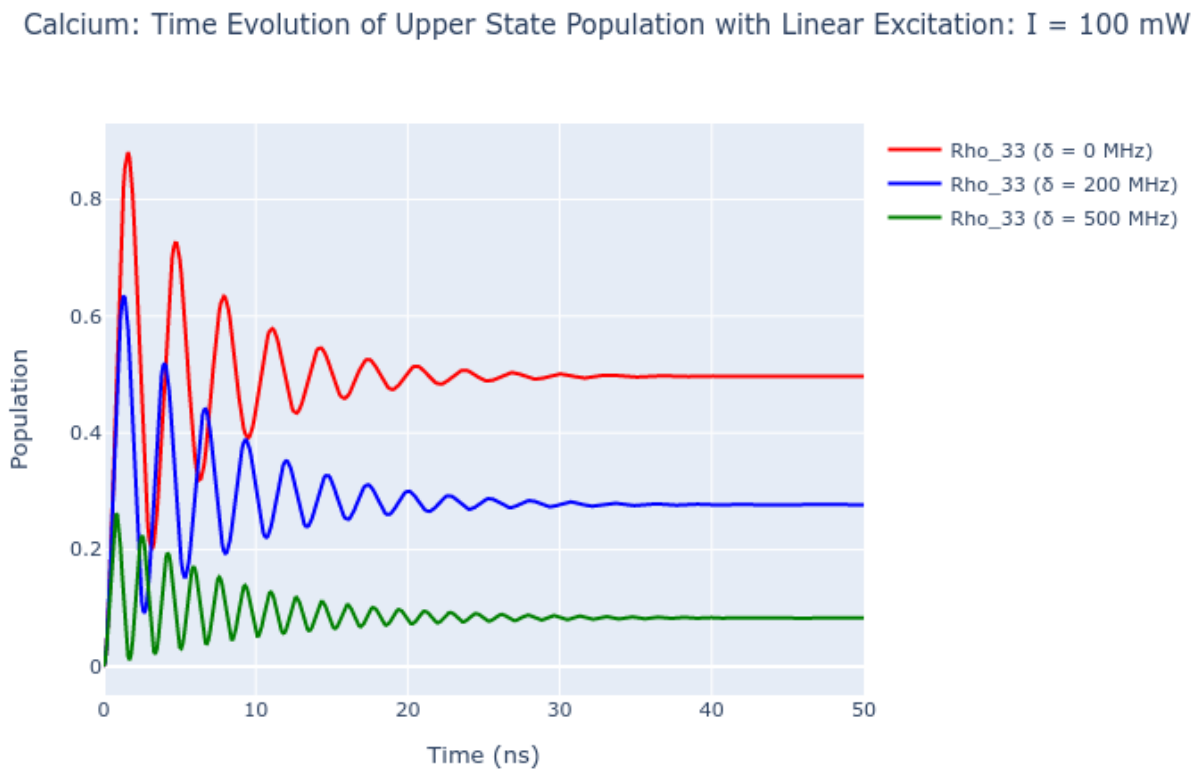
[8]:
```
calcium_system.saveToCSV("SavedData/SimpleCalciumNoDetuning.csv")
calcium_system200MHzdetuned.saveToCSV("SavedData/SimpleCalcium200MHzDetuning.csv")
calcium_system500MHzdetuned.saveToCSV("SavedData/SimpleCalcium500MHzDetuning.csv")
```

Now, we can plot the evolution of Calcium with no detuning, 200 MHz detuning, and 500 MHz detuning.

To access the density matrix elements over the time evolution use the function `Rho_t(e, g)` on the `LaserAtomSystem`. This gives a list of the element ρ_{eg} for each interval in `time`. Elements of `Rho_t()` and `Rho_0()`can also be accessed by addressing the labels of the states directly e.g. if a user wanted $\rho_{13}(t)$ the would use `Rho_t(1, 3)`. Each element of the density matrix is `complex` so the real part is taken by using the `abs()` function.

[9]:
```
# Using state-based indexing to get rho_33(t)
rho_33 = [abs(rho) for rho in calcium_system.Rho_t(s3, s3)]
# Using label-based indexing to get rho_33(t)
rho_33_200MHzdetuned = [abs(rho) for rho in calcium_system200MHzdetuned.Rho_t(3, 3)]
rho_33_500MHzdetuned = [abs(rho) for rho in calcium_system500MHzdetuned.Rho_t(3, 3)]

fig_ca = go.Figure()
fig_ca.add_trace(go.Scatter(x = time_ca,
                                y = rho_33,
                                mode = 'lines',
                                   name = "Rho_33 ( = 0 MHz)",
                                   marker = dict(
                               color = 'red',
                               symbol = 'circle',
                           )))
fig_ca.add_trace(go.Scatter(x = time_ca,
                                y = rho_33_200MHzdetuned,
                                mode = 'lines',
                                   name = "Rho_33 ( = 200 MHz)",
                                   marker = dict(
                               color = 'blue',
                               symbol = 'circle',
                           )))
fig_ca.add_trace(go.Scatter(x = time_ca,
                                y = rho_33_500MHzdetuned,
                                mode = 'lines',
                                   name = "Rho_33 ( = 500 MHz)",
                                   marker = dict(
                               color = 'green',
                               symbol = 'circle',
                           )))

fig_ca.update_layout(title = "Calcium: Time Evolution of Upper State Population with␣
↪Linear Excitation: I = 100 mW",
                  xaxis_title = "Time (ns)",
                  yaxis_title = "Population",
                 font = dict(
                     size = 11))

fig_ca.write_image("SavedPlots/tutorial1-ca.png")
Image("SavedPlots/tutorial1-ca.png")
```

[9]:



2.1.4 Elliptical Polarisation

To excite an atomic system with ellipticaly polarised light use the relation that elliptically polarisaed light can be composed of right-hand circular (RHC) σ^+ and left-hand circular (LHC) σ^- light in varying weights. So, elliptical light can be described as:

$$\epsilon = \frac{1}{\sqrt{L^2 + R^2}}(L\sigma^- + R\sigma^+)$$

where R and L are weights denoting how elliptically polarised the light is in either right or left-handed direction. If $R = L$ then the light is just linearly polarised. The ratio of L to R determines how elliptically polarised the light is.

The Rabi frequency for elliptically polarisation follows from the expression above and is the superposition of the Rabi frequency for RHC and LHC light with differing weights:

$$\Omega^\epsilon = \frac{1}{\sqrt{L^2 + R^2}}(L\Omega^{-1} - R\Omega^{+1})$$

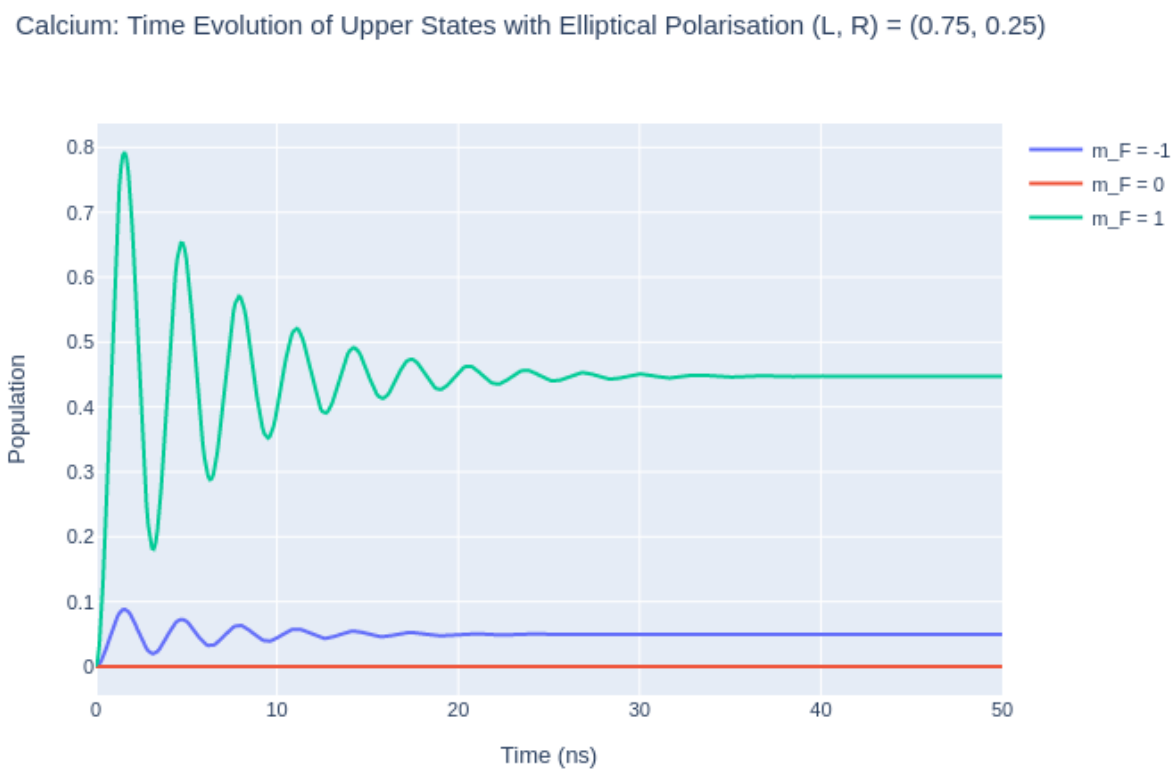
To encode this in `LASED` the polarisation key word `Q = [1, -1]` with `rabi_factors = [L, R]` and `rabi_scaling = 1/np.sqrt(L*L+R*R)` for normalisation. With this description of the `LaserAtomSystem` the laser beam's direction of travel is down the quantisation axis. As an exmaple, the simple calcium system will be modelled with different weights: one system with more a more LHC ellipse and the other with a more RHC ellipse.

For the first system, use values of L = 0.75 and R = 0.25.

[10]:
```
# Declare polarisation and Rabi parameters
Q_ellipse = [1, -1]
rabi_factors_lhc = [0.75, 0.25]
rabi_scaling_lhc = 1/np.sqrt(0.75*0.75+0.25*0.25)

# Create laser-atom system
calcium_system_lhc = las.LaserAtomSystem(E_ca, G_ca, tau_ca, Q_ellipse, wavelength_ca,
                                 laser_intensity = intensity_ca, rabi_scaling = rabi_
↪scaling_lhc,
                                    rabi_factors = rabi_factors_lhc)

# Time eveolve system
calcium_system_lhc.timeEvolution(time_ca)

# Plot all excited states
rho_e = [[abs(rho) for rho in calcium_system_lhc.Rho_t(s, s)] for s in E_ca]
fig_ca_lhc = go.Figure()
for i, rho in enumerate(rho_e):
    fig_ca_lhc.add_trace(go.Scatter(x = time_ca,
                                y = rho,
                                name = f"m_F = {E_ca[i].m}",
                               mode = 'lines'))

fig_ca_lhc.update_layout(title = "Calcium: Time Evolution of Upper States with␣
↪Elliptical Polarisation (L, R) = (0.75, 0.25)",
                 xaxis_title = "Time (ns)",
                 yaxis_title = "Population",
                font = dict(
                    size = 11))
fig_ca_lhc.write_image("SavedPlots/tutorial1-ellipselhc.png")
Image("SavedPlots/tutorial1-ellipselhc.png")
```

```
Populating ground states equally as the initial condition.
```

[10]:



Now, model a system with a slightly more RHC elliptcal polarisation.

[11]:
```
# Rabi parameters
rabi_factors_rhc = [0.45, 0.55]
rabi_scaling_rhc = 1/np.sqrt(0.45*0.45+0.55*0.55)

# Create laser-atom system
calcium_system_rhc = las.LaserAtomSystem(E_ca, G_ca, tau_ca, Q_ellipse, wavelength_ca,
                                 laser_intensity = intensity_ca, rabi_scaling = rabi_
↪scaling_rhc,
                                    rabi_factors = rabi_factors_rhc)

# Time eveolve system
calcium_system_rhc.timeEvolution(time_ca)

# Plot all excited states
rho_e = [[abs(rho) for rho in calcium_system_rhc.Rho_t(s, s)] for s in E_ca]
fig_ca_rhc = go.Figure()
for i, rho in enumerate(rho_e):
    fig_ca_rhc.add_trace(go.Scatter(x = time_ca,
                                    y = rho,
                                    name = f"m_F = {E_ca[i].m}",
                                   mode = 'lines'))
```

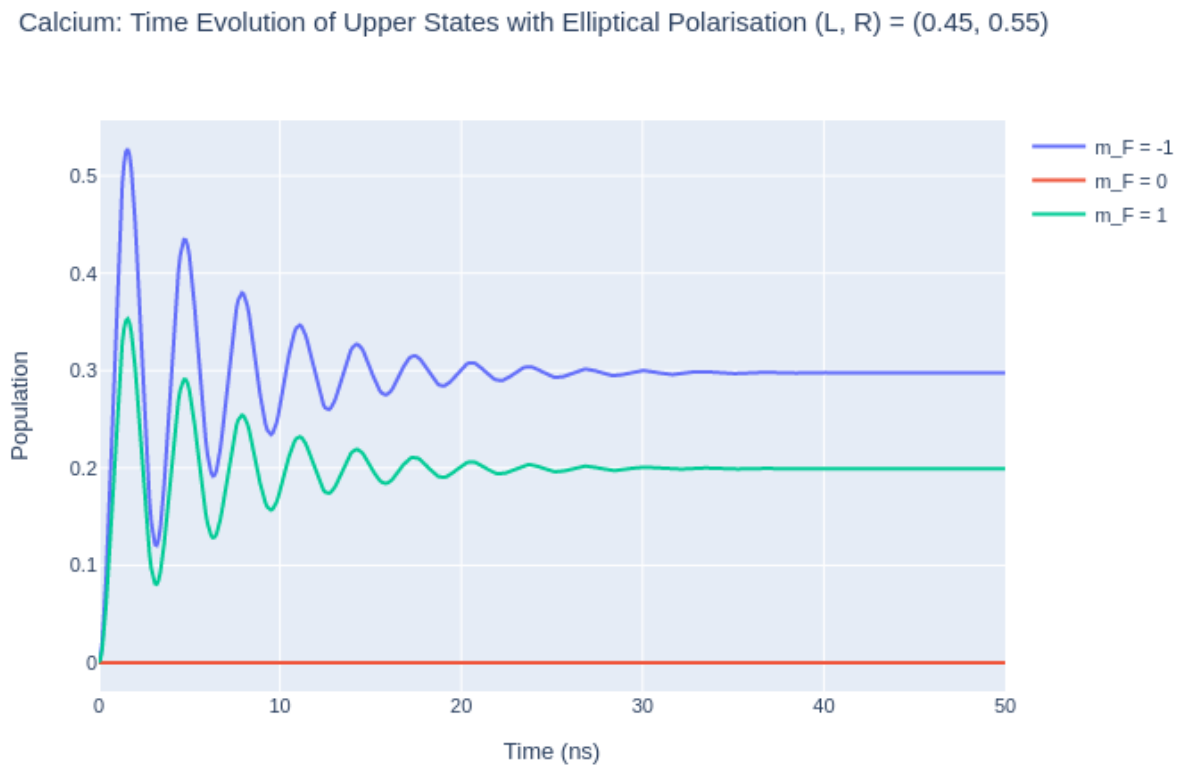
(continues on next page)

(continued from previous page)

```
fig_ca_rhc.update_layout(title = "Calcium: Time Evolution of Upper States with␣
↪Elliptical Polarisation (L, R) = (0.45, 0.55)",
                 xaxis_title = "Time (ns)",
                 yaxis_title = "Population",
                font = dict(
                    size = 11))
fig_ca_rhc.write_image("SavedPlots/tutorial1-ellipserhc.png")
Image("SavedPlots/tutorial1-ellipserhc.png")
```

```
Populating ground states equally as the initial condition.
```

[11]:



If L = R then this would be the same as a linear polarisation in the reference frame where the laser beam is travelling down the quantisation axis. This is called the natural frame. To put this in the collision frame, where the polarisation vector is along the quantisation axis, then rotations must be used.

2.1.5 Setting the Initial Conditions

There are two ways to specify the initial condition ρ_0 in LASED: one way is to set the entire density matrix for the system in one function using the object's variable `rho_0` directly and the other way is to set the initial condition for each sub-state equally using the function `setRho_0(s1,s2, value)` where `s1` and `s2` are state objects specifying the density matrix element which is initialised to `value`. In the example below calcium will be initialised with two different conditions.

The sum of the populations of the density matrix must be equal to 1 and the coherences must be set so that $\rho_{eg} = \rho^*_{ge}$.

[12]:
```
calcium_system_init1 = las.LaserAtomSystem(E_ca, G_ca, tau_ca, Q_ca, wavelength_ca,
                                   laser_intensity = intensity_ca)

# Set the populations of the ground state 1 and excited state 3 to 0.5 each with no␣
↪coherences between them
calcium_system_init1.setRho_0(s1, s1, 0.2)
calcium_system_init1.setRho_0(s3, s3, 0.8)
# Perform time evolution
calcium_system_init1.timeEvolution(time_ca)
# Plot the excited population
rho_33_init1 = [abs(rho) for rho in calcium_system_init1.Rho_t(s3, s3)]
```

Now, we are going to build a flattened density matrix `rho_0` and input this straight into the system. To build the initial density matrix we must input the density matrix elements in the correct position. To do this, use the `index(e, g, n)` function to find the index which the density matrix element will sit in a flattened density matrix.

[13]:
```
calcium_system_init2 = las.LaserAtomSystem(E_ca, G_ca, tau_ca, Q_ca, wavelength_ca,
                                   laser_intensity = intensity_ca)

# Set out the density matrix elements to be initialised
rho_11_t0 = 0.2
rho_33_t0 = 0.8
rho_13_t0 = 0.5+0.5j  # These coherences can be complex!
rho_31_t0 = 0.5-0.5j

# Now build a flattened density matrix with these conditions
n = 4 # nunber of sub-states in the Calcium system
rho_0 = np.zeros((n*n, 1), dtype = complex) # Make an empty 2D array with only one column
rho_0[las.index(s1, s1, n), 0] = rho_11_t0
rho_0[las.index(s3, s3, n), 0] = rho_33_t0
rho_0[las.index(s1, s3, n), 0] = rho_13_t0
rho_0[las.index(s3, s1, n), 0] = rho_31_t0
print(rho_0)
```

```
[[0.2+0.j ]
 [0. +0.j ]
 [0.5+0.5j]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]
 [0.5-0.5j]
 [0. +0.j ]
 [0.8+0.j ]
```

(continues on next page)

(continued from previous page)

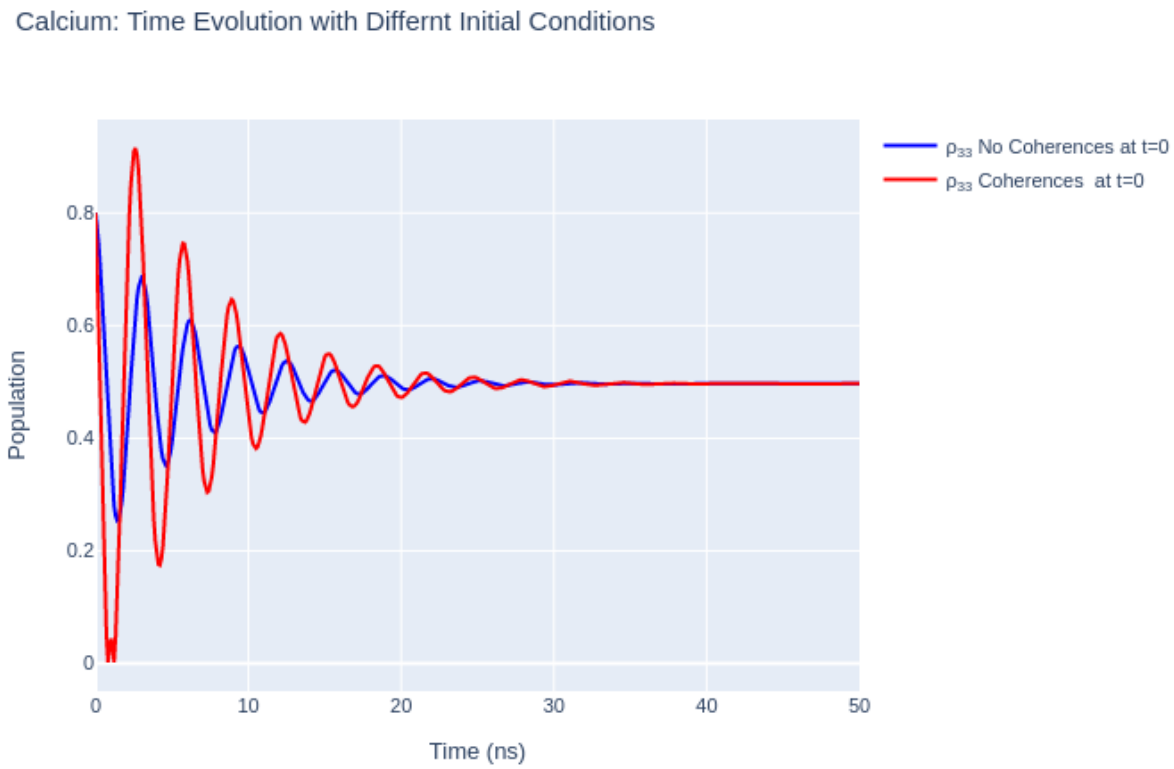
```
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]]
```

Above is the correct form of `rho_0` we want to input. We can directly input this into the `LaserAtomSystem` object by using the key identifier `LaserAtomSystem.rho_0`.

Note: be careful here as there is no safety conditions which have to be met to input this variable. Make sure that this is the correct size and format of the flattened density matrix.

[14]:
```
calcium_system_init2.rho_0 = rho_0  # Set the variable directly

# Perform time evolution
calcium_system_init2.timeEvolution(time_ca)
# Plot the excited population
rho_33_init2 = [abs(rho) for rho in calcium_system_init2.Rho_t(s3, s3)]
fig_ca_init = go.Figure(go.Scatter(x = time_ca,
                                   y = rho_33_init1,
                                   mode = 'lines',
                                      name = "<sub>33</sub> No Coherences at t=0",
                                      marker = dict(
                                  color = 'blue',
                                  symbol = 'cross'
                              )))
fig_ca_init.add_trace(go.Scatter(x = time_ca,
                                   y = rho_33_init2,
                                   mode = 'lines',
                                      name = "<sub>33</sub> Coherences\n at t=0",
                                      marker = dict(
                                  color = 'red',
                                  symbol = 'circle'
                              )))
fig_ca_init.update_layout(title = f"Calcium: Time Evolution with Differnt Initial␣
↪Conditions",
                  xaxis_title = "Time (ns)",
                  yaxis_title = "Population",
                 font = dict(
                     size = 11))

fig_ca_init.write_image("SavedPlots/tutorial1-ca-init.png")
Image("SavedPlots/tutorial1-ca-init.png")
```

[14]:



2.1.6 Gaussian Laser Beam Profiles

A laser beam usually does not have a flat beam profile (known as a "top-hat" distribution) in intensity. As the beam has spatial variation in intensity the atoms being excited experience a non-uniform time evolution. To model the effects of the beam profile the beam can be split up into regions of approximate uniform intensity and each spatial portion of the beam is used to time-evolve a part of the system being illuminated. Then, each part of the system is summed together and normalised which results in the entire system being modelled.

`LASED` supports the modelling of a Gaussian TEM_{00} laser beam profile. The 2D standard deviation of the Gaussian must be declared with keyword `r_sigma` when performing the `timeEvolution()` of the `LaserAtomSystem`. The number of portions which the beam is split into must be chosen as well. This is declared with the keyword `n_beam_averaging` when using `timeEvolution()`. The Gaussian averaging is applied to the time evolution of the system only if it is declared that `beam_profile_averaging = True` inside the `timeEvolution()` function. If these are left out then a "top-hat" distribution of laser intensity is assumed. Also, to use the Gaussian avergaing over the beam profile, the keyword `laser_power` must be defined in the `LaserAtomSystem`. This is the total power which the laser delivers as opposed to the intensity over a mm^2.

Below, the laser parameters are declared for this system and the simple calcium system is modelled using these parameters.

Note: If using this averaging the model will loop over the time evolution with the number defined in `n_intensity` so the model will be much slower if a larger number is input. The larger number also results in a more accurate representation of the beam profile. Usually, a `n_intensity` of around 50 is enough for most cases.

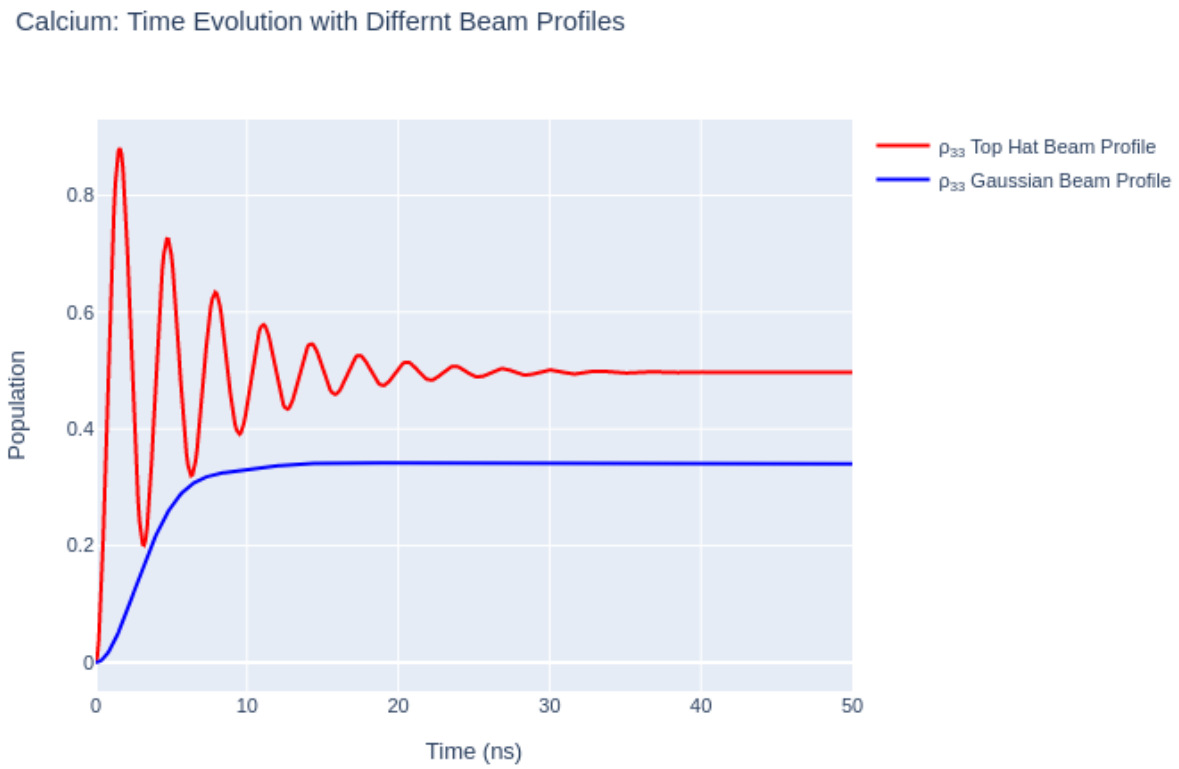
[15]:
```
# Laser parameters
laser_power = 100 # laser intensity in mW
r_sigma = 0.75  # radial distance to 2D standard deviation in mm
n_intensity = 20

calcium_system_gaussian = las.LaserAtomSystem(E_ca, G_ca, tau_ca, Q_ca, wavelength_ca,
                                  laser_power = laser_power)
calcium_system_gaussian.timeEvolution(time_ca,
                              r_sigma = r_sigma,
                              n_beam_averaging = n_intensity,
                              beam_profile_averaging = True)
# Plot the excited population
rho_33_gaussian = [abs(rho) for rho in calcium_system_gaussian.Rho_t(s3, s3)]
```

```
Populating ground states equally as the initial condition.
```

Now, we can plot the result and compare to the time evolution when excited with a "top-hat" beam profile.

[16]:
```
fig_ca_gauss = go.Figure()
fig_ca_gauss.add_trace(go.Scatter(x = time_ca,
                                  y = rho_33,
                                  mode = 'lines',
                                 name = "<sub>33</sub> Top Hat Beam Profile",
                                 marker = dict(
                                     color = 'red',
                                     symbol = 'circle'
                              )))
fig_ca_gauss.add_trace(go.Scatter(x = time_ca,
                                  y = rho_33_gaussian,
                                  mode = "lines",
                                  name = "<sub>33</sub> Gaussian Beam Profile",
                                  marker = dict(
                                      color = "blue",
                                      symbol = "x"
                                  )))
fig_ca_gauss.update_layout(title = f"Calcium: Time Evolution with Differnt Beam Profiles
↪",
                  xaxis_title = "Time (ns)",
                  yaxis_title = "Population",
                 font = dict(
                     size = 11))
fig_ca_gauss.write_image("SavedPlots/tutorial1-ca-gaussian.png")
Image("SavedPlots/tutorial1-ca-gaussian.png")
```

[16]:



2.1.7 Doppler Detuning from the Atomic Velocity Profile

When using LASED the atoms being excited are usually defined as being stationary unless specified. If the atoms are not stationary and have some velocity with respect to the laser beam then the frequency of the laser is detuned from resonance due to the fixed velocity. In experiments an atomic beam is sometimes used to provide the atoms to some interaction region where the laser-excitation takes place. If a velocity component is in (or opposite to) the direction of the laser beam then detuning occurs. The velocity component can be specified using the `atomic_velocity` keyword in the `timeEvolution()`. This is specified in units of m/s in the direction of the laser beam. If the direction is opposite to this then the `atomic_velocity` is negative.

Detuning can also occur due to the Maxwell-Boltzmann distribution of atomic velocities. This results in a Gaussian detuning profile. This can be modelled by splitting the detuning due to the velocity distribution of atoms into uniform sections and time-evolving the system with these uniform detunings and then summing up the time evolution for each detuning and normalising. The detuning due to this Doppler broadening can be modelled in LASED by defining a `doppler_width` in Grad/s in `timeEvolution()` and a list with all the detunings to be used for the averaging process called `doppler_detunings`. The more elements in `doppler_detunings` the more the time evolution of the system is calculated and the more time it will take to model the system.

[17]:
```
# Model the atomic velocity introducing a Doppler shift
atomic_velocity = 50  # Velocity component of atoms in direction of laser beam in m/s
# Set up the system
calcium_system_atomic_velocity = las.LaserAtomSystem(E_ca, G_ca, tau_ca, Q_ca,␣
↪wavelength_ca,
```

(continues on next page)

(continued from previous page)

```
                                    laser_intensity = intensity_ca)
# Perform time evolution
calcium_system_atomic_velocity.timeEvolution(time_ca, atomic_velocity = atomic_velocity)
# Plot the excited state population
rho_33_atomic_velocity = [abs(rho) for rho in calcium_system_atomic_velocity.Rho_t(s3,␣
↪s3)]
```

```
Populating ground states equally as the initial condition.
```

Detuning can also occur due to the Maxwell-Boltzmann distribution of atomic velocities. This results in a Gaussian detuning profile. This can be modelled by splitting the detuning due to the velocity distribution of atoms into uniform sections and time-evolving the system with these uniform detunings and then summing up the time evolution for each detuning and normalising. The detuning due to this Doppler broadening can be modelled in `LASED` by defining a `doppler_width` in Grad/s in `timeEvolution()` and a list with all the detunings to be used for the averaging process called `doppler_detunings`. The more elements in `doppler_detunings` the more the time evolution of the system is calculated and the more time it will take to model the system. Then, use the statement `doppler_averaging = True` in the `timeEvolution()` function.

[18]:
```
# Declare the Doppler profile parameters
doppler_width = 0.3*2*np.pi # doppler width here is 300 MHz but have to convert it into␣
↪Grad/s so multiply by 2*PI and scale
delta_upper = 3*doppler_width
delta_lower = -3*doppler_width
doppler_steps = 30
doppler_detunings = np.linspace(delta_lower, delta_upper, doppler_steps)
# Set up the system
calcium_system_doppler = las.LaserAtomSystem(E_ca, G_ca, tau_ca, Q_ca, wavelength_ca,
                                    laser_intensity = intensity_ca)
# Perform time evolution
calcium_system_doppler.timeEvolution(time_ca,
                             doppler_width = doppler_width,
                             doppler_detunings = doppler_detunings,
                             doppler_averaging = True)
# Plot the excited state population
rho_33_doppler = [abs(rho) for rho in calcium_system_doppler.Rho_t(s3, s3)]
```

```
Populating ground states equally as the initial condition.
```

Plot the results to compare.

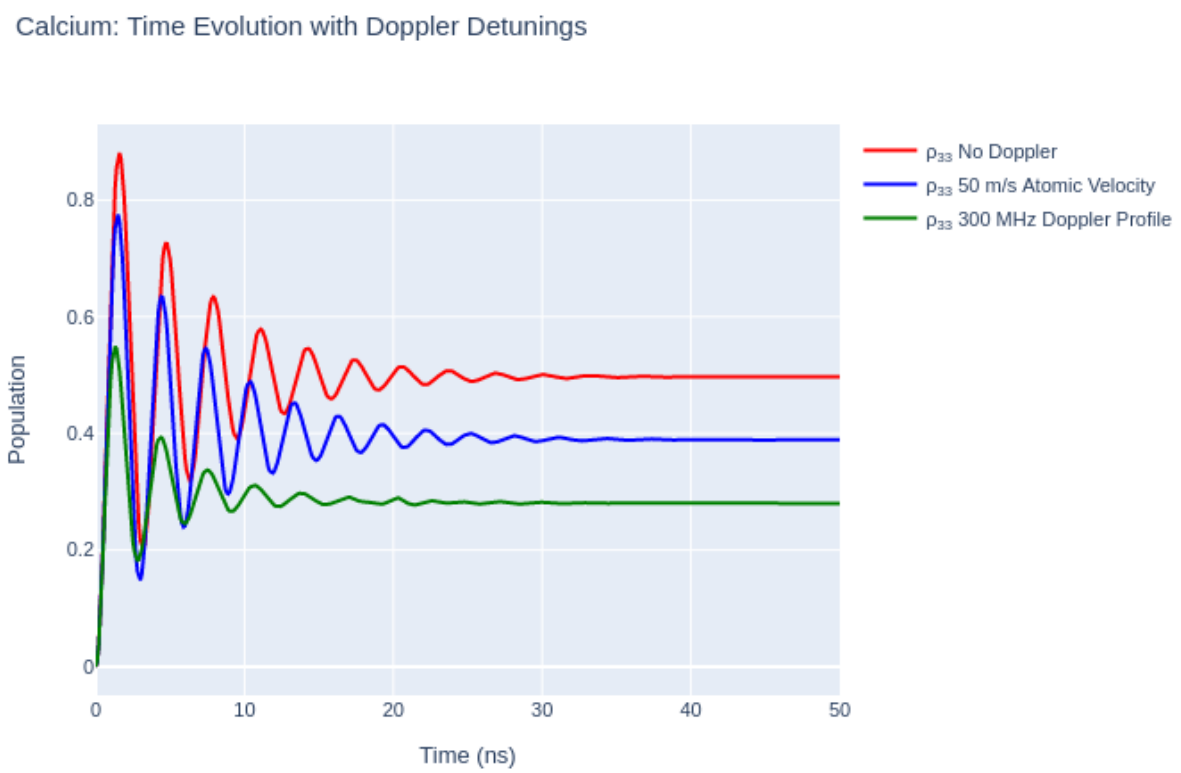
[19]:
```
fig_ca_doppler = go.Figure()
fig_ca_doppler.add_trace(go.Scatter(x = time_ca,
                                    y = rho_33,
                                    mode = 'lines',
                                   name = "<sub>33</sub> No Doppler",
                                   marker = dict(
                                       color = 'red',
                                       symbol = 'circle'
                             )))
fig_ca_doppler.add_trace(go.Scatter(x = time_ca,
                                    y = rho_33_atomic_velocity,
                                    mode = "lines",
                                    name = "<sub>33</sub> 50 m/s Atomic Velocity",
```

(continues on next page)

(continued from previous page)

```
                                    marker = dict(
                                        color = "blue",
                                        symbol = "x"
                                    )))
fig_ca_doppler.add_trace(go.Scatter(x = time_ca,
                                    y = rho_33_doppler,
                                    mode = "lines",
                                    name = "<sub>33</sub> 300 MHz Doppler Profile",
                                    marker = dict(
                                        color = "green",
                                        symbol = "x"
                                    )))
fig_ca_doppler.update_layout(title = f"Calcium: Time Evolution with Doppler Detunings",
                  xaxis_title = "Time (ns)",
                  yaxis_title = "Population",
                 font = dict(
                     size = 11))
fig_ca_doppler.write_image("SavedPlots/tutorial1-ca-doppler.png")
Image("SavedPlots/tutorial1-ca-doppler.png")
```

[19]:



2.1.8 Doppler Detuning and Gaussian Beam Profile

In LASED Doppler profile and Gaussian beam averaging can be modelled in the same system.

Note: When using *both* Doppler and Gaussian beam averaging the number of times the system is time evolved will be `n_intensity` multiplied by the number of elements in `doppler_detunings`.

[20]:
```
# Set up system
calcium_system_gauss_and_dopp = las.LaserAtomSystem(E_ca, G_ca, tau_ca, Q_ca, wavelength_
↪ca,
                                    laser_power = laser_power)
# Time evolve the system
calcium_system_gauss_and_dopp.timeEvolution(time_ca,
                            r_sigma = r_sigma,
                            n_beam_averaging = n_intensity,
                            doppler_width = doppler_width,
                            doppler_detunings = doppler_detunings,
                            doppler_averaging = True,
                            beam_profile_averaging = True)
# Plot the excited state population
rho_33_gauss_and_dopp = [abs(rho) for rho in calcium_system_gauss_and_dopp.Rho_t(s3, s3)]
```

```
Populating ground states equally as the initial condition.
```

Now, plot the results.

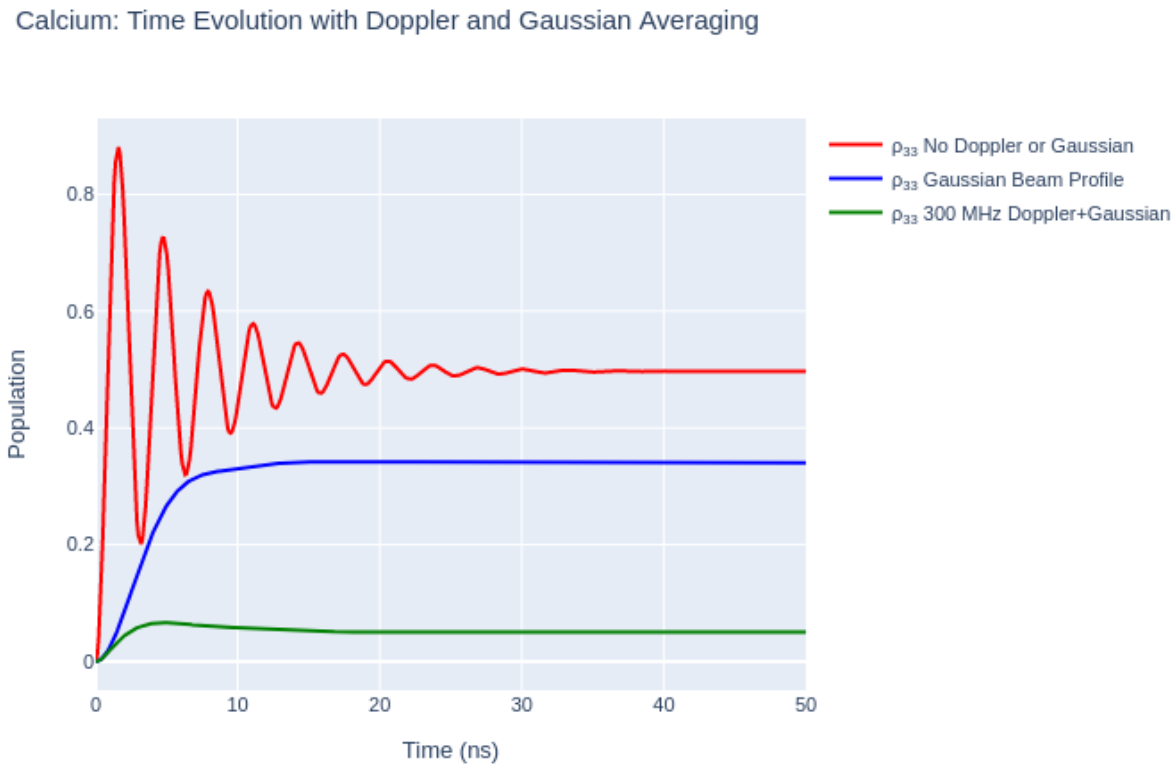
[21]:
```
fig_ca_gauss_and_dopp = go.Figure()
fig_ca_gauss_and_dopp.add_trace(go.Scatter(x = time_ca,
                                   y = rho_33,
                                   mode = 'lines',
                                  name = "<sub>33</sub> No Doppler or Gaussian",
                                  marker = dict(
                                      color = 'red',
                                      symbol = 'circle'
                            )))
fig_ca_gauss_and_dopp.add_trace(go.Scatter(x = time_ca,
                                   y = rho_33_gaussian,
                                   mode = "lines",
                                   name = "<sub>33</sub> Gaussian Beam Profile",
                                   marker = dict(
                                       color = "blue",
                                       symbol = "x"
                                   )))
fig_ca_gauss_and_dopp.add_trace(go.Scatter(x = time_ca,
                                   y = rho_33_gauss_and_dopp,
                                   mode = "lines",
                                   name = "<sub>33</sub> 300 MHz Doppler+Gaussian",
                                   marker = dict(
                                       color = "green",
                                       symbol = "x"
                                   )))
fig_ca_gauss_and_dopp.update_layout(title = f"Calcium: Time Evolution with Doppler and␣
↪Gaussian Averaging",
                  xaxis_title = "Time (ns)",
                  yaxis_title = "Population",
```

(continues on next page)

(continued from previous page)

```
                font = dict(
                    size = 11))
fig_ca_gauss_and_dopp.write_image("SavedPlots/tutorial1-ca-gauss-and-dopp.png")
Image("SavedPlots/tutorial1-ca-gauss-and-dopp.png")
```

[21]:



2.1.9 Exporting the Equations of Motion

LASED can print the equations of motion and/or export the equations to a .tex and a .pdf file. Note, that you must have a filename included if you want to export to a .tex file or .pdf file. Use the key idenifier `pretty_print_eq_filename` in `timeEvolution()`.

Note: You must have `pdflatex` installed on your system to generate a .pdf file. This is used to convert the .tex file produced to a .pdf file. You can do this on Windows or Mac by installing MiKTeX on your system.

[22]:

```
calcium_system_to_print = las.LaserAtomSystem(E_ca, G_ca, tau_ca, Q_ca, wavelength_ca,
                                  laser_intensity = intensity_ca)
calcium_system_to_print.timeEvolution(time_ca,
                                      pretty_print_eq = True,
                                      pretty_print_eq_tex = True,
                                      pretty_print_eq_pdf = True,
                                      pretty_print_eq_filename = "CalciumSystemEquations")
```

```
Populating ground states equally as the initial condition.
```

$$\dot{\rho}_{11} = i\rho_{13}\Omega(3,1,0) + \frac{1.0\rho_{22}}{\tau} - i\rho_{31}\Omega(3,1,0) + \frac{1.0\rho_{33}}{\tau} + \frac{1.0\rho_{44}}{\tau}$$

$$\dot{\rho}_{22} = -\frac{\rho_{22}}{\tau}$$

$$\dot{\rho}_{23} = i\rho_{21}\Omega(3,1,0) - \frac{\rho_{23}}{\tau}$$

$$\dot{\rho}_{24} = -\frac{\rho_{24}}{\tau}$$

$$\dot{\rho}_{32} = -i\rho_{12}\Omega(3,1,0) - \frac{\rho_{32}}{\tau}$$

$$\dot{\rho}_{33} = -i\rho_{13}\Omega(3,1,0) + i\rho_{31}\Omega(3,1,0) - \frac{\rho_{33}}{\tau}$$

$$\dot{\rho}_{34} = -i\rho_{14}\Omega(3,1,0) - \frac{\rho_{34}}{\tau}$$

$$\dot{\rho}_{42} = -\frac{\rho_{42}}{\tau}$$

$$\dot{\rho}_{43} = i\rho_{41}\Omega(3,1,0) - \frac{\rho_{43}}{\tau}$$

$$\dot{\rho}_{44} = -\frac{\rho_{44}}{\tau}$$

$$\dot{\rho}_{12} = -\frac{\rho_{12}}{2\tau} - i\rho_{32}\Omega(3,1,0)$$

$$\dot{\rho}_{13} = -i\rho_{11}\Omega(3,1,0) - \frac{\rho_{13}}{2\tau} - i\rho_{33}\Omega(3,1,0)$$

$$\dot{\rho}_{14} = -\frac{\rho_{14}}{2\tau} - i\rho_{34}\Omega(3,1,0)$$

$$\dot{\rho}_{21} = -\frac{\rho_{21}}{2\tau} + i\rho_{23}\Omega(3,1,0)$$

$$\dot{\rho}_{31} = i\rho_{11}\Omega(3,1,0) - \frac{\rho_{31}}{2\tau} + i\rho_{33}\Omega(3,1,0)$$

$$\dot{\rho}_{41} = -\frac{\rho_{41}}{2\tau} + i\rho_{43}\Omega(3,1,0)$$

2.2 Tutorial 2: D to P Exciation in Helium

This tutorial demonstrates how to set up an excitation from a D state to a P state for Helium. The Helium D-state we are considering is now not the ground state of the atom so decay can occur to lower states. The upper P-state can also decay to other states non-radiatively which can also be modelled. In this tutorial the decay to other states is modelled.

The use of simultaneous polarisations and the normalisation of the half-Rabi frequencies is demonstrated here as well.

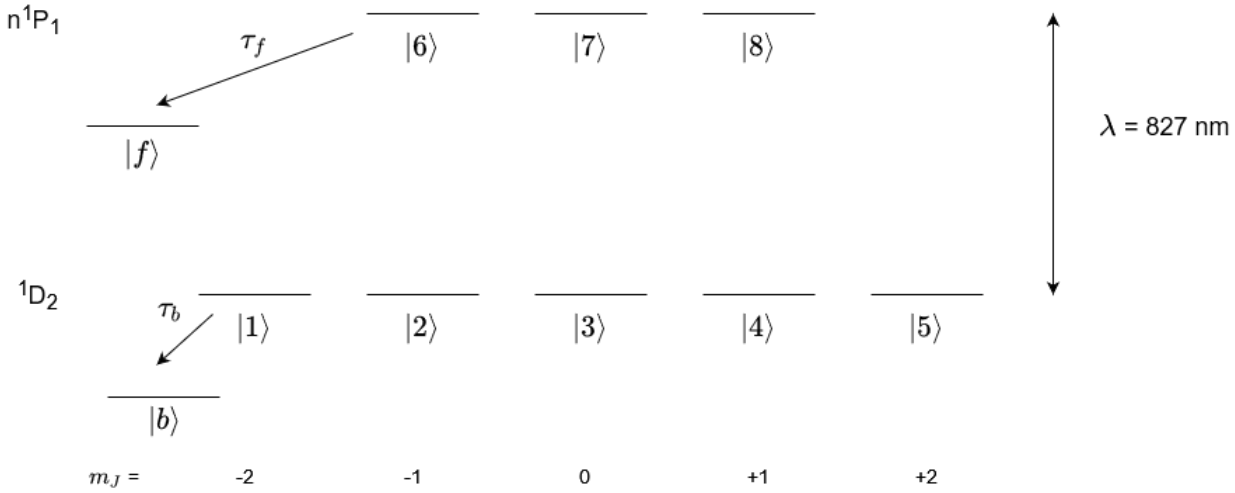
Also, the Wigner-D matrices are used to rotate the system to a different reference frame to show that this model is physically consistent in all reference frames.

Start by importing the libraries we will be using.

[1]:
```
import LASED as las
import numpy as np
from IPython.display import Image # This is to display html images in a notebook
import plotly.graph_objects as go
```

2.2.1 Decay to Other States

To set up a Laser-Atom system you must first declare the atomic states which you want to work with and label them. We are going to set up an example system for a D-state to a P-state transistion for helium where the P-state is a high principle quantum number Rydberg state. Therefore, we will assume that the wavelength of this fictitious transition is the ionisation energy of helium as this is a very high lying Rydberg state. This system is only for example purposes only and does not exist.



A level diagram of the system we will model is shown above.

The `LaserAtomSystem` is setup in the code block below. The decay (e.g. non-radiative decay) to other states outside the system from the excited state is characterised by the arrow from the upper states to state $|f\rangle$. This can be modelled into the `LaserAtomSystem` using the keyword `tau_f` and inputting the lifetime of this decay. The decay to other states from the lower state is shown in the diagram by the arrow from the lower state to the state $|b\rangle$. This can be modelled using the keyword `tau_b` and inputting the lifetime of this decay.

We will excite this system with simultaneous right-hand circular and left-hand circular polarised light in the natural frame with the laser beam travelling down the quantisation axis. This will be linearly-polarised light in the collision frame where the transverse E-field of the laser is oscillating along the quantisation axis. Therefore, we must set `Q = [-1, 1]` and set the `rabi_factors` to `[1, -1]` as noted in Tutorial 1 and scale the Rabi frequency in the system by using `rabi_scaling`. In this case set it to $1/\sqrt{2}$.

Then, we create the sub-states and put them into either the ground or excited states.

[2]:
```
# System parameters
laser_wavelength = 900e-9 # wavelength of transition
w_e = las.angularFreq(laser_wavelength)

# Create states
one = las.State(label = 1, w = 0, m = -2, L = 2, S = 0)
two = las.State(label = 2, w = 0, m = -1, L = 2, S = 0)
three = las.State(label = 3, w = 0, m = 0, L = 2, S = 0)
four = las.State(label = 4, w = 0, m = 1, L = 2, S = 0)
five = las.State(label = 5, w = 0, m = 2, L = 2, S = 0)

six = las.State(label = 6, w = w_e, m = -1, L = 1, S = 0)
seven = las.State(label = 7, w = w_e, m = 0, L = 1, S = 0)
eight = las.State(label = 8, w = w_e, m = 1, L = 1, S = 0)
```

(continues on next page)

(continued from previous page)

```
G = [one, two, three, four, five] # ground states
E = [six, seven, eight] # excited states
Q = [-1, 1] # laser radiation polarisation
rabi_scaling_he = 1/np.sqrt(2)
rabi_factors_he = [1, -1]
laser_intensity = 100 # mW/mm^2
tau = 60e3 # lifetime in ns (estimated)
tau_f = 1000 # non-radiative lifetime of rydberg upper state to other high-lying states␣
↪(ns)
tau_b = 5000  # non-radiatuve lifetime of metastable D-state (ns)
```

Set the simulation time for 1000 ns every 1 ns as follows:

[3]:
```
# Simulation parameters
start_time = 0
stop_time = 1000 # in ns
time_steps = 1001
time = np.linspace(start_time, stop_time, time_steps)
```

Create the `LaserAtomSystem` object. To set the initial conditions of the density matrix at t = 0 ns $\rho(t = 0)$ we can use the `setRho_0(s1, s2, val)` where `s1` and `s2` are `State` objects denoting the element of the density matrix to be set as $\rho_{s1,s2}$ and `val` denotes the value assigned to this element.

For this system we have set the populations of states $|1\rangle$, $|3\rangle$, and $|5\rangle$ as 1/3. So the density matrix elements ρ_{11} = ρ_{33} = ρ_{55} = 1/3.

[4]:
```
helium_system = las.LaserAtomSystem(E, G, tau, Q, laser_wavelength,
                                    laser_intensity = laser_intensity, rabi_scaling =␣
↪rabi_scaling_he,
                                    rabi_factors = rabi_factors_he)
helium_system_tauf = las.LaserAtomSystem(E, G, tau, Q, laser_wavelength,
                                         tau_f = tau_f,
                                    laser_intensity = laser_intensity, rabi_scaling =␣
↪rabi_scaling_he,
                                    rabi_factors = rabi_factors_he)
helium_system_tauf_taub = las.LaserAtomSystem(E, G, tau, Q, laser_wavelength,
                                         tau_f = tau_f, tau_b = tau_b,
                                    laser_intensity = laser_intensity, rabi_scaling =␣
↪rabi_scaling_he,
                                    rabi_factors = rabi_factors_he)
helium_system.setRho_0(one, one, 1/3)
helium_system.setRho_0(three, three, 1/3)
helium_system.setRho_0(five, five, 1/3)
helium_system_tauf.setRho_0(one, one, 1/3)
helium_system_tauf.setRho_0(three, three, 1/3)
helium_system_tauf.setRho_0(five, five, 1/3)
helium_system_tauf_taub.setRho_0(one, one, 1/3)
helium_system_tauf_taub.setRho_0(three, three, 1/3)
helium_system_tauf_taub.setRho_0(five, five, 1/3)
```

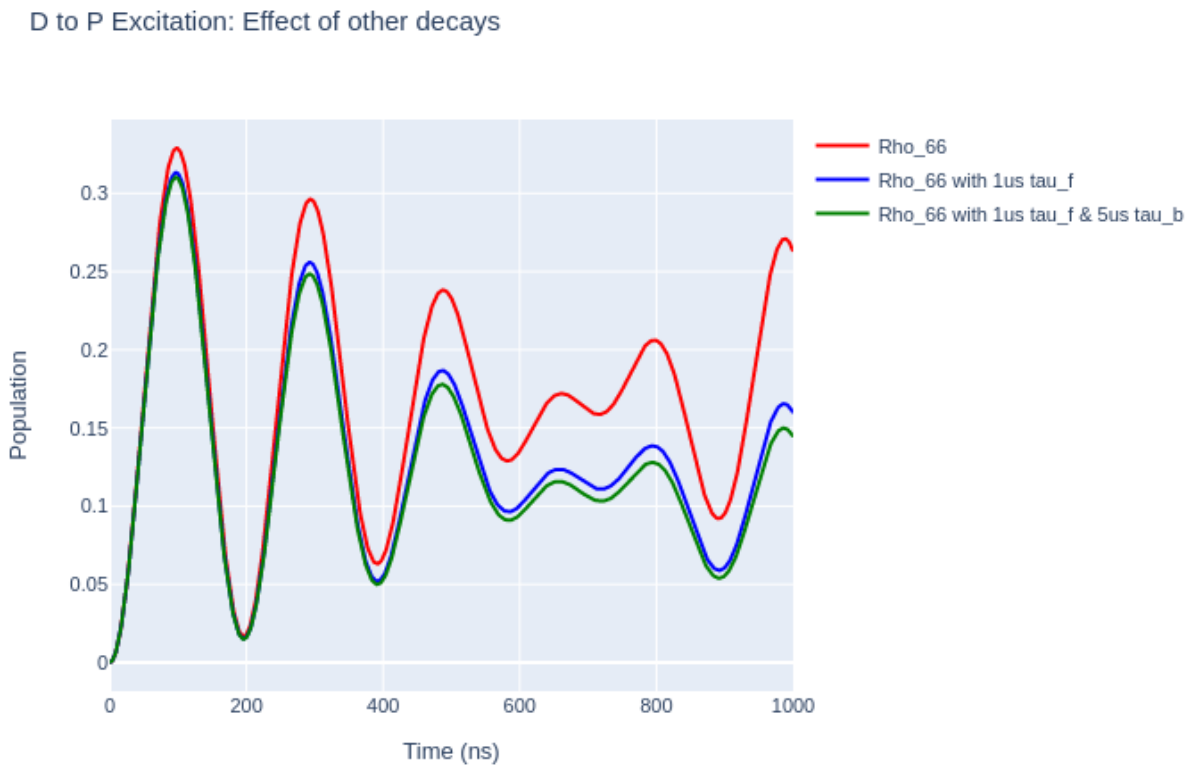
Time evolve the system.

[5]:
```
helium_system.timeEvolution(time)
helium_system_tauf.timeEvolution(time)
helium_system_tauf_taub.timeEvolution(time)
```

Now, we can plot the populations using `Plotly` (or any other plotting package).

[6]:
```
rho_66 = [ abs(rho) for rho in helium_system.Rho_t(six, six)]
rho_66_tauf = [abs(rho) for rho in helium_system_tauf.Rho_t(six, six)]
rho_66_tauf_taub = [abs(rho) for rho in helium_system_tauf_taub.Rho_t(six, six)]

fig_upper = go.Figure(data = go.Scatter(x = time,
                                        y = rho_66,
                                        mode = 'lines',
                                           name = "Rho_66",
                                           marker = dict(
                                       color = 'red',
                                       symbol = 'x',
                                    )))

fig_upper.add_trace(go.Scatter(x = time,
                                        y = rho_66_tauf,
                                        mode = 'lines',
                                           name = "Rho_66 with 1us tau_f",
                                           marker = dict(
                                       color = 'blue',
                                       symbol = 'square',
                                   )))
fig_upper.add_trace(go.Scatter(x = time,
                                        y = rho_66_tauf_taub,
                                        mode = 'lines',
                                           name = "Rho_66 with 1us tau_f & 5us tau_b",
                                           marker = dict(
                                       color = 'green',
                                       symbol = 'circle',
                                   )))

fig_upper.update_layout(showlegend = True,
                          title = "D to P Excitation: Effect of other decays",
                        xaxis_title = "Time (ns)",
                        yaxis_title = "Population",
                       font = dict(
                           size = 11))
fig_upper.write_image("SavedPlots/tutorial2-HeLifetimes.png")
fig_upper.write_image("SavedPlots/tutorial2-HeLifetimes.svg")
Image("SavedPlots/tutorial2-HeLifetimes.png")
```

[6]:



2.2.2 Rotation

With `LASED` we can rotate density matrices to different reference frames using the Wigner-D matrix.

Note: The Wigner-D matrix is only defined for J-represenattion (when isospin I = 0) and therefore when in F-representation the rotation may not be correct. Also, density matrices can only be rotated for single atomic states so the optical coherences between ground and excited states cannot be rotated. To obtain optical coherences in a different reference frame the initial density matrix (at t = 0) must be rotated and time evolved with the polarisation rotated to that frame.

The rotation is defined by Euler angles `alpha`, `beta`, and `gamma`. The frame is rotated with each angle in succession so that: - `alpha` is the rotation (in radians) around the initial z-axis to obtain the new frame Z' - `beta` is the rotation (in radians) about the new y'-axis to obtain the new frame Z" - `gamma` is the rotation (in radians) about the new z"-axis to obtain the final frame

In `LASED` to rotate the initial density matrix use `rotateRho_0(alpha, beta, gamma)` on the `LaserAtomSystem`. In this he lium system the atom is changed to be defined in the collision frame from the natural frame so the polarisation is changed to be purely linear with `Q = [0]` and scaled to 1.

Note: To simulate a linear polarisation with an angle with respect to the x-axis we can initialise the density matrix in that frame, rotate to the collision frame (with the polarisation aligned with the x-axis, time evolve the system, and then rotate back to the frame where the polarisation is at an angle.

[7]:
```
alpha = np.pi/2  # Angles are in radians
beta = np.pi/2
gamma = -np.pi/2
helium_system_rot = helium_system
helium_system_rot.rotateRho_0(alpha, beta, gamma)
helium_system_rot.Q = [0]
helium_system_rot.rabi_scaling = 1
helium_system_rot.rabi_factors = [1]
```

Now, we can time evolve this system in this new reference frame.

[8]:
```
print(helium_system_rot)
helium_system_rot.timeEvolution(time)
```

```
LaserAtomSystem([6, 7, 8], [1, 2, 3, 4, 5], 60000.0, [0], [1, 0, -1], 9e-07, None, 100,␣
↪None)
```

Now we can plot what the populations look like.

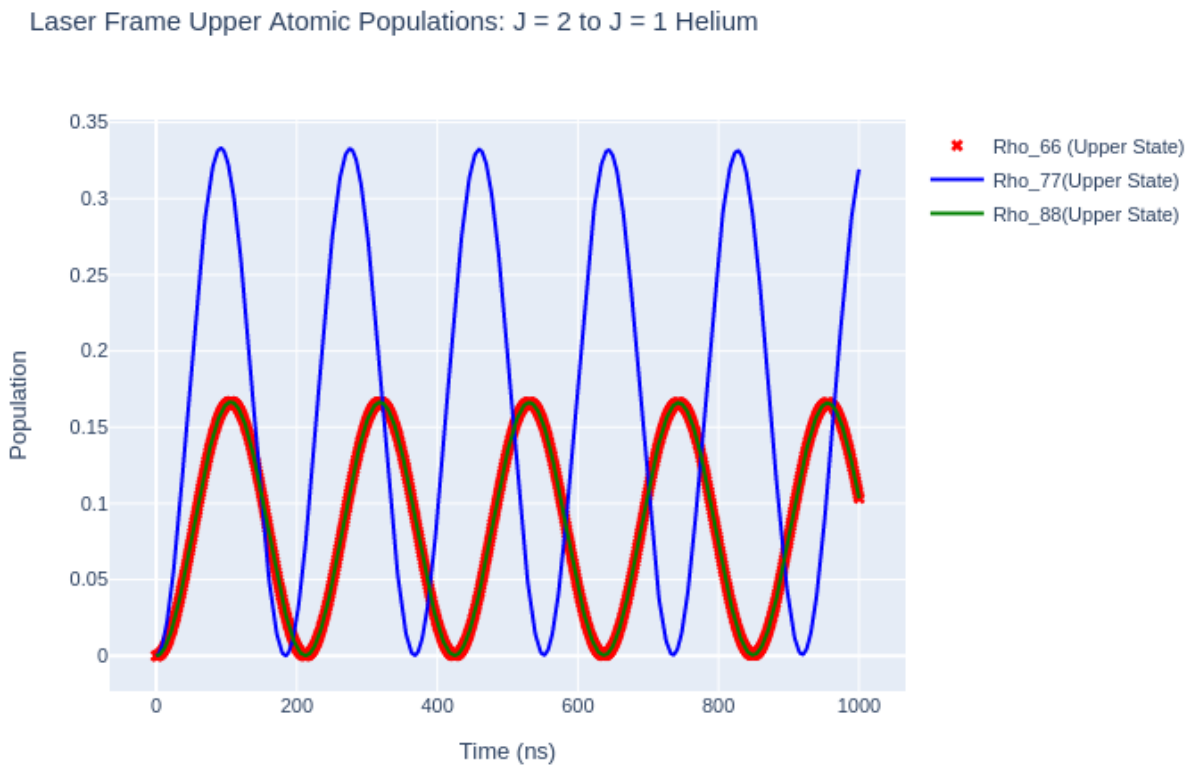
[9]:
```
las_sys = helium_system_rot
rho_66 = [ abs(rho) for rho in las_sys.Rho_t(six, six)]
rho_77 = [abs(rho) for rho in las_sys.Rho_t(seven, seven)]
rho_88 = [abs(rho) for rho in las_sys.Rho_t(eight, eight)]

fig_upper = go.Figure(data = go.Scatter(x = time,
                                        y = rho_66,
                                        mode = 'markers',
                                           name = "Rho_66 (Upper State)",
                                           marker = dict(
                                       color = 'red',
                                       symbol = 'x',
                                 )))

fig_upper.add_trace(go.Scatter(x = time,
                                        y = rho_77,
                                        mode = 'lines',
                                           name = "Rho_77(Upper State)",
                                           marker = dict(
                                       color = 'blue',
                                       symbol = 'square',
                               )))
fig_upper.add_trace(go.Scatter(x = time,
                                        y = rho_88,
                                        mode = 'lines',
                                           name = "Rho_88(Upper State)",
                                           marker = dict(
                                       color = 'green',
                                       symbol = 'circle',
                               )))

fig_upper.update_layout(title = "Laser Frame Upper Atomic Populations: J = 2 to J = 1␣
↪Helium",
                  xaxis_title = "Time (ns)",
                  yaxis_title = "Population",
```

(continues on next page)

(continued from previous page)

```
                font = dict(
                    size = 11))

fig_upper.write_image("SavedPlots/tutorial2-HeFigUpperCollFrame.png")
Image("SavedPlots/tutorial2-HeFigUpperCollFrame.png")
```

[9]:



[10]:

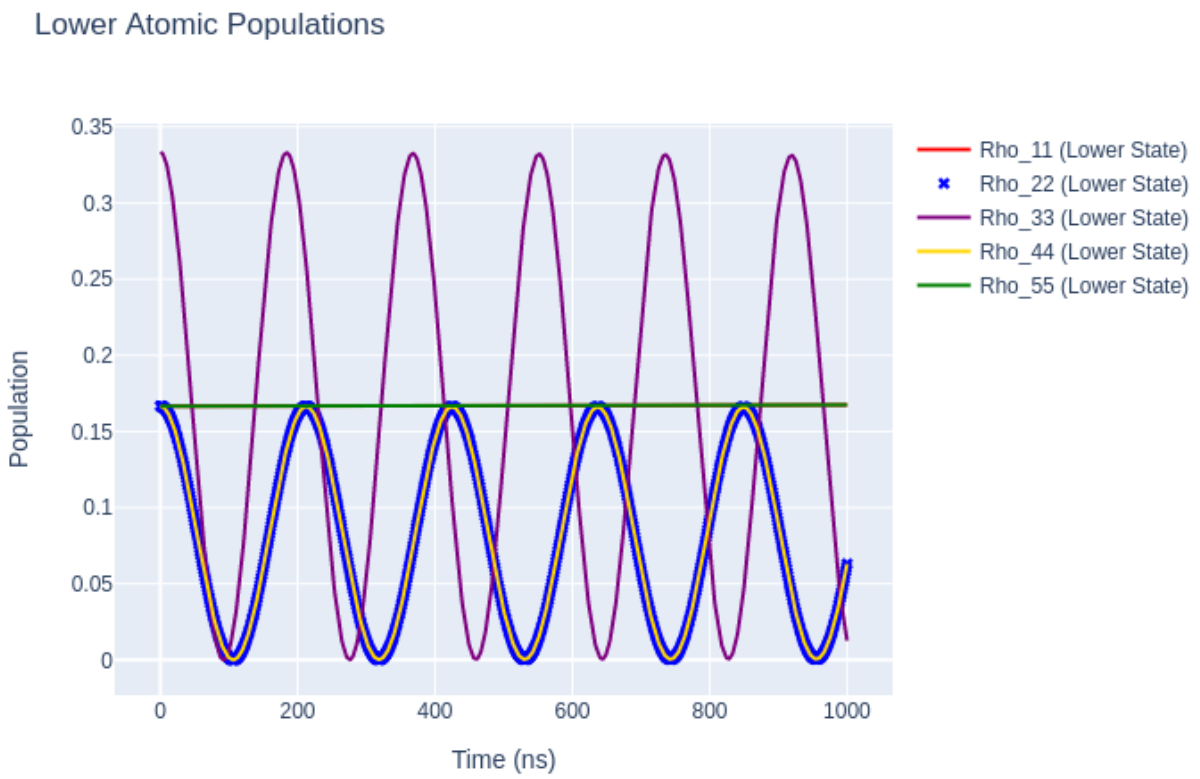
```
rho11 = [ abs(rho) for rho in las_sys.Rho_t(one, one)]
rho22 = [ abs(rho) for rho in las_sys.Rho_t(two, two)]
rho33 = [ abs(rho) for rho in las_sys.Rho_t(three, three)]
rho44 = [ abs(rho) for rho in las_sys.Rho_t(four, four)]
rho55 = [ abs(rho) for rho in las_sys.Rho_t(five, five)]


fig_lower = go.Figure(data = go.Scatter(x = time,
                                        y = rho11,
                                        mode = 'lines',
                                           name = "Rho_11 (Lower State)",
                                           marker = dict(
                                       color = 'red',
                                       symbol = 'circle',
                                    )))
fig_lower.add_trace(go.Scatter(x = time,
                                        y = rho22,
```

(continues on next page)

(continued from previous page)

```
                                    mode = 'markers',
                                       name = "Rho_22 (Lower State)",
                                       marker = dict(
                                    color = 'blue',
                                    symbol = 'x',
                                )))
fig_lower.add_trace(go.Scatter(x = time,
                                    y = rho33,
                                    mode = 'lines',
                                       name = "Rho_33 (Lower State)",
                                       marker = dict(
                                    color = 'purple',
                                    symbol = 'x',
                                )))
fig_lower.add_trace(go.Scatter(x = time,
                                    y = rho44,
                                    mode = 'lines',
                                       name = "Rho_44 (Lower State)",
                                       marker = dict(
                                    color = 'gold',
                                    symbol = 'x',
                                )))
fig_lower.add_trace(go.Scatter(x = time,
                                    y = rho55,
                                    mode = 'lines',
                                       name = "Rho_55 (Lower State)",
                                       marker = dict(
                                    color = 'green',
                                    symbol = 'square',
                                )))
fig_lower.update_layout(title = "Lower Atomic Populations",
                        xaxis_title = "Time (ns)",
                  yaxis_title = "Population")

fig_lower.write_image("SavedPlots/tutorial2-HeFigLowerCollFrame.png")
Image("SavedPlots/tutorial2-HeFigLowerCollFrame.png")
```

[10]:



If we rotate the time-evolved density matrix `rho_t` back to the natural frame we should get the same populations as when we evolved in the natural frame as the excitation must be the same in all reference frames. We can do this rotation by using `rotateRho_t` on the `LaserAtomSystem`.

Note: A message will be thrown saying that the optical coherences are not rotated.

[11]:
```
helium_system_rot.rotateRho_t(alpha, -beta, gamma)
```

Now we can plot the results and obtain the same plots as in the first time evolution.

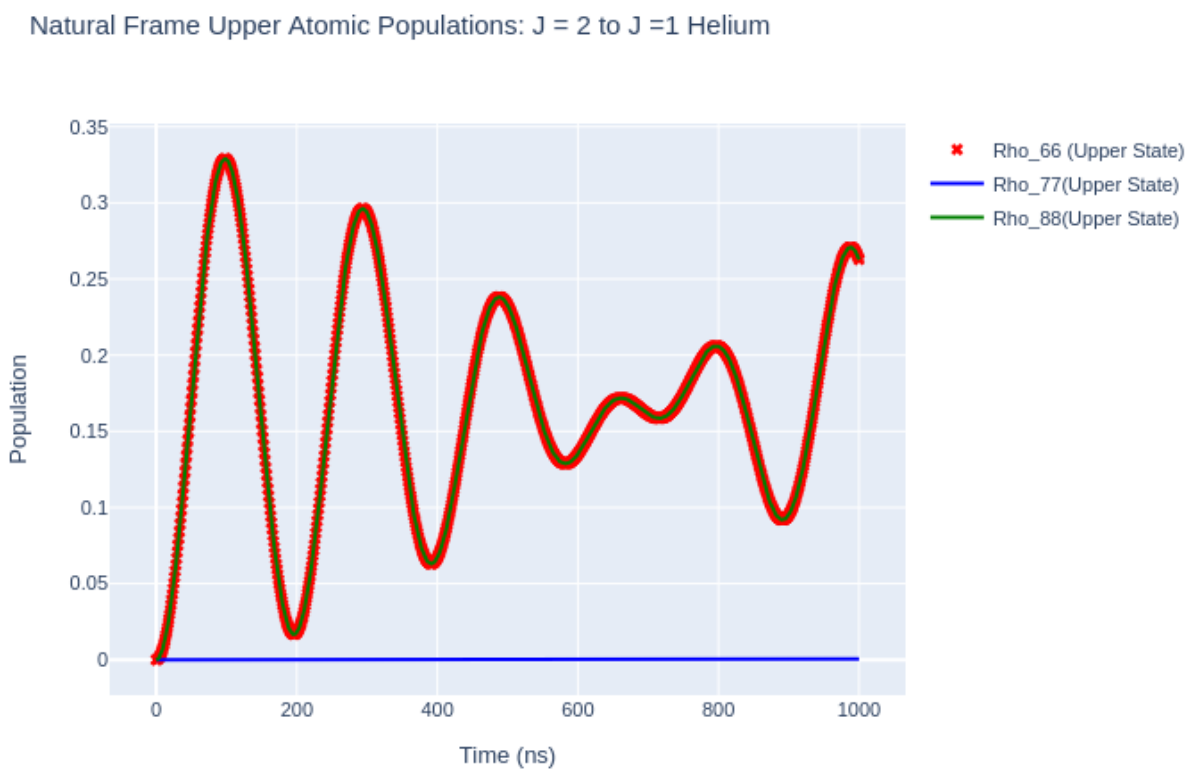
[12]:
```
las_sys = helium_system_rot
rho_66 = [ abs(rho) for rho in las_sys.Rho_t(six, six)]
rho_77 = [abs(rho) for rho in las_sys.Rho_t(seven, seven)]
rho_88 = [abs(rho) for rho in las_sys.Rho_t(eight, eight)]

fig_upper = go.Figure(data = go.Scatter(x = time,
                                        y = rho_66,
                                        mode = 'markers',
                                           name = "Rho_66 (Upper State)",
                                           marker = dict(
                                       color = 'red',
                                       symbol = 'x',
                                   )))

fig_upper.add_trace(go.Scatter(x = time,
```

(continues on next page)

(continued from previous page)

```
                                    y = rho_77,
                                    mode = 'lines',
                                       name = "Rho_77(Upper State)",
                                       marker = dict(
                                    color = 'blue',
                                    symbol = 'square',
                                )))
fig_upper.add_trace(go.Scatter(x = time,
                                    y = rho_88,
                                    mode = 'lines',
                                       name = "Rho_88(Upper State)",
                                       marker = dict(
                                    color = 'green',
                                    symbol = 'circle',
                                )))

fig_upper.update_layout(title = "Natural Frame Upper Atomic Populations: J = 2 to J =1␣
↪Helium",
                  xaxis_title = "Time (ns)",
                  yaxis_title = "Population",
                 font = dict(
                     size = 11))

fig_upper.write_image("SavedPlots/tutorial2-HeFigUpperRotatedToNatFrame.png")
Image("SavedPlots/tutorial2-HeFigUpperRotatedToNatFrame.png")
```

[12]:



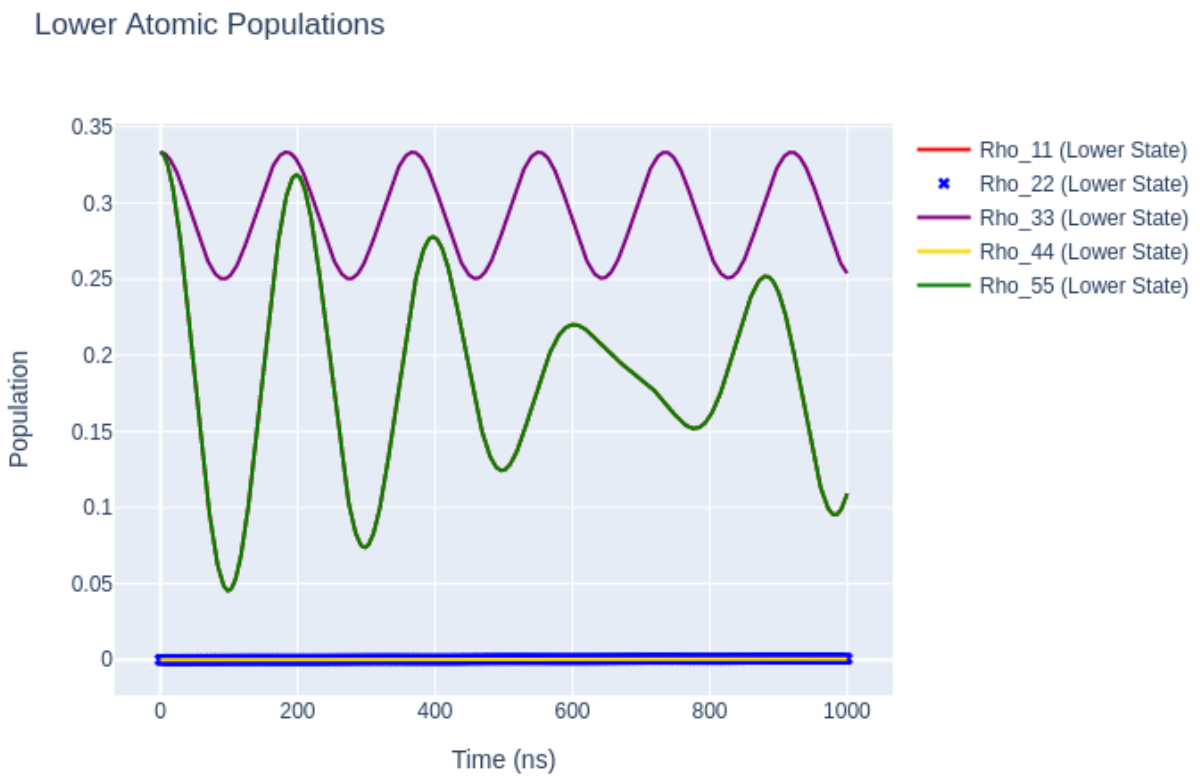
[13]:
```
rho11 = [ abs(rho) for rho in las_sys.Rho_t(one, one)]
rho22 = [ abs(rho) for rho in las_sys.Rho_t(two, two)]
rho33 = [ abs(rho) for rho in las_sys.Rho_t(three, three)]
rho44 = [ abs(rho) for rho in las_sys.Rho_t(four, four)]
rho55 = [ abs(rho) for rho in las_sys.Rho_t(five, five)]


fig_lower = go.Figure(data = go.Scatter(x = time,
                                        y = rho11,
                                        mode = 'lines',
                                           name = "Rho_11 (Lower State)",
                                           marker = dict(
                                       color = 'red',
                                       symbol = 'circle',
                                    )))
fig_lower.add_trace(go.Scatter(x = time,
                                        y = rho22,
                                        mode = 'markers',
                                           name = "Rho_22 (Lower State)",
                                           marker = dict(
                                       color = 'blue',
                                       symbol = 'x',
                                    )))
```

(continues on next page)

(continued from previous page)

```
fig_lower.add_trace(go.Scatter(x = time,
                               y = rho33,
                               mode = 'lines',
                                  name = "Rho_33 (Lower State)",
                                  marker = dict(
                              color = 'purple',
                              symbol = 'x',
                           )))
fig_lower.add_trace(go.Scatter(x = time,
                               y = rho44,
                               mode = 'lines',
                                  name = "Rho_44 (Lower State)",
                                  marker = dict(
                              color = 'gold',
                              symbol = 'x',
                           )))
fig_lower.add_trace(go.Scatter(x = time,
                               y = rho55,
                               mode = 'lines',
                                  name = "Rho_55 (Lower State)",
                                  marker = dict(
                              color = 'green',
                              symbol = 'square',
                           )))
fig_lower.update_layout(title = "Lower Atomic Populations",
                        xaxis_title = "Time (ns)",
                  yaxis_title = "Population")

fig_lower.write_image("SavedPlots/tutorial2-HeFigLowerRotatedToNatFrame.png")
Image("SavedPlots/tutorial2-HeFigLowerRotatedToNatFrame.png")
```

[13]:



2.2.3 Polarisation Angle

The polarisation angle in `LASED` is assumed to be 0 during excitation i.e. the polarisation vector is in the same direction as the quantisation axis (QA). To model a non-zero polarisation angle with respect to the QA the `LaserAtomSystem()` must be rotated by the polarisation angle θ to the frame where the polarisation is in the direction of the QA, time evolved using `timeEvolution()` and then rotated back to the old frame (so rotate by -θ).

As an example, the Helium system will be excited by liner light polarised at an angle of 30° to the QA. To do this rotate the He system by (α, β, γ) = (90:math:^{*circ*}, 30°, -90:math:^{*circ*}), excite with polarisation `Q = [0]`, and then rotate the system back to the old reference frame by performing the rotation (90:math:^{*circ*}, -30:math:^{*circ*}, -90:math:^{*circ*}).

Rotate the Helium system and time evolve using a linear polarisation:

[14]:
```
theta = 30 # Polarisation angle
alpha = np.pi/2
beta = 30*np.pi/180  # Convert to radians
gamma = -np.pi/2

# Set up the laser-atom system
helium_system_polarisation_angle = helium_system_rot
helium_system_polarisation_angle.rotateRho_0(alpha, beta, gamma)
```

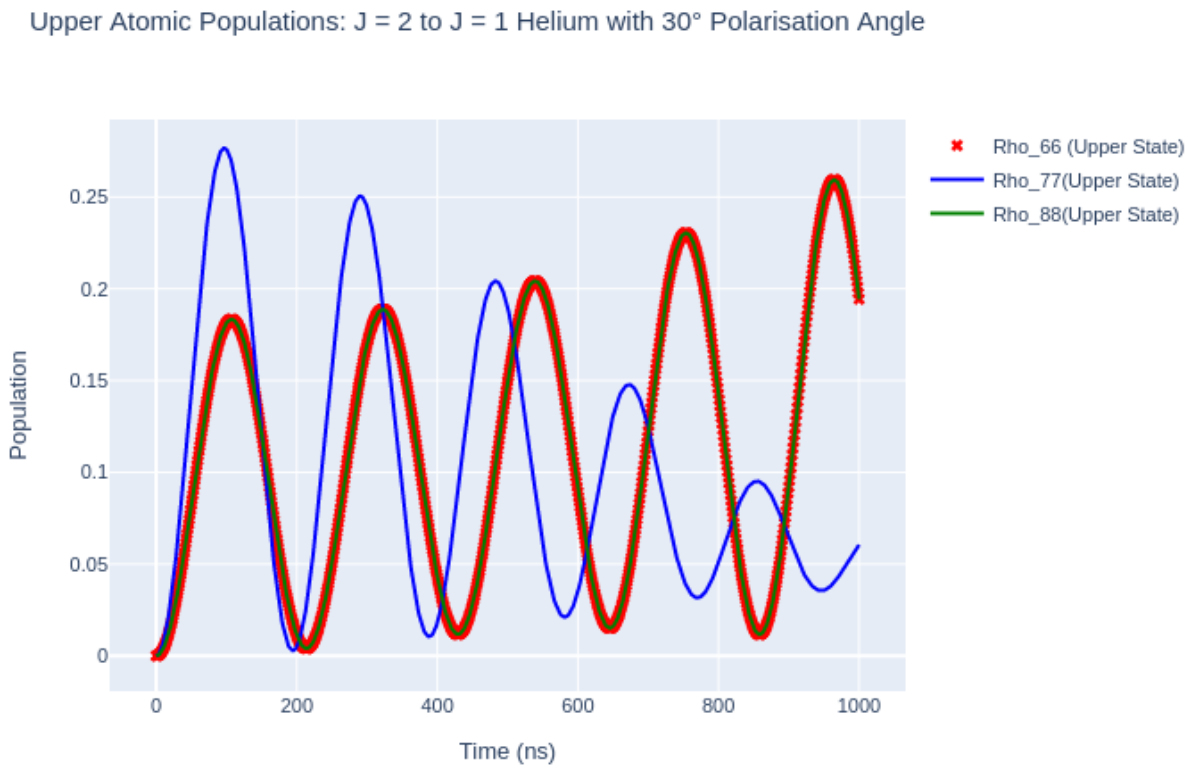
(continues on next page)

(continued from previous page)

```
# Time evolve and plot
helium_system_polarisation_angle.timeEvolution(time)

# Rotate rho_t back to the original reference frame (the optical coherences will not be␣
↪preserved)
helium_system_polarisation_angle.rotateRho_t(alpha, -beta, gamma)
```

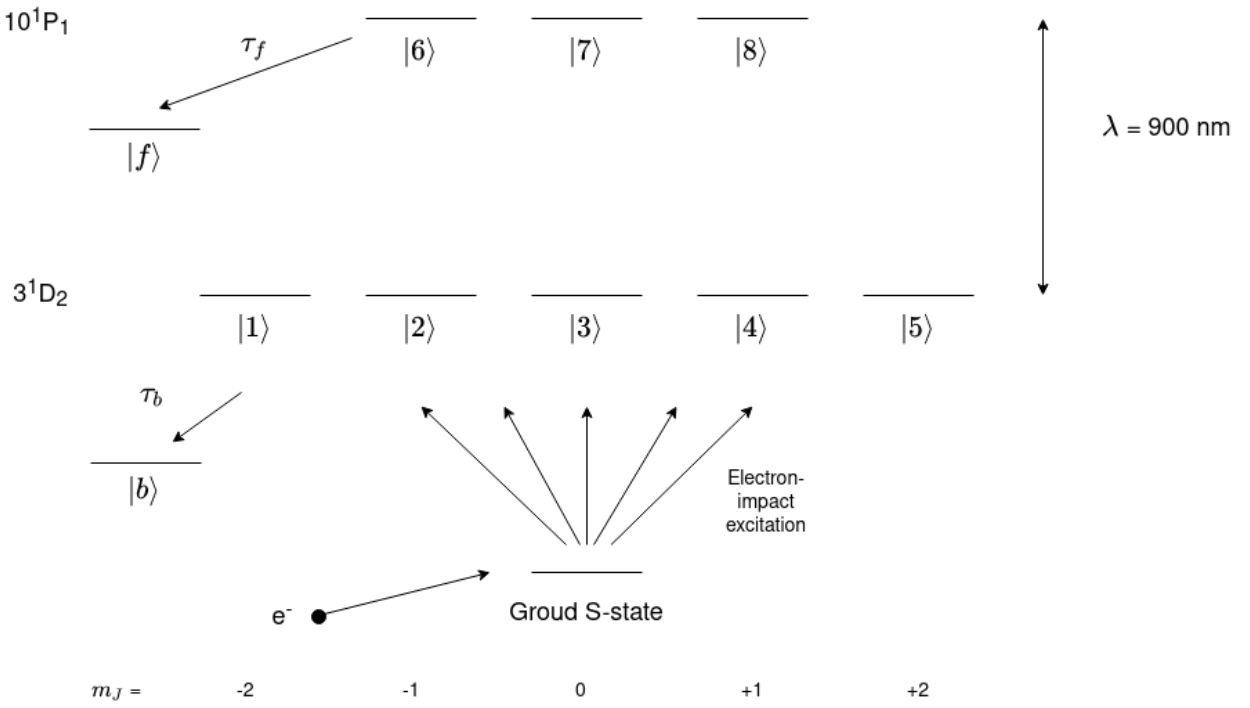
[15]:
```
las_sys = helium_system_polarisation_angle
rho_66 = [ abs(rho) for rho in las_sys.Rho_t(six, six)]
rho_77 = [abs(rho) for rho in las_sys.Rho_t(seven, seven)]
rho_88 = [abs(rho) for rho in las_sys.Rho_t(eight, eight)]

fig_upper = go.Figure(data = go.Scatter(x = time,
                                    y = rho_66,
                                    mode = 'markers',
                                       name = "Rho_66 (Upper State)",
                                       marker = dict(
                                   color = 'red',
                                   symbol = 'x',
                               )))

fig_upper.add_trace(go.Scatter(x = time,
                                    y = rho_77,
                                    mode = 'lines',
                                       name = "Rho_77(Upper State)",
                                       marker = dict(
                                   color = 'blue',
                                   symbol = 'square',
                              )))
fig_upper.add_trace(go.Scatter(x = time,
                                    y = rho_88,
                                    mode = 'lines',
                                       name = "Rho_88(Upper State)",
                                       marker = dict(
                                   color = 'green',
                                   symbol = 'circle',
                              )))

fig_upper.update_layout(title = f"Upper Atomic Populations: J = 2 to J = 1 Helium with
↪{theta}° Polarisation Angle",
                  xaxis_title = "Time (ns)",
                  yaxis_title = "Population",
                 font = dict(
                     size = 11))
fig_upper.write_image("SavedPlots/tutorial2-HePolarisationAngle.png")
Image("SavedPlots/tutorial2-HePolarisationAngle.png")
```

[15]:



2.3 Tutorial 3: He Excitation: Initialisation with Data and Angular Shape

In this tutorial a more relaistic system will be simulated which could be the process in an experiment and the angular shape of the electron cloud will be plotted and a video of the angular shape of the electron cloud over time will be plotted. Let's say that helium is excited to the 3D-state via electron impact from its ground state whilst a laser excites these electron-excited atoms into a high-lying 10P-state. A level diagram of the system is shown below.



Let's import all the libraries needed. We will need a library to read in data from a datafile. `pandas` is a good library to do this.

[1]:
```
import LASED as las
import numpy as np
import plotly.graph_objects as go
from IPython.display import Image # This is to display html images in a notebook
import pandas as pd  #This librray is used to read in the data
```

2.3.1 Setting up the System

This is a very similar system to last tutorial except for the lifetime and laser wavelength. Both of these values are taken from looking at the transition on the National Institute of Standards and Technology's (NIST) atomic database. The page for He can be found here. The 10P state data is found here and the 3D data is found here.

To calculate the total lifetime for τ_f and τ_b sum the decay rates A_{ki} for each transition from the upper state and lower state to states not in the laser-excitation and then take the reciprocal of this to get the total lifetime. For example, τ_f for the 10P state of He has many decays to lower states however the largest decay rates will contribute the most so the first three are

$$\tau_f = \frac{1}{\Gamma_{f,total}} \approx \frac{1}{\Gamma_{f,1} + \Gamma_{f,2} + \Gamma_{f,3}}$$

[2]:
```
tau = 80.7e3 # lifetime in ns (take from NIST 1/A_ki as this is in rad/s)
tau_f = 59.6 # lifetime of upper state to other states in ns
tau_b = 15.7 # lifetime of lower state decay to other states in ns
laser_wavelength = 900e-9 # wavelength of laser 3D to 9P state from NIST

w_e = las.angularFreq(laser_wavelength)  # Converted to angular frequency in Grad/s
```

(continues on next page)

(continued from previous page)

```
# Create states for J = 2 -> J = 1
one = las.State(L = 2, S = 0, m = -2, w = 0, label = 1)
two = las.State(L = 2, S = 0, m = -1, w = 0, label = 2)
three = las.State(L = 2, S = 0, m = 0, w = 0, label = 3)
four = las.State(L = 2, S = 0, m = 1, w = 0, label = 4)
five = las.State(L = 2, S = 0, m = 2, w = 0, label = 5)
six = las.State(L = 1, S = 0, m = -1, w = w_e, label = 6)
seven = las.State(L = 1, S = 0, m = 0, w = w_e, label = 7)
eight = las.State(L = 1, S = 0, m = 1, w = w_e, label = 8)

G = [one, two, three, four, five] # ground states
E = [six, seven, eight] # excited states
Q = [0] # laser radiation polarisation

# Simulation parameters
start_time = 0
stop_time = 200 # in ns
steps = 201
time = np.linspace(start_time, stop_time, steps)

# Laser parameters
laser_intensity = 1500 # laser intensity in mW/mm^2
```

Next, make the `LaserAtomSystem`.

[3]:
```
helium_system = las.LaserAtomSystem(E, G, tau, Q, laser_wavelength, tau_f = tau_f,
                                    tau_b = tau_b, laser_intensity = laser_intensity)
```

2.3.2 Initialisating the Laser-Atom System with Data

Now, we need to extract the data from the datafile. This is scattering amplitude data kindly given by Igor Brey who has calculated these amplitudes from a model. The model is quite accurate for helium and gives amplitudes for various impact angles.

Igor calculated electron impact excitation amplitudes for all angles to the 3^1D_2 state. With this data we can generate an initial density matrix to excite from real data. The data was generated in the collison frame.

The density matrix elements correspond to $\rho_{M,M'} = \sum_{m'} \langle M m'_f | \rho_f | M' m'_f \rangle = \langle f(M) f^*(M') \rangle$

where $f(M)$ is the scattering amplitude for state $|JM\rangle$.

Select data from the scattering angle of 45 degrees.

[4]:
```
scat_data = pd.read_csv("31DAmplitudes.csv")
scat_angle_data = scat_data.loc[scat_data['Angle'] == 45]
print(scat_angle_data)
```

```
    Angle  Re a(+2)*1e-12  Im(a+2)*1e-12  Re a(+1)*1e-12  Im(a+1)*1e-12  \
45     45           8.821           19.5           14.84         -35.66

    Re a(0)*1e-12  Im(a0)*1e-12  Re(a-1)*1e-12  Im(a-1)*1e-12  Re(a-2)*1e-12  \
45          85.82        -12.27         -14.84          35.66          8.821
```

(continues on next page)

(continued from previous page)

```
    Im(a-2)*1e-12
45           19.5
```

Now we have read in what values the amplitudes are and we can now obtain the diagonal density matrix elements. First, put the scattering amplitudes in variables and put them into a list.

[5]:
```
f_2 = scat_angle_data.iloc[0].at["Re a(+2)*1e-12"] + scat_angle_data.iloc[0].at[
↪"Im(a+2)*1e-12"]*1.j
f_1 = scat_angle_data.iloc[0].at["Re a(+1)*1e-12"] + scat_angle_data.iloc[0].at[
↪"Im(a+1)*1e-12"]*1.j
f_0 = scat_angle_data.iloc[0].at["Re a(0)*1e-12"] + scat_angle_data.iloc[0].at[
↪"Im(a0)*1e-12"]*1.j
f_minus1 = scat_angle_data.iloc[0].at["Re(a-1)*1e-12"] + scat_angle_data.iloc[0].at[
↪"Im(a-1)*1e-12"]*1.j
f_minus2 = scat_angle_data.iloc[0].at["Re(a-2)*1e-12"] + scat_angle_data.iloc[0].at[
↪"Im(a-2)*1e-12"]*1.j
f = [f_minus2, f_minus1, f_0, f_1, f_2]
```

Now, calculate the density matrix ρ. Normalise the populations so that tr() = 1.

Note: The density matrix created must be in the order with the value of $\rho_{-m,-m}$ starting the upper left-hand side of the matrix to $\rho_{m,m}$ at the bottom right-hand corner. m is the projection of the total angular momentum.

[6]:
```
rho_collision = np.zeros((len(G), len(G)), dtype = complex)

for i, population in enumerate(f):
    for j, population in enumerate(f):
        rho_collision[i][j] = f[i]*np.conj(f[j])

# Normalisation
norm = 0
for i in range(len(G)):
    norm += rho_collision[i][i]
rho_collision = rho_collision/norm
print("The J = 2 density matrix in the collision frame is \n", pd.DataFrame(np.flip(rho_
↪collision)))
```

```
The J = 2 density matrix in the collision frame is
                      0                     1                     2  \
0  0.040126+0.000000j -0.049448+0.052905j  0.045355+0.156080j
1 -0.049448-0.052905j  0.130688+0.000000j  0.149895-0.252136j
2  0.045355-0.156080j  0.149895+0.252136j  0.658372+0.000000j
3  0.049448+0.052905j -0.130688+0.000000j -0.149895+0.252136j
4  0.040126+0.000000j -0.049448+0.052905j  0.045355+0.156080j

                      3                     4
0  0.049448-0.052905j  0.040126+0.000000j
1 -0.130688+0.000000j -0.049448-0.052905j
2 -0.149895-0.252136j  0.045355-0.156080j
3  0.130688+0.000000j  0.049448+0.052905j
4  0.049448-0.052905j  0.040126+0.000000j
```

Note: Above `pandas` is used to convert the matrix into a `DataFrame` just for display purposes and the display of a

matrix as a DataFrame is neater and the indeces are easier to read.

Now, we must initialise the density matrix of the `LaserAtomSystem` with this calculated ground state density matrix. To do this, we can use the function `appendDensityMatrixToRho_0(density_rho, state_type)` where: * `rho_0` is a column vector which is of size n^2, where n is the number of energy levels in the system. This column vector represents the denisty matrix for the laser-atom system. It is flattened (one-dimensional) as it is easier to solve the equations of motion generated with this form. It is in the form of:

$$\rho_{flattened} = \begin{bmatrix} \rho_{11} \\ \rho_{12} \\ . \\ . \\ \rho_{1n} \\ \rho_{21} \\ . \\ . \\ \rho_{nn} \end{bmatrix}$$

- `density_rho` is the density matrix of the ground or upper state density matrix to be input into the flattened coupled denisty matrix and then initialised as `rho_0` in the `LaserAtomSystem`. This is of type `ndarray`. This is a usual 2D matrix.
- `state_type` is a character of 'e' or 'g' for appending the excited state density matrix or 'g' for the ground state density matrix.

`rho_0` is already initialised as empty with the correct size when a `LaserAtomSystem` is initialised.

[7]:
```
helium_system.appendDensityMatrixToRho_0(rho_collision, "g")
print("The initial  for the laser-atom system is: \n", helium_system.rho_0)
```

```
The initial  for the laser-atom system is:
 [[ 0.04012626+0.j        ]
 [ 0.0494475 -0.05290513j]
 [ 0.04535541+0.15607978j]
 [-0.0494475 +0.05290513j]
 [ 0.04012626+0.j        ]
 [ 0.        +0.j        ]
 [ 0.        +0.j        ]
 [ 0.        +0.j        ]
 [ 0.0494475 +0.05290513j]
 [ 0.1306877 +0.j        ]
 [-0.1498946 +0.25213635j]
 [-0.1306877 +0.j        ]
 [ 0.0494475 +0.05290513j]
 [ 0.        +0.j        ]
 [ 0.        +0.j        ]
 [ 0.        +0.j        ]
 [ 0.04535541-0.15607978j]
 [-0.1498946 -0.25213635j]
 [ 0.65837208+0.j        ]
 [ 0.1498946 +0.25213635j]
 [ 0.04535541-0.15607978j]
 [ 0.        +0.j        ]
 [ 0.        +0.j        ]
 [ 0.        +0.j        ]
 [-0.0494475 -0.05290513j]
```

(continues on next page)

(continued from previous page)

```
 [-0.1306877 +0.j         ]
 [ 0.1498946 -0.25213635j]
 [ 0.1306877 +0.j         ]
 [-0.0494475 -0.05290513j]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.04012626+0.j         ]
 [ 0.0494475 -0.05290513j]
 [ 0.04535541+0.15607978j]
 [-0.0494475 +0.05290513j]
 [ 0.04012626+0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]
 [ 0.         +0.j         ]]
```

2.3.3 Time Evolution

Model the polarisation at 30° to the QA so rotate the system, then time evolve, and then rotate back.

[8]:
```
# Rotate by (90°, 30°, -90°)
theta = 30
alpha = np.pi/2
beta = theta*np.pi/180
gamma = -np.pi/2
helium_system.rotateRho_0(alpha, beta, gamma)
```

Time evolve:

[9]:
```
helium_system.timeEvolution(time)
```

Rotate back:

[10]:
```
# Rotate by (90°, -30°, -90°)
helium_system.rotateRho_t(alpha, -beta, gamma)
```

2.3.4 Plotting and Saving Data

Now, save and plot the data.

[11]:
```
helium_system.saveToCSV("SavedData/MetastableHeBrey1500mW.csv")
```

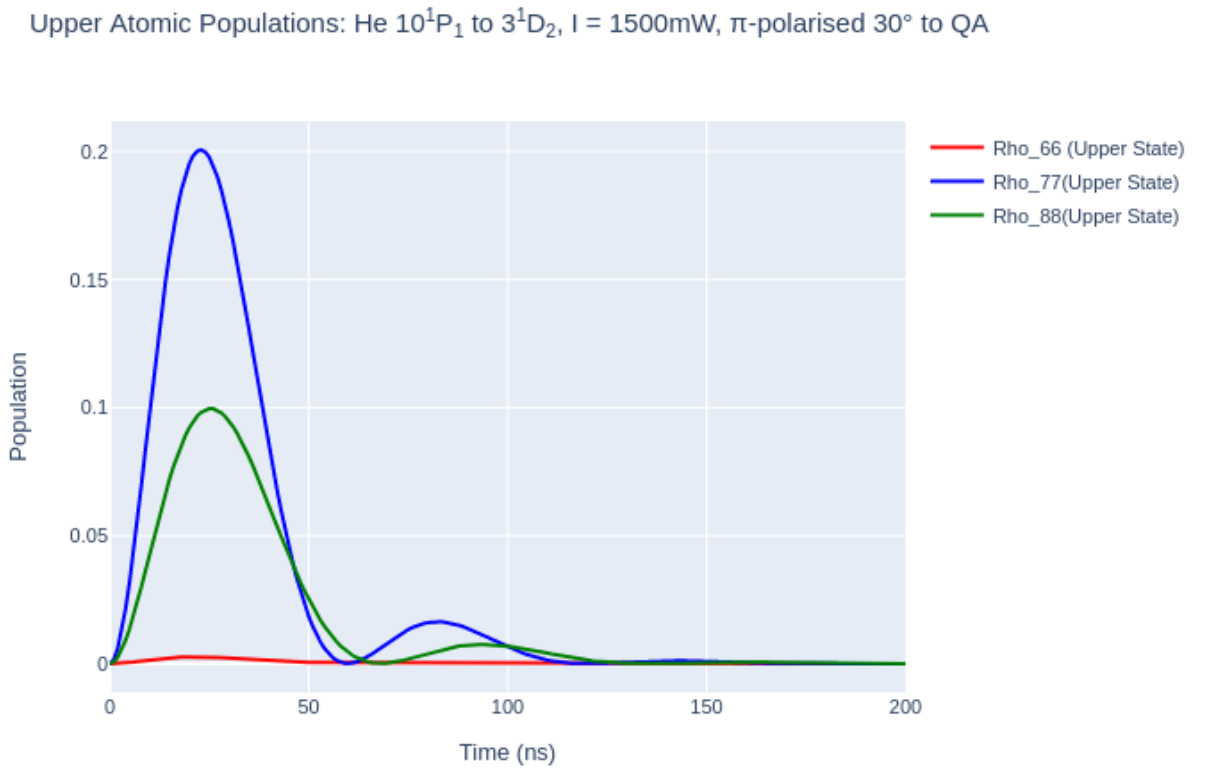
[12]:
```
rho_66 = [ abs(rho) for rho in helium_system.Rho_t(six, six)]
rho_77 = [abs(rho) for rho in helium_system.Rho_t(seven, seven)]
rho_88 = [abs(rho) for rho in helium_system.Rho_t(eight, eight)]

fig_upper = go.Figure(data = go.Scatter(x = time,
                                    y = rho_66,
                                    mode = 'lines',
                                       name = "Rho_66 (Upper State)",
                                       marker = dict(
                                    color = 'red',
                                    symbol = 'x',
                                 )))

fig_upper.add_trace(go.Scatter(x = time,
                                    y = rho_77,
                                    mode = 'lines',
                                       name = "Rho_77(Upper State)",
                                       marker = dict(
                                    color = 'blue',
                                    symbol = 'square',
                               )))
fig_upper.add_trace(go.Scatter(x = time,
                                    y = rho_88,
                                    mode = 'lines',
                                       name = "Rho_88(Upper State)",
                                       marker = dict(
                                    color = 'green',
                                    symbol = 'circle',
                               )))

fig_upper.update_layout(title = "Upper Atomic Populations: He 10<sup>1</sup>P<sub>1</sub>
↪ to 3<sup>1</sup>D<sub>2</sub>, I = {:0.0f}mW, -polarised 30° to QA".format(laser_
↪intensity, tau/1e3, tau_f/1e3),
                  xaxis_title = "Time (ns)",
                  yaxis_title = "Population",
                 font = dict(
                     size = 11))
```

(continues on next page)

(continued from previous page)

```
fig_upper.write_image("SavedPlots/tutorial3-HeFigUpper.png")
Image("SavedPlots/tutorial3-HeFigUpper.png")
```

[12]:



[13]:

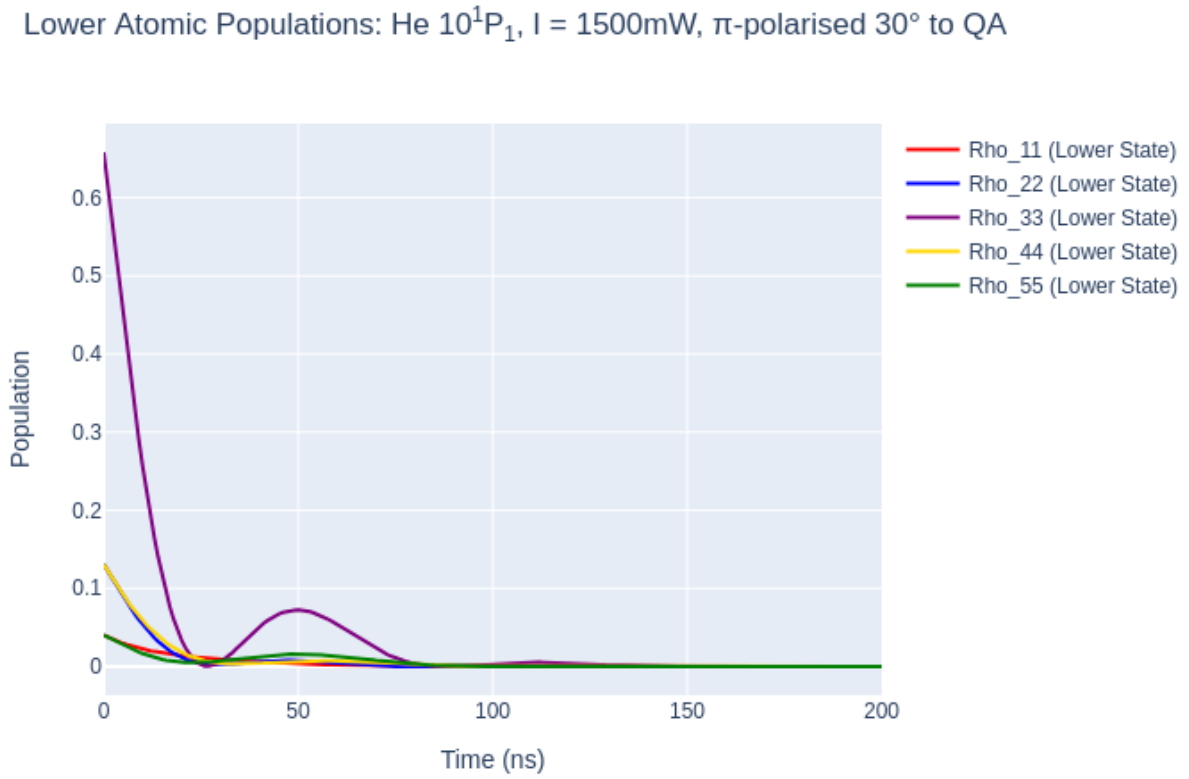
```
rho11 = [ abs(rho) for rho in helium_system.Rho_t(one, one)]
rho22 = [ abs(rho) for rho in helium_system.Rho_t(two, two)]
rho33 = [ abs(rho) for rho in helium_system.Rho_t(three, three)]
rho44 = [ abs(rho) for rho in helium_system.Rho_t(four, four)]
rho55 = [ abs(rho) for rho in helium_system.Rho_t(five, five)]


fig_lower = go.Figure(data = go.Scatter(x = time,
                                        y = rho11,
                                        mode = 'lines',
                                           name = "Rho_11 (Lower State)",
                                           marker = dict(
                                       color = 'red',
                                       symbol = 'circle',
                                    )))
fig_lower.add_trace(go.Scatter(x = time,
                                        y = rho22,
                                        mode = 'lines',
                                           name = "Rho_22 (Lower State)",
                                           marker = dict(
```

(continues on next page)

(continued from previous page)

```
                                color = 'blue',
                                symbol = 'x',
                            )))
fig_lower.add_trace(go.Scatter(x = time,
                                y = rho33,
                                mode = 'lines',
                                   name = "Rho_33 (Lower State)",
                                   marker = dict(
                                color = 'purple',
                                symbol = 'x',
                            )))
fig_lower.add_trace(go.Scatter(x = time,
                                y = rho44,
                                mode = 'lines',
                                   name = "Rho_44 (Lower State)",
                                   marker = dict(
                                color = 'gold',
                                symbol = 'x',
                            )))
fig_lower.add_trace(go.Scatter(x = time,
                                y = rho55,
                                mode = 'lines',
                                   name = "Rho_55 (Lower State)",
                                   marker = dict(
                                color = 'green',
                                symbol = 'square',
                            )))
fig_lower.update_layout(title = "Lower Atomic Populations: He 1<sup>1</sup>P<sub>1</sub>
↪, I = {:0.0f}mW, -polarised 30° to QA".format(laser_intensity, tau/1e3, tau_f/1e3),
                        xaxis_title = "Time (ns)",
                        yaxis_title = "Population")

fig_lower.write_image("SavedPlots/tutorial3-HeFigLower.png")
Image("SavedPlots/tutorial3-HeFigLower.png")
```

[13]:



2.3.5 Plotting the Angular Shape

We can plot the angular "shape" of the states $W(\theta, \phi)$ as they evolve over time Murray 2008:

$$W(\theta,\phi) = \sum_{m_J, m'_J} \rho_{m_J, m'_J} Y_{J,m_J}(\theta,\phi) Y^*_{J,m'_J}(\theta,\phi)$$

, where $Y_{J,m_J}(\theta,\phi)$ are the spherical harmonics for the state with total angular momentum J and projection m_J.

Note: this only works for a single angular momentum state so will not produce the correct results for multiple F-states but will be correct for a single J-state.

In LASED we can plot the initial shape of the atomic state using the `angularShape_0()` function which takes the argument of `theta` and `phi` lists for the azimuthal (0 to 2π) and polar coordinates (0 to π) respectively as well as a `state` argument which equals either "g" or "e" depending on whether the angular shape of the excited or lower state in the `LaserAtomSystem()` will be calculated. The "angular shape" is the radius of the electron charge cloud as the azimuthal and polar coordinate are varied.

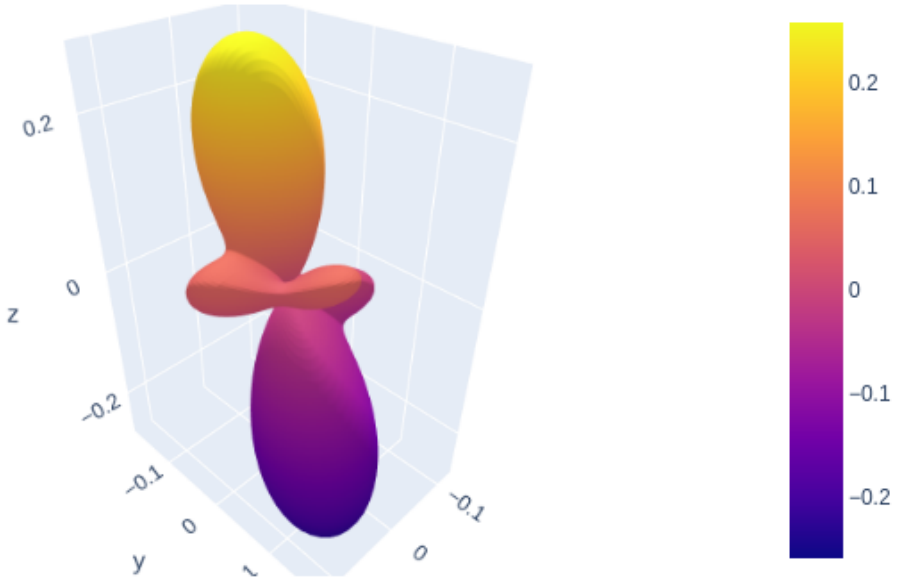
[14]:
```
phi = np.linspace(0, np.pi, 100)
theta = np.linspace(0, 2*np.pi, 100)
W = helium_system.angularShape_0("g", theta, phi)
```

The angular shape can now be plotted using the plotting package desired. Here, I use `Plotly`. Before plotting, a mesh grid must be made for `theta` and `phi` and these must be converted to cartesian coordinates along with the radial

coordinate calculated. A helpful function is the `SphericalToCartesian` function which is provided by `LASED` which takes the meshgrid and radius as inputs and provides x,y,z coordinates to be plotted.

[16]:
```
phi, theta = np.meshgrid(phi, theta)  # Create a mesh grid using numpy
x, y, z = las.SphericalToCartesian(W, theta, phi)  # Create cartesian coords
fig = go.Figure(data=[go.Surface(x=x, y=y, z=z)])  # Plot the result
fig.write_image("SavedPlots/tutorial3-HeAngularShapeDStatet=0.png")
Image("SavedPlots/tutorial3-HeAngularShapeDStatet=0.png")
```

[16]:



We can make a video for the evolution of the 3D-state's electron cloud as it evolves over time in the laser-atom system. To do this, we have to generate the angular shape of the time evolution using the function `angularShape_t` which gives an array of angular shapes. Each angular shape must then be plotted and the images saved. Once all the images are saved they must be stitched together to create a video file. This can be done using any software package, `ffmpeg` is used here.

[20]:
```
phi = np.linspace(0, np.pi, 100)
theta = np.linspace(0, 2*np.pi, 100)
W_t = helium_system.angularShape_t("g", theta, phi)
phi, theta = np.meshgrid(phi, theta)
# Generate the images
for i, W in enumerate(W_t):
    x, y, z = las.SphericalToCartesian(W, theta, phi)
    # Plot the result
    fig = go.Figure(data=[go.Surface(x=x, y=y, z=z)])
    camera = dict(
```

(continues on next page)

(continued from previous page)

```
    up=dict(x=0, y=0, z=1),
    center=dict(x=0, y=0, z=0),
    eye=dict(x=1.25, y=-1.25, z=1.5))
    fig.update_layout(scene_camera=camera)

    filenumber = "%0.5d"%i
    fig.write_image(f"Videos/3DShape" + filenumber + ".png", scale = 2.2)
```

[21]:
```
# Generate mp4 using ffpmeg
import subprocess

subprocess.call([
        'ffmpeg', '-framerate', '8', '-i', 'Videos/3DShape%05d.png', '-r', '30', '-pix_
↪fmt', 'yuv420p',
        'He3DState1500mWCollFrame30deg.mp4'
    ])
```

[21]:
```
0
```

2.4 Tutorial 4: Hyperfine Structure of Sodium D2 Line

For atoms with hyperfine structure the nuclear spin `I` is no longer zero so there is coupling between the nuclear magnetic moment of the atom and the angular momentum of the electron cloud. This gives the atom hyperfine stucture. This can be modelled by using `LASED`. The advantage of this full quantum electrodynamical treatment of the atom is that the equations of motion for an atom with hyperfine structure are more accurately described than using a semiclassical approach.

[1]:
```
import LASED as las
import numpy as np
import plotly.graph_objects as go
import time

from IPython.display import Image  # To display images in a Jupyter notebook
```

2.4.1 Hyperfine Structure Coupling

Hyperfine splitting of atomic energy levels is from the coupling of the nuclear-spin (iso-spin) I with the sum of the state's spin S and orbiatl angular momentum L. This results in a total angular momentum F:

$$F = I + J = I + L + S$$

If an atom has non-zero nuclear spin then the projection of the electron's angular momentum is m_F. This results in a different coupling between states (different dipole operator matrix element which couple two states together). The coupling between ground state |g> with angular momentum |F', m_F> and excited state |e> with angular momentum |F, m_F> is [Farrell 1995]:

$$C_{eg}^{q} = (-1)^{1/2(1+q)F+F'+J+J'+I+L+S-m_F+1}\sqrt{(2F+1)(2F'+1)(2J+1)(2J'+1)(2L+1)}\begin{pmatrix} F & 1 & F' \\ -m_F & q & m'_F \end{pmatrix}\begin{Bmatrix} J & F & I \\ F' & J' & 1 \end{Bmatrix}\begin{Bmatrix} L & J & S \\ J' & L' & 1 \end{Bmatrix}$$

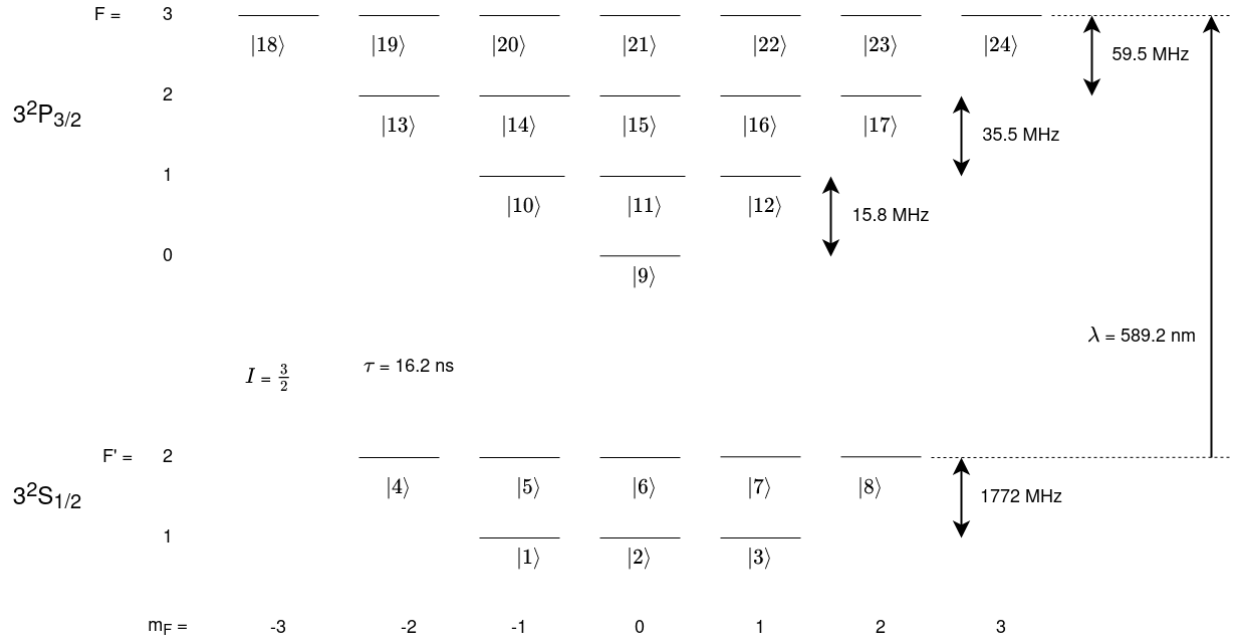
where q is the polarisation of the laser radiation: -1 if $\sigma+$, +1 if $\sigma-$, and 0 if π.

These couplings are automatically calculated when using `LASED` with the input parameters given to the `LaserAtomSystem`.

2.4.2 Sodium D_2 Line

In 1988 Farrell published a paper modelling the orange Sodium D_2 line, specifically the $3^2S_{1/2}$ (F' = 2) to the $3^2P_{3/2}$ (F = 3,2,1) transition. See Farrell's paper here. Farrell also compares the QED approach to the semi-classical (SC) model and shows that the SC model agrees with the QED model at low intensities but the approaches diverge at higher intensities. In this tutorial we will reproduce Farrell's results.

The data for the Sodium D line can be found here. A level diagram of the sructure of this Sodium transition is displayed below.



As you can see by the diagram above, each ground and excited state has multiple sub-states at different energies. The differing ernergies of the sub-states can be declared by using the keyword `w` and entering the angular frequency in Grad/s corresponding to the energy when creating `State` objects. These energies are relative so a zero point must be defined. In this example the zero energy is the energy of the F' = 2 ground state.

Since there are many states and it is cumbersome to write out all states explicitly `LASED` has a function to generate all the sub-states for a given set of quantum numbers and angular frequency. This function is the `generateSubStates(label_from, w, L, S, I, F)` function. The `label_from` keyword defines the`label` attached to each sub-state generated. The function will generate the sub-state with the least `m` value (projection of total angular momentum) first with the integer given with the keyword `label_from` and generate the rest of the states in ascending order of `m`, giving each `State` an increasing label number.

[2]:
```
# 3^2S_{1/2} -> 3^2P_{3/2}
wavelength_na = 589.159e-9  # Wavelength in m
w_e = las.angularFreq(wavelength_na)
tau_na = 16.24  # in ns

I_Na = 3/2  # Isospin for sodium
```

(continues on next page)

(continued from previous page)

```
PI = np.pi

# Energy Splittings
w1 = 1.77*2*PI # Splitting of 3^2S_{1/2}(F = 1) - (F = 2) in Grad/s
w2 = 0.0158*2*PI  # Splitting between 3^2P_{3/2} (F = 0) and F = 1 in Grad/s
w3 = 0.0355*2*PI  # Splitting between 3^2P_{3/2} (F = 1) and F = 2 in Grad/s
w4 = 0.0595*2*PI  # Splitting between 3^2P_{3/2} (F = 2) and F = 3 in Grad/s

# Detunings
w_Fp1 = -1*w1
w_F0 = w_e-(w4+w3+w2)
w_F1 = w_e-(w4+w3)
w_F2 = w_e-w4
w_F3 = w_e

# Model
# 3^2S_{1/2} F' = 1
Fp1 = las.generateSubStates(label_from = 1, w = w_Fp1, L = 0, S = 1/2, I = I_Na, F = 1)
# 3^2S_{1/2} F' = 2
Fp2 = las.generateSubStates(label_from = 4, w = 0, L = 0, S = 1/2, I = I_Na, F = 2)
# 3^2P_{3/2} F = 0
F0 = las.generateSubStates(label_from = 9, w = w_F0, L = 1, S = 1/2, I = I_Na, F = 0)
# 3^2P_{3/2} F = 1
F1 = las.generateSubStates(label_from = 10, w = w_F1, L = 1, S = 1/2, I = I_Na, F = 1)
# 3^2P_{3/2} F = 2
F2 = las.generateSubStates(label_from = 13, w = w_F2, L = 1, S = 1/2, I = I_Na, F = 2)
# 3^2P_{3/2} F = 3
F3 = las.generateSubStates(label_from = 18, w = w_F3, L = 1, S = 1/2, I = I_Na, F = 3)

G_na = Fp1 + Fp2
E_na = F0 + F1 + F2 + F3

# Laser parameters
intensity_na = 85.6 # mW/mm^-2
Q_na = [0]

# Simulation parameters
start_time = 0
stop_time = 500 # in ns
time_steps = 501
time_na = np.linspace(start_time, stop_time, time_steps)
```

Now, we can declare the `LaserAtomSystem`. The timing code does not have to be input and shows how much time a 24-level system takes to solve.

[3]:
```
sodium_system = las.LaserAtomSystem(E_na, G_na, tau_na, Q_na, wavelength_na,
                                    laser_intensity = intensity_na)
tic = time.perf_counter()
sodium_system.timeEvolution(time_na)
toc = time.perf_counter()
print(f"The code finished in {toc-tic:0.4f} seconds")
```

```
Populating ground states equally as the initial condition.
The code finished in 283.0380 seconds
```

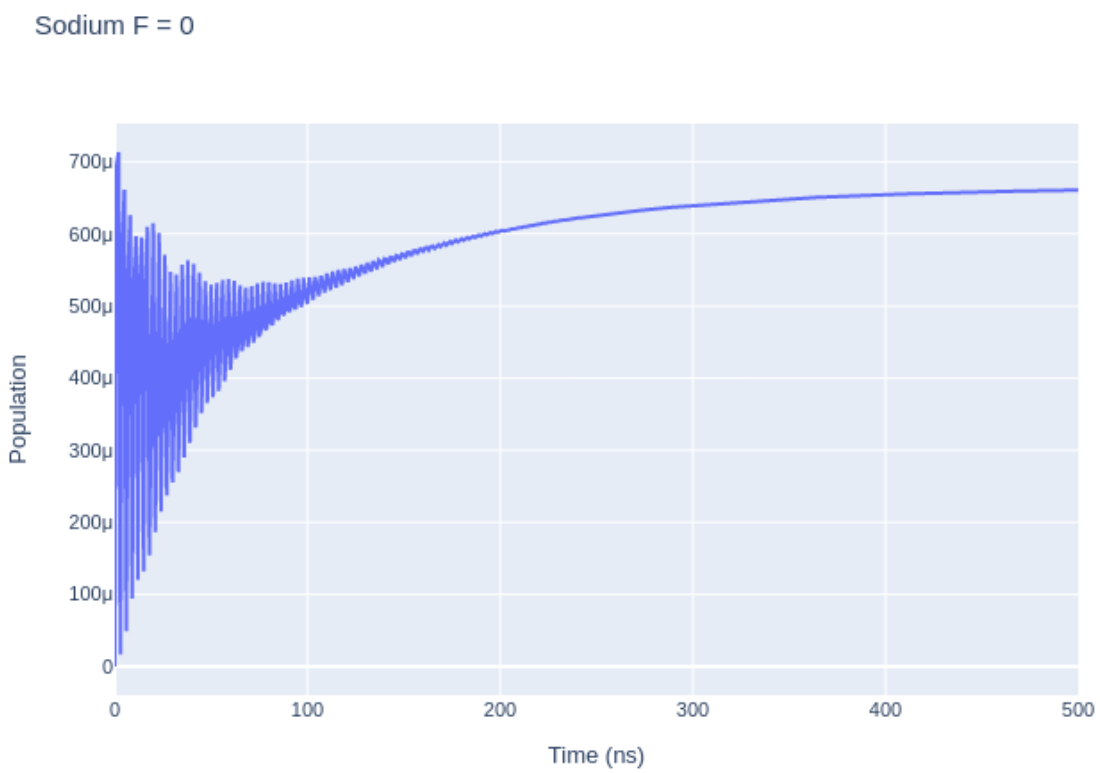
2.4.3 Saving and Plotting

Save the data as a csv file and plot the data.

[4]:
```
sodium_system.saveToCSV("SavedData/SodiumD2Line86mW.csv")
```

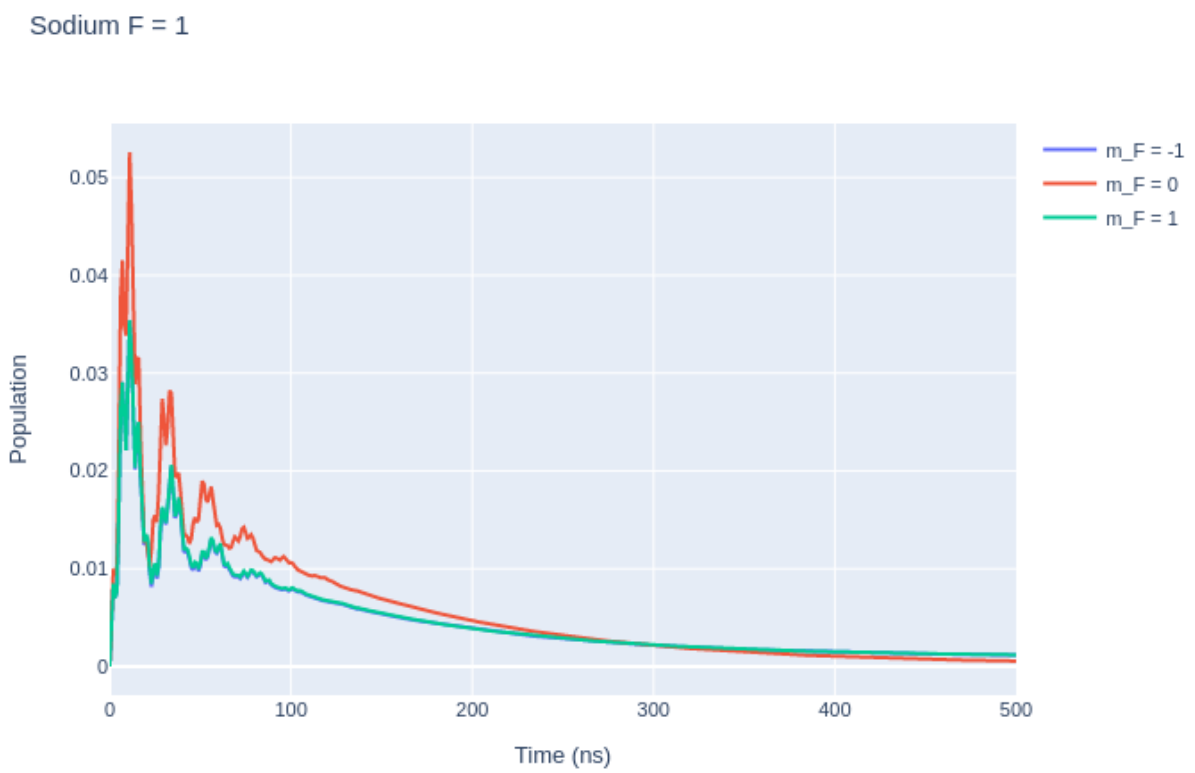
[5]:
```
las_sys = sodium_system

rho_to_plot = [ [abs(rho) for rho in las_sys.Rho_t(s, s)] for s in F0]

fig_na = go.Figure()

for i, rho_ee in enumerate(rho_to_plot):
    fig_na.add_trace(go.Scatter(x = time_na,
                                y = rho_ee,
                                name = f"m_F = {F0[i].m}",
                               mode = 'lines'))

fig_na.update_layout(title = "Sodium F = 0",
                 xaxis_title = "Time (ns)",
                 yaxis_title = "Population",
                font = dict(
                    size = 11))
fig_na.write_image("SavedPlots/NaF=0I=856.png")
Image("SavedPlots/NaF=0I=856.png")
```

[5]:



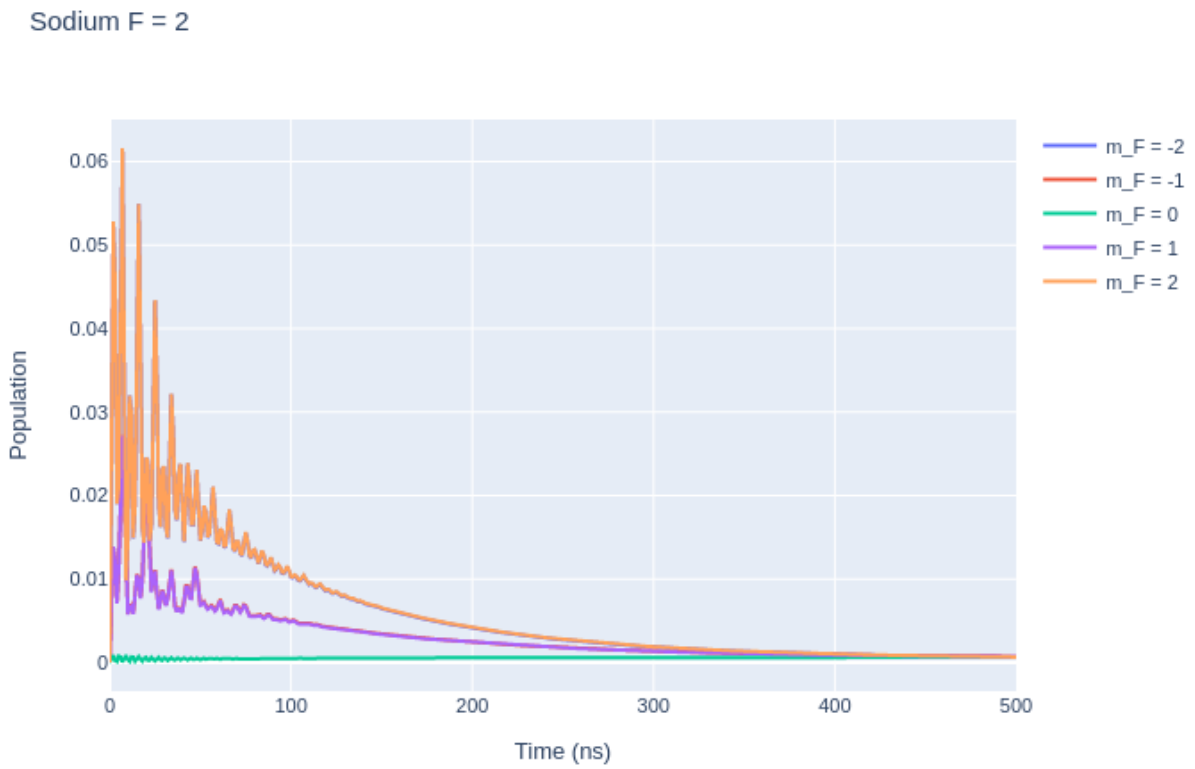
[6]:
```
las_sys = sodium_system

rho_to_plot = [ [abs(rho) for rho in las_sys.Rho_t(s, s)] for s in F1]

fig_na = go.Figure()

for i, rho_ee in enumerate(rho_to_plot):
    fig_na.add_trace(go.Scatter(x = time_na,
                                y = rho_ee,
                                name = f"m_F = {F1[i].m}",
                               mode = 'lines'))

fig_na.update_layout(title = "Sodium F = 1",
                 xaxis_title = "Time (ns)",
                 yaxis_title = "Population",
                font = dict(
                    size = 11))
fig_na.write_image("SavedPlots/NaF=1I=856.png")
Image("SavedPlots/NaF=1I=856.png")
```

[6]:



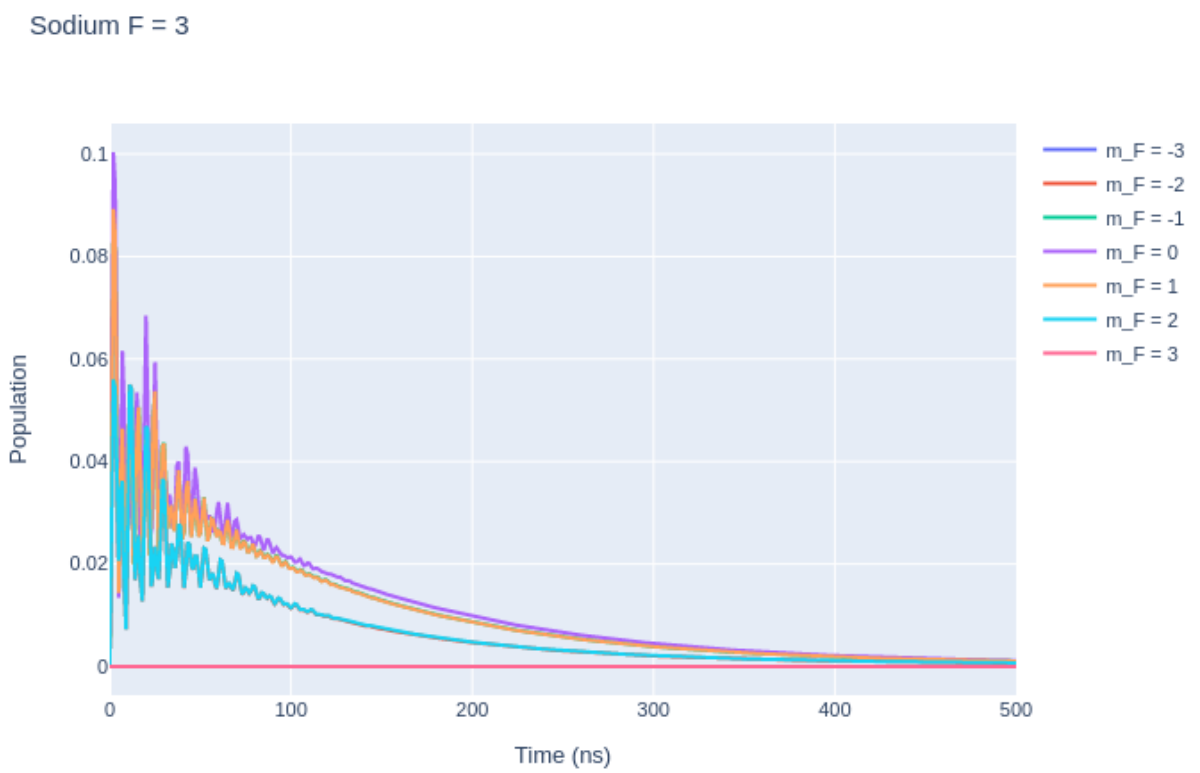
[7]:
```
las_sys = sodium_system

rho_to_plot = [ [abs(rho) for rho in las_sys.Rho_t(s, s)] for s in F2]

fig_na = go.Figure()

for i, rho_ee in enumerate(rho_to_plot):
    fig_na.add_trace(go.Scatter(x = time_na,
                                y = rho_ee,
                                name = f"m_F = {F2[i].m}",
                               mode = 'lines'))

fig_na.update_layout(title = "Sodium F = 2",
                 xaxis_title = "Time (ns)",
                 yaxis_title = "Population",
                font = dict(
                    size = 11))
fig_na.write_image("SavedPlots/NaF=2I=856.png")
Image("SavedPlots/NaF=2I=856.png")
```

[7]:



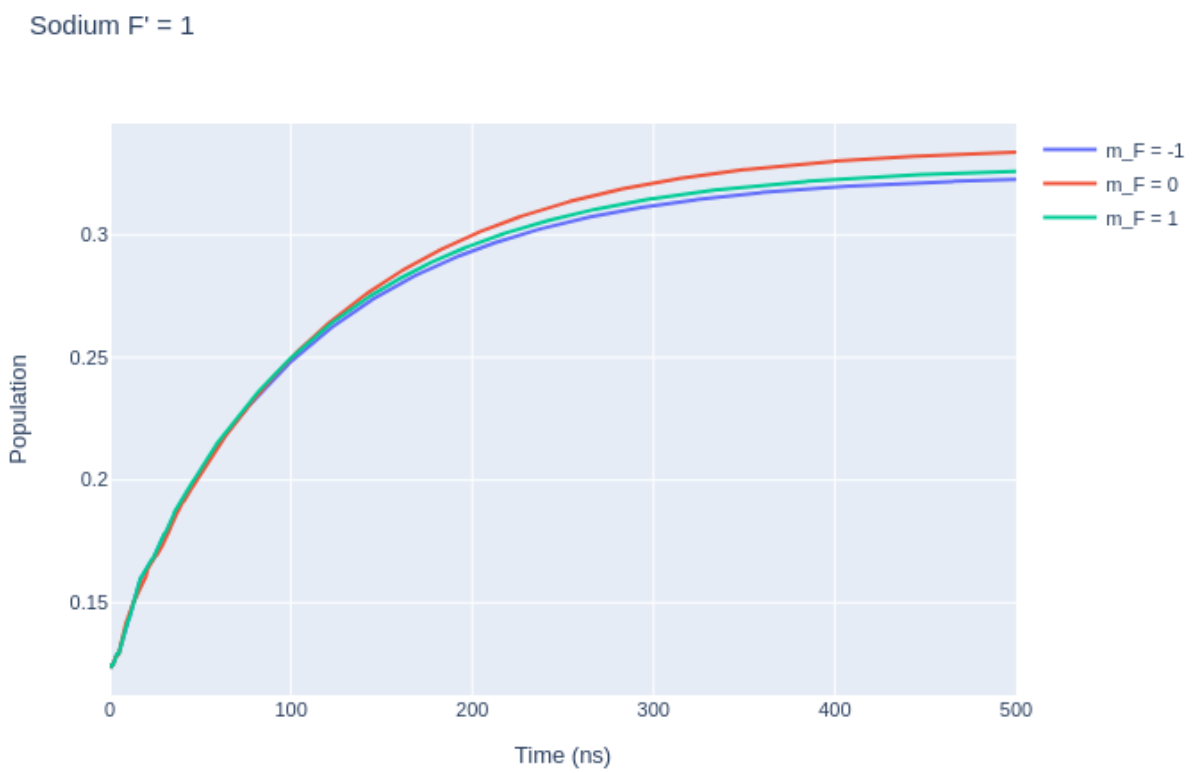
[8]:
```
las_sys = sodium_system

rho_to_plot = [ [abs(rho) for rho in las_sys.Rho_t(s, s)] for s in F3]

fig_na = go.Figure()

for i, rho_ee in enumerate(rho_to_plot):
    fig_na.add_trace(go.Scatter(x = time_na,
                                y = rho_ee,
                                name = f"m_F = {F3[i].m}",
                               mode = 'lines'))

fig_na.update_layout(title = "Sodium F = 3",
                 xaxis_title = "Time (ns)",
                 yaxis_title = "Population",
                font = dict(
                    size = 11))
fig_na.write_image("SavedPlots/NaF=3I=856.png")
Image("SavedPlots/NaF=3I=856.png")
```

[8]:



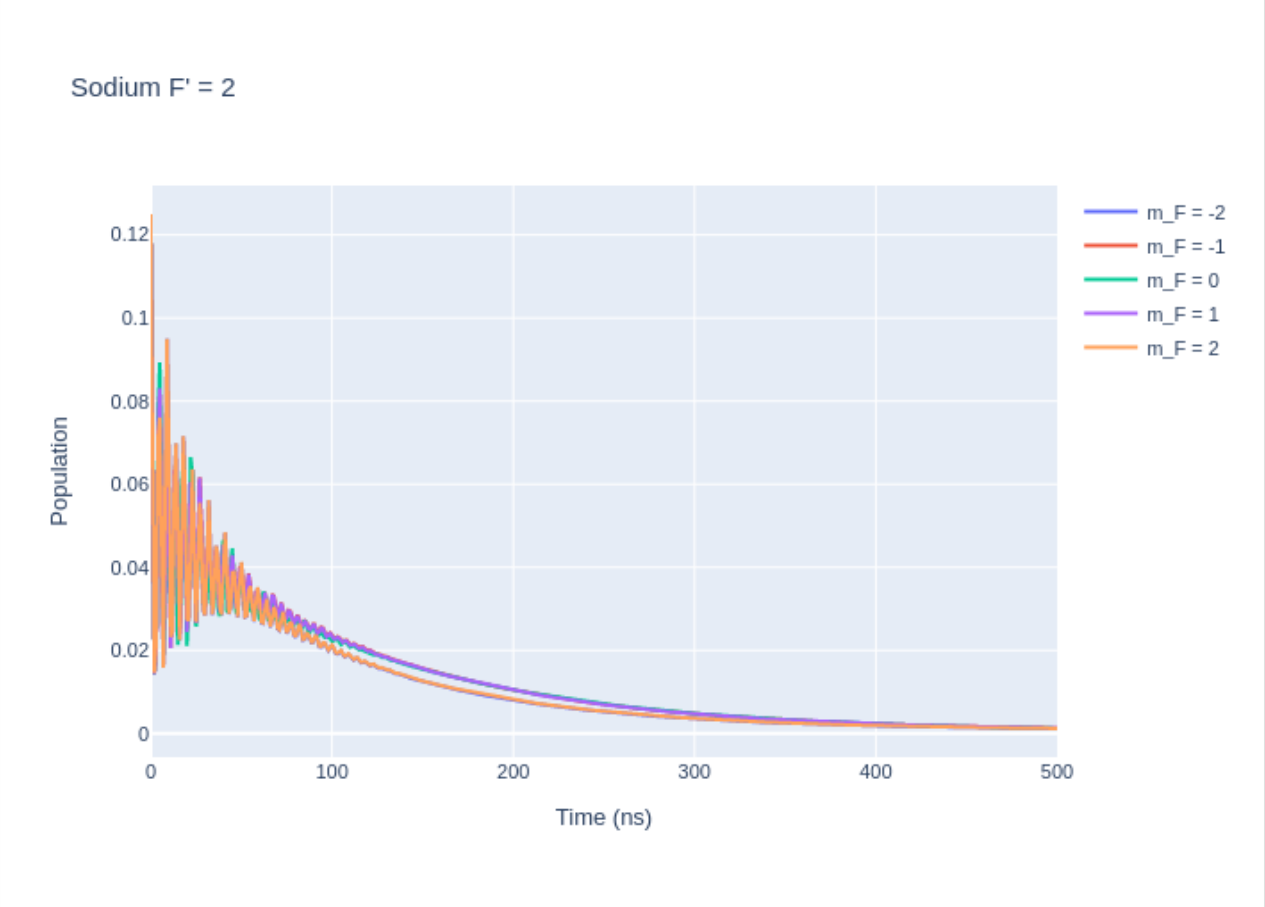
[9]:
```
fig_na_lower = go.Figure()

rho_to_plot = [ [abs(rho) for rho in las_sys.Rho_t(s, s)] for s in Fp1]

for i, rho_gg in enumerate(rho_to_plot):
    fig_na_lower.add_trace(go.Scatter(x = time_na,
                                      y = rho_gg,
                                      name = f"m_F = {Fp1[i].m}",
                                     mode = 'lines'))
fig_na_lower.update_layout(title = "Sodium F' = 1",
                  xaxis_title = "Time (ns)",
                  yaxis_title = "Population",
                 font = dict(
                     size = 11))
fig_na_lower.write_image("SavedPlots/NaFp=1I=856.png")
Image("SavedPlots/NaFp=1I=856.png")
```

[9]:



[10]:
```
fig_na_lower = go.Figure()

rho_to_plot = [ [abs(rho) for rho in las_sys.Rho_t(s, s)] for s in Fp2]

for i, rho_gg in enumerate(rho_to_plot):
    fig_na_lower.add_trace(go.Scatter(x = time_na,
                                      y = rho_gg,
                                      name = f"m_F = {Fp2[i].m}",
                                     mode = 'lines'))
fig_na_lower.update_layout(title = "Sodium F' = 2",
                  xaxis_title = "Time (ns)",
                  yaxis_title = "Population",
                 font = dict(
                     size = 11))
fig_na_lower.write_image("SavedPlots/NaFp=2I=856.png")
Image("SavedPlots/NaFp=2I=856.png")
```

[10]:

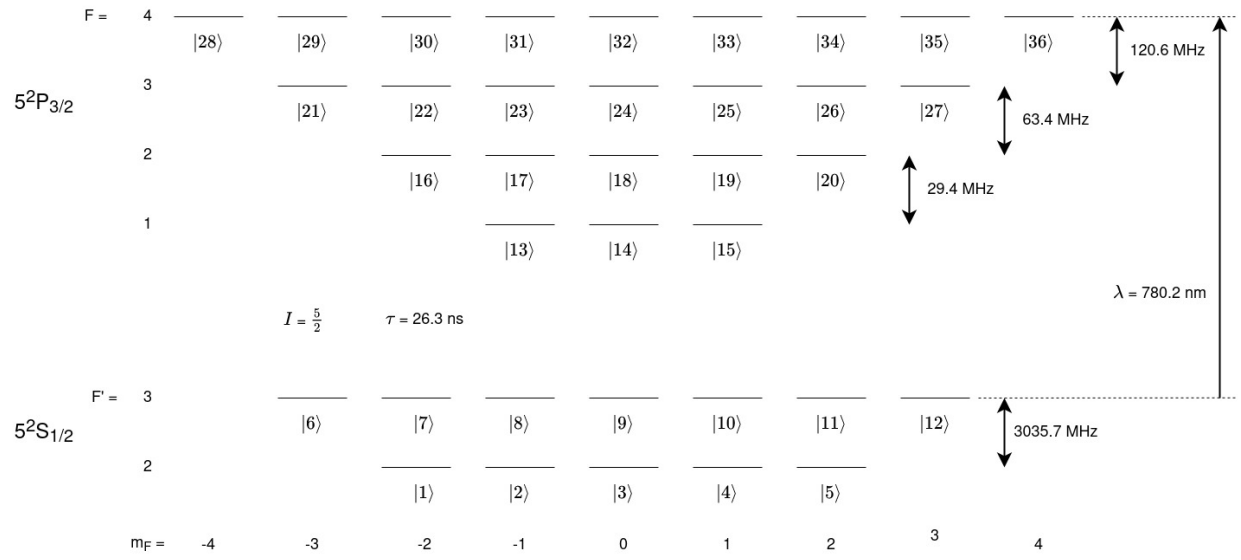


2.5 Tutorial 5: Rubidium-85 D Line

Rubidium is a heavy atom with hyperfine structure. Modelling of the Rb85 D line is shown in this tutorial using the LASED library. The calculations will be compared with J.Pursehouse's PhD thesis as he modelled the same system.

[1]:
```
import LASED as las
import plotly.graph_objects as go
import numpy as np
import time

from IPython.display import Image  # To display images in Jupyter notebooks
```

The data for Rubidium can be accessed here. The $5^2S_{1/2}$ to the $5^2P_{3/2}$ transition will be modelled here with F' = 3 to the F = 4 level as the resonant transition and all other terms will be detuned from this resonance. A level diagram of the system is shown below. This system has more energy levels than the sodium D-line.



2.5.1 Setting up the System

Set up the system in the exact same way that the sodium system was setup but with different numbers.

[2]:
```
# 5^2S_{1/2} -> 5^2P_{3/2}
wavelength_rb = 780.2e-9  # Wavelength in nm
w_e = las.angularFreq(wavelength_rb)
tau_rb = 26.3 # in ns

I_rb = 5/2  # Isospin for sodium
PI = np.pi

# Energy Splittings
w1 = 3.0357*2*PI # Splitting of 5^2S_{1/2}(F' = 2) -> (F' = 3) in Grad/s
w2 = 0.0294*2*PI  # Splitting between 5^2P_{3/2} F = 1 and F = 2 in Grad/s
w3 = 0.0634*2*PI  # Splitting between 5^2P_{3/2} F = 2 and F = 3 in Grad/s
w4 = 0.1206*2*PI  # Splitting between 5^2P_{3/2} F = 3 and F = 3 in Grad/s

# Detunings
w_Fp2 = -1*w1
w_F1 = w_e-(w4+w3+w2)
w_F2 = w_e-(w4+w3)
w_F3 = w_e-w4
w_F4 = w_e

# Create states
# 5^2S_{1/2}
Fp2 = las.generateSubStates(label_from = 1, w = w_Fp2, L = 0, S = 1/2, I = I_rb, F = 2)
Fp3 = las.generateSubStates(label_from = 6, w = 0, L = 0, S = 1/2, I = I_rb, F = 3)

# 5^2P_{3/2}
F1 = las.generateSubStates(label_from = 13, w = w_F1, L = 1, S = 1/2, I = I_rb, F = 1)
F2 = las.generateSubStates(label_from = 16, w = w_F2, L = 1, S = 1/2, I = I_rb, F = 2)
F3 = las.generateSubStates(label_from = 21, w = w_F3, L = 1, S = 1/2, I = I_rb, F = 3)
```

(continues on next page)

(continued from previous page)

```
F4 = las.generateSubStates(label_from = 28, w = w_F4, L = 1, S = 1/2, I = I_rb, F = 4)

# Declare excited and ground states
G_rb = Fp2 + Fp3
E_rb = F1 + F2 + F3 + F4

# Laser parameters
intensity_rb = 20 # mW/mm^-2
Q_rb = [0]
Q_decay = [1, 0, -1]

# Simulation parameters
start_time = 0
stop_time = 500 # in ns
time_steps = 501
time_rb = np.linspace(start_time, stop_time, time_steps)
```

2.5.2 Time Evolution

The time evolution of this system will take much longer than the sodium as the complexity scales as n^2 where n is the number of energy levels and sodium has 24 compared to rubidium's 36.

Create a `LaserAtomSystem` object and time evolve the system using `timeEvolution()`

[3]:
```
rb_system = las.LaserAtomSystem(E_rb, G_rb, tau_rb, Q_rb, wavelength_rb, laser_intensity␣
↪= intensity_rb)
tic = time.perf_counter()
rb_system.timeEvolution(time_rb)
toc = time.perf_counter()
print(f"The code finished in {toc-tic:0.4f} seconds")
```

```
Populating ground states equally as the initial condition.
The code finished in 2268.3734 seconds
```

2.5.3 Saving and Plotting

Now save and plot the results

[4]:
```
rb_system.saveToCSV("SavedData/RubidiumDLine20mW.csv")
```

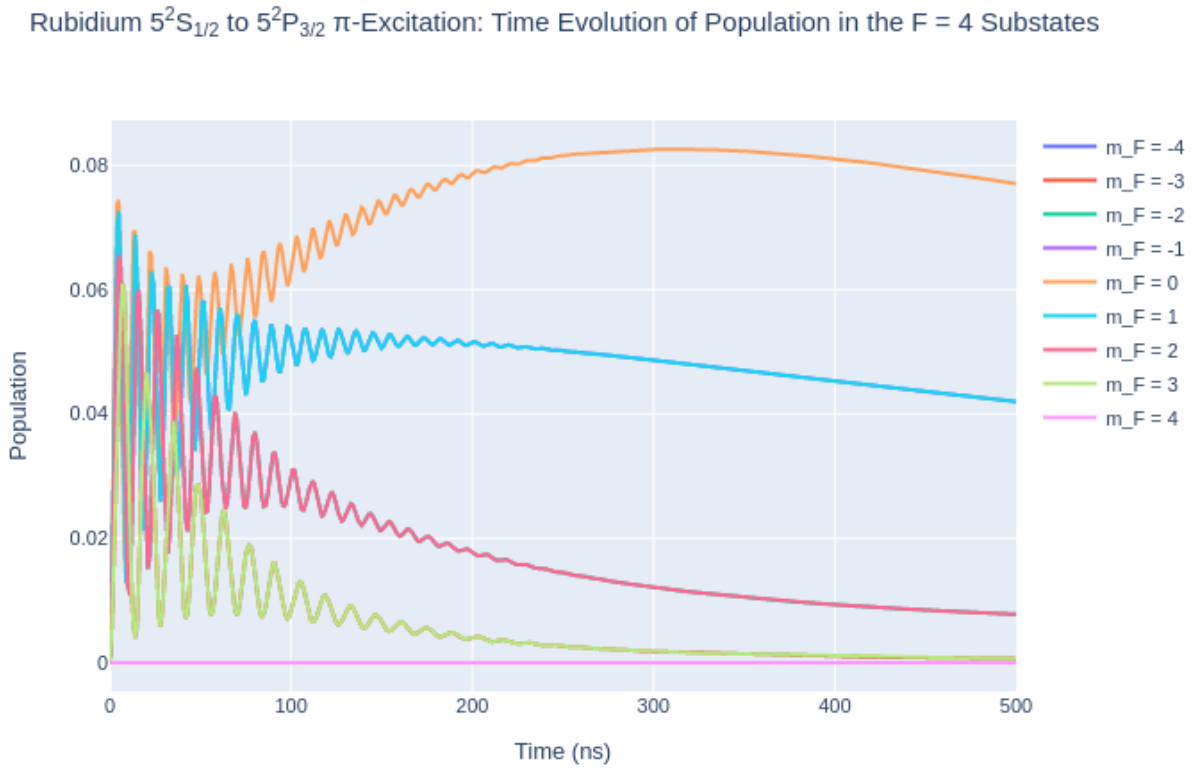
[5]:
```
rho_to_plot = [ [abs(rho) for rho in rb_system.Rho_t(s, s)] for s in F4]

fig_rbF4 = go.Figure()

for i, rho in enumerate(rho_to_plot):
    fig_rbF4.add_trace(go.Scatter(x = time_rb,
                                  y = rho,
                                  name = f"m_F = {F4[i].m}",
                                 mode = 'lines'))
```

(continues on next page)

(continued from previous page)

```
fig_rbF4.update_layout(title = "Rubidium 5<sup>2</sup>S<sub>1/2</sub> to 5<sup>2</sup>P
↪<sub>3/2</sub> -Excitation: Time Evolution of Population in the F = 4 Substates",
                  xaxis_title = "Time (ns)",
                  yaxis_title = "Population",
                 font = dict(
                     size = 11))
fig_rbF4.write_image(f"SavedPlots/RbF=4I={intensity_rb}.png")
Image(f"SavedPlots/RbF=4I={intensity_rb}.png")
```

[5]:



[6]:

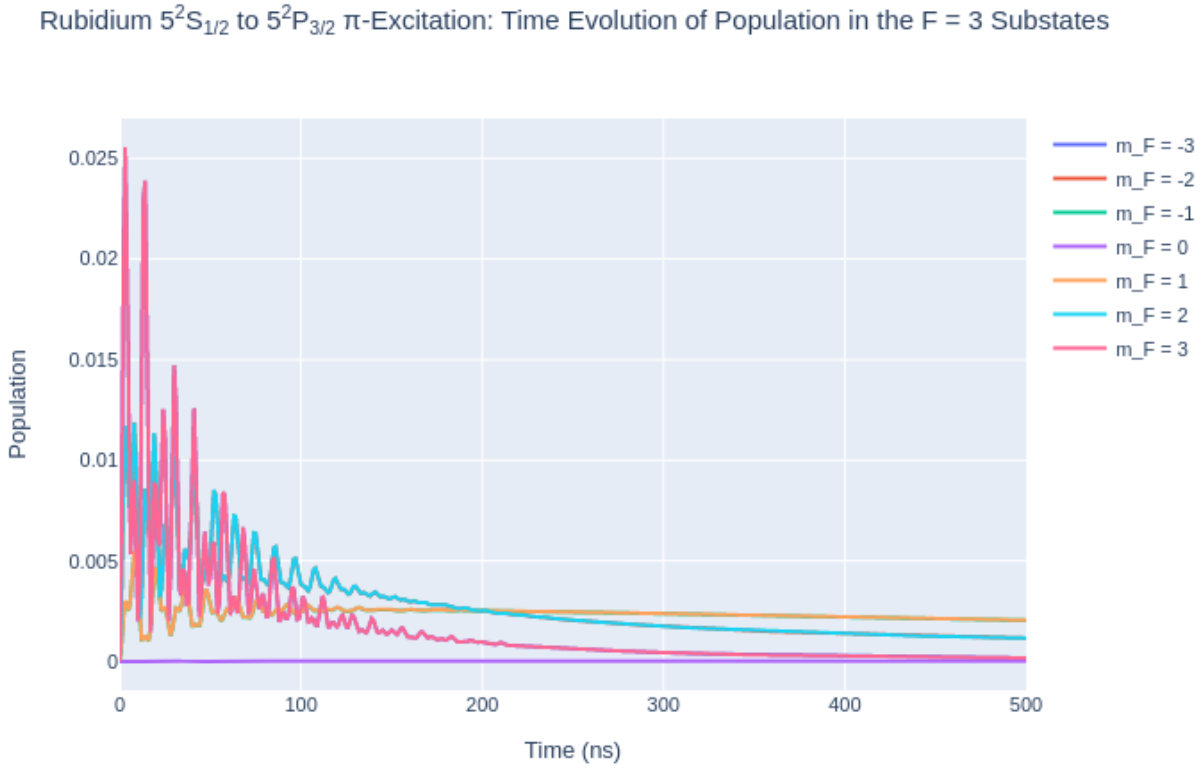
```
rho_to_plot = [ [abs(rho) for rho in rb_system.Rho_t(s, s)] for s in F3]

fig_rbF3 = go.Figure()

for i, rho in enumerate(rho_to_plot):
    fig_rbF3.add_trace(go.Scatter(x = time_rb,
                                  y = rho,
                                  name = f"m_F = {F3[i].m}",
                                 mode = 'lines'))

fig_rbF3.update_layout(title = "Rubidium 5<sup>2</sup>S<sub>1/2</sub> to 5<sup>2</sup>P
↪<sub>3/2</sub> -Excitation: Time Evolution of Population in the F = 3 Substates",
                  xaxis_title = "Time (ns)",
                  yaxis_title = "Population",
```

(continues on next page)

(continued from previous page)

```
                font = dict(
                    size = 11))
fig_rbF3.write_image(f"SavedPlots/RbF=3I={intensity_rb}.png")
Image(f"SavedPlots/RbF=3I={intensity_rb}.png")
```

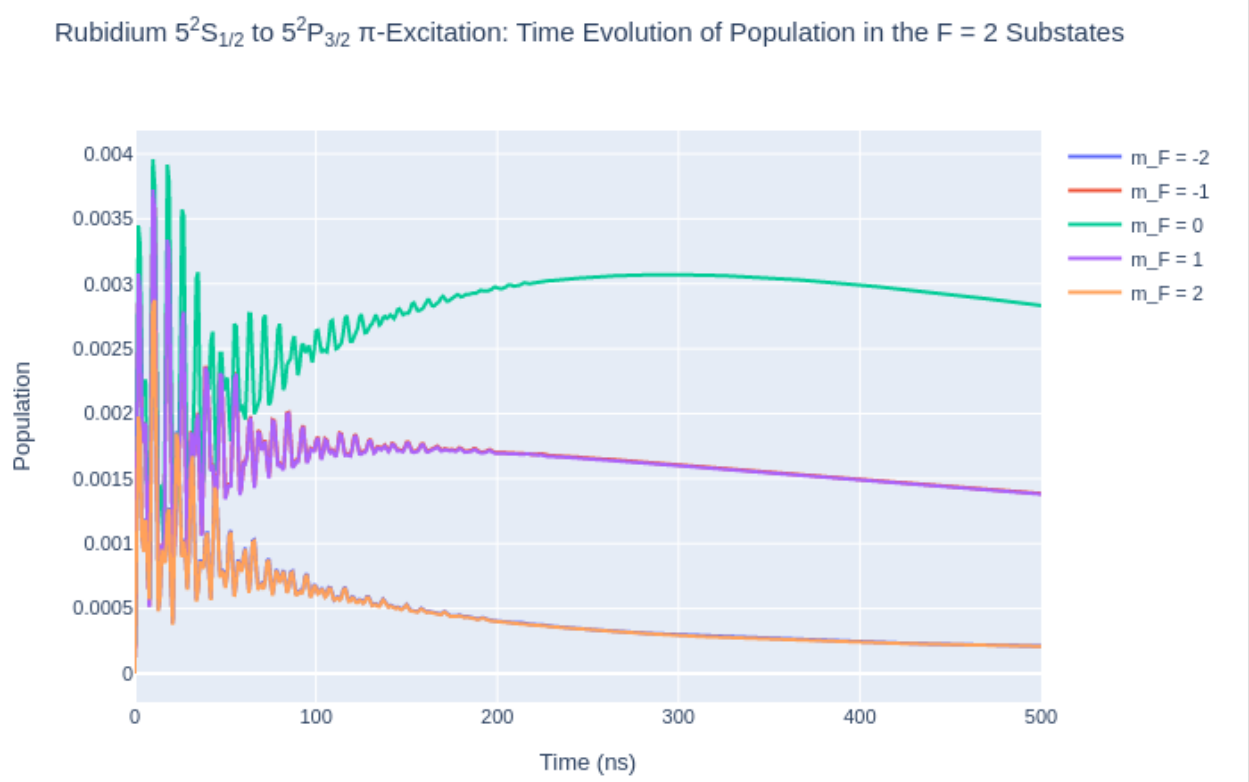
[6]:



[7]:

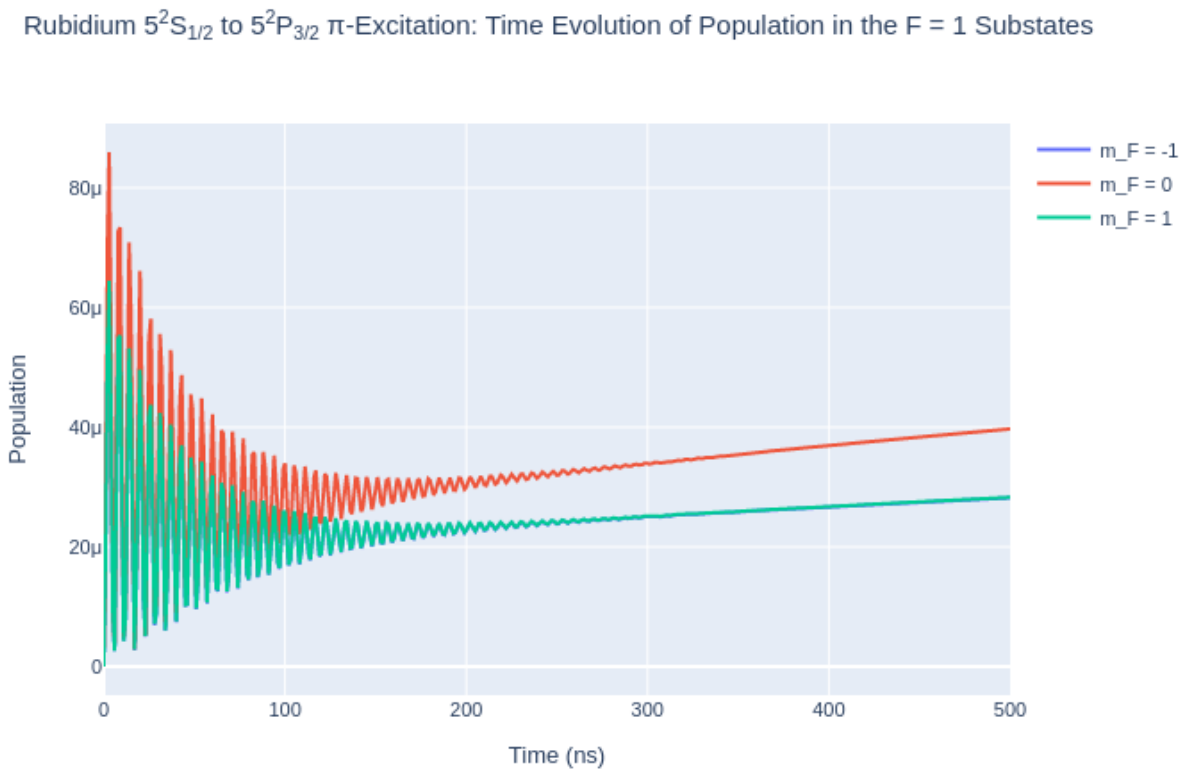
```
rho_to_plot = [ [abs(rho) for rho in rb_system.Rho_t(s, s)] for s in F2]

fig_rbF2 = go.Figure()

for i, rho in enumerate(rho_to_plot):
    fig_rbF2.add_trace(go.Scatter(x = time_rb,
                                  y = rho,
                                  name = f"m_F = {F2[i].m}",
                                 mode = 'lines'))

fig_rbF2.update_layout(title = "Rubidium 5<sup>2</sup>S<sub>1/2</sub> to 5<sup>2</sup>P
↪<sub>3/2</sub> -Excitation: Time Evolution of Population in the F = 2 Substates",
                       xaxis_title = "Time (ns)",
                       yaxis_title = "Population",
                      font = dict(
                          size = 11))
fig_rbF2.write_image(f"SavedPlots/RbF=2I={intensity_rb}.png")
Image(f"SavedPlots/RbF=2I={intensity_rb}.png")
```

[7]:



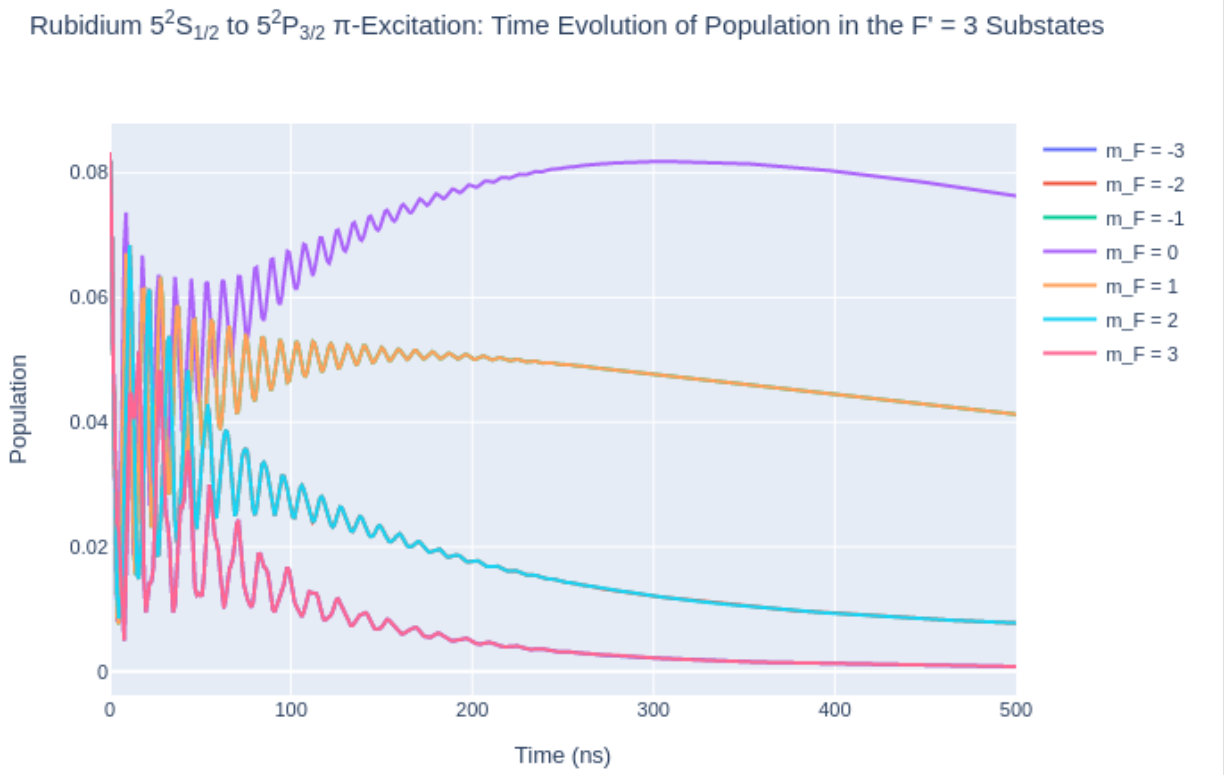
[8]:
```
rho_to_plot = [ [abs(rho) for rho in rb_system.Rho_t(s, s)] for s in F1]

fig_rbF1 = go.Figure()

for i, rho in enumerate(rho_to_plot):
    fig_rbF1.add_trace(go.Scatter(x = time_rb,
                                  y = rho,
                                  name = f"m_F = {F1[i].m}",
                                 mode = 'lines'))

fig_rbF1.update_layout(title = "Rubidium 5<sup>2</sup>S<sub>1/2</sub> to 5<sup>2</sup>P
↪<sub>3/2</sub> -Excitation: Time Evolution of Population in the F = 1 Substates",
                  xaxis_title = "Time (ns)",
                  yaxis_title = "Population",
                 font = dict(
                     size = 11))
fig_rbF1.write_image(f"SavedPlots/RbF=1I={intensity_rb}.png")
Image(f"SavedPlots/RbF=1I={intensity_rb}.png")
```

[8]:



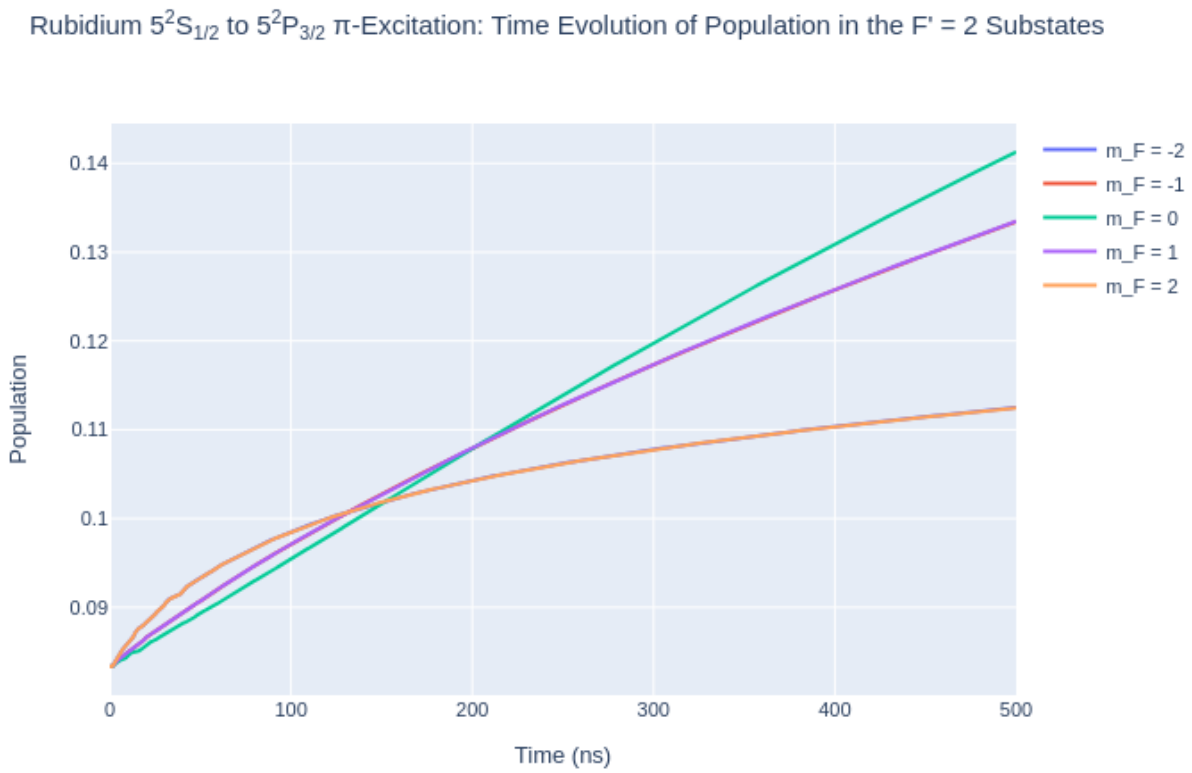
[9]:
```
rho_to_plot = [ [abs(rho) for rho in rb_system.Rho_t(s, s)] for s in Fp3]

fig_rbFp3 = go.Figure()

for i, rho in enumerate(rho_to_plot):
    fig_rbFp3.add_trace(go.Scatter(x = time_rb,
                                   y = rho,
                                   name = f"m_F = {Fp3[i].m}",
                                  mode = 'lines'))

fig_rbFp3.update_layout(title = "Rubidium 5<sup>2</sup>S<sub>1/2</sub> to 5<sup>2</sup>P
↪<sub>3/2</sub> -Excitation: Time Evolution of Population in the F' = 3 Substates",
                 xaxis_title = "Time (ns)",
                 yaxis_title = "Population",
                font = dict(
                    size = 11))
fig_rbFp3.write_image(f"SavedPlots/RbFp=3I={intensity_rb}.png")
Image(f"SavedPlots/RbFp=3I={intensity_rb}.png")
```

[9]:



[10]:
```
rho_to_plot = [ [abs(rho) for rho in rb_system.Rho_t(s, s)] for s in Fp2]

fig_rbFp2 = go.Figure()

for i, rho in enumerate(rho_to_plot):
    fig_rbFp2.add_trace(go.Scatter(x = time_rb,
                                   y = rho,
                                   name = f"m_F = {Fp2[i].m}",
                                  mode = 'lines'))

fig_rbFp2.update_layout(title = "Rubidium 5<sup>2</sup>S<sub>1/2</sub> to 5<sup>2</sup>P
↪<sub>3/2</sub> -Excitation: Time Evolution of Population in the F' = 2 Substates",
                 xaxis_title = "Time (ns)",
                 yaxis_title = "Population",
                font = dict(
                    size = 11))
fig_rbFp2.write_image(f"SavedPlots/RbFp=2I={intensity_rb}.png")
Image(f"SavedPlots/RbFp=2I={intensity_rb}.png")
```

[10]:

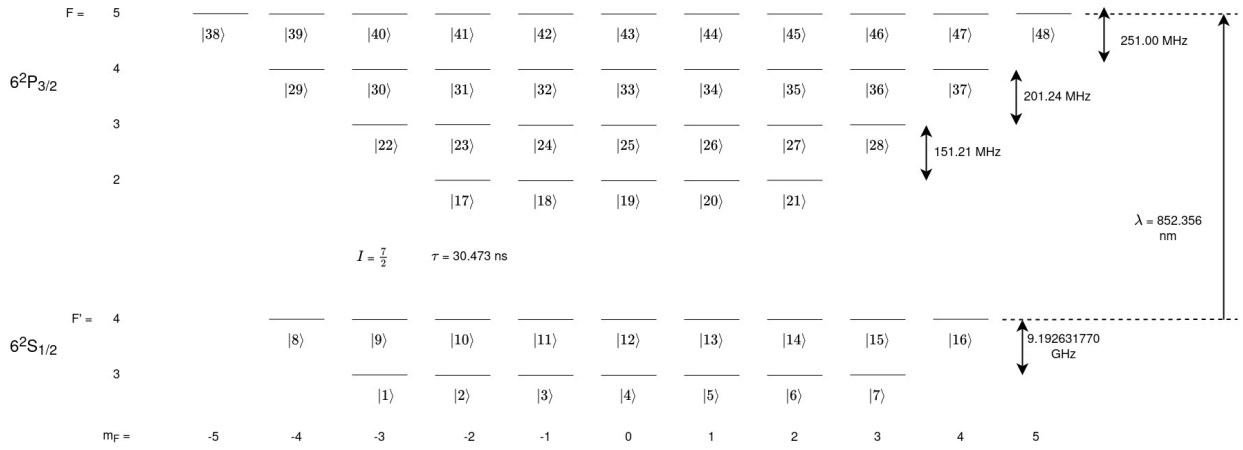


2.6 Tutorial 6: Caesium-133 D Line

Caesium is a heavy atom with hyperfine structure. Modelling of the 133Cs D line is shown in this tutorial using the LASED library.

[1]:
```
import LASED as las
import plotly.graph_objects as go
import time
import numpy as np

from IPython.display import Image  # To display images in Jupyter notebook
```

The data for Caesium can be accessed here. The $6^2S_{1/2}$ to the $6^2P_{3/2}$ transition will be modelled here with F' = 4 to the F = 5 level as the resonant transition and all other terms will be detuned from this resonance. A level diagram of the system being modelled is shown below.



2.6.1 Setting up the System

[2]:
```
# 6^2S_{1/2} -> 6^2P_{3/2}
wavelength_cs = 852.356e-9  # Wavelength in nm
w_e = las.angularFreq(wavelength_cs)
tau_cs = 30.473 # in ns

I_cs = 7/2  # Isospin for sodium
PI = np.pi

# Energy Splittings
w1 = 9.192631770*2*PI # Splitting of 6^2S_{1/2}(F' = 3) -> (F' = 4) in Grad/s (Exact due␣
↪to definition of the second)
w2 = 0.15121*2*PI  # Splitting between 6^2P_{3/2} F = 2 and F = 3 in Grad/s
w3 = 0.20124*2*PI  # Splitting between 6^2P_{3/2} F = 3 and F = 4 in Grad/s
w4 = 0.251*2*PI  # Splitting between 6^2P_{3/2} F = 4 and F = 5 in Grad/s

# Detunings
w_Fp3 = -1*w1
w_F2 = w_e-(w4+w3+w2)
w_F3 = w_e-(w4+w3)
w_F4 = w_e-w4
w_F5 = w_e

# Create states
# 6^2S_{1/2}
Fp3 = las.generateSubStates(label_from = 1, w = w_Fp3, L = 0, S = 1/2, I = I_cs, F = 3)
Fp4 = las.generateSubStates(label_from = 8, w = 0, L = 0, S = 1/2, I = I_cs, F = 4)

# 5^2P_{3/2}
F2 = las.generateSubStates(label_from = 17, w = w_F2, L = 1, S = 1/2, I = I_cs, F = 2)
F3 = las.generateSubStates(label_from = 22, w = w_F3, L = 1, S = 1/2, I = I_cs, F = 3)
F4 = las.generateSubStates(label_from = 29, w = w_F4, L = 1, S = 1/2, I = I_cs, F = 4)
F5 = las.generateSubStates(label_from = 38, w = w_F5, L = 1, S = 1/2, I = I_cs, F = 5)

# Declare excited and ground states
G_cs = Fp3 + Fp4
```

(continues on next page)

(continued from previous page)

```
E_cs = F2 + F3 + F4 + F5

# Laser parameters
intensity_cs = 50 # mW/mm^-2
Q_cs = [0]

# Simulation parameters
start_time = 0
stop_time = 500 # in ns
time_steps = 501
time_cs = np.linspace(start_time, stop_time, time_steps)
```

Create a `LaserAtomSystem` object and time evolve the system using `timeEvolution()`.

[3]:
```
cs_system = las.LaserAtomSystem(E_cs, G_cs, tau_cs, Q_cs, wavelength_cs, laser_intensity␣
↪= intensity_cs)
tic = time.perf_counter()
cs_system.timeEvolution(time_cs)
toc = time.perf_counter()
print(f"The code finished in {toc-tic:0.4f} seconds")
```

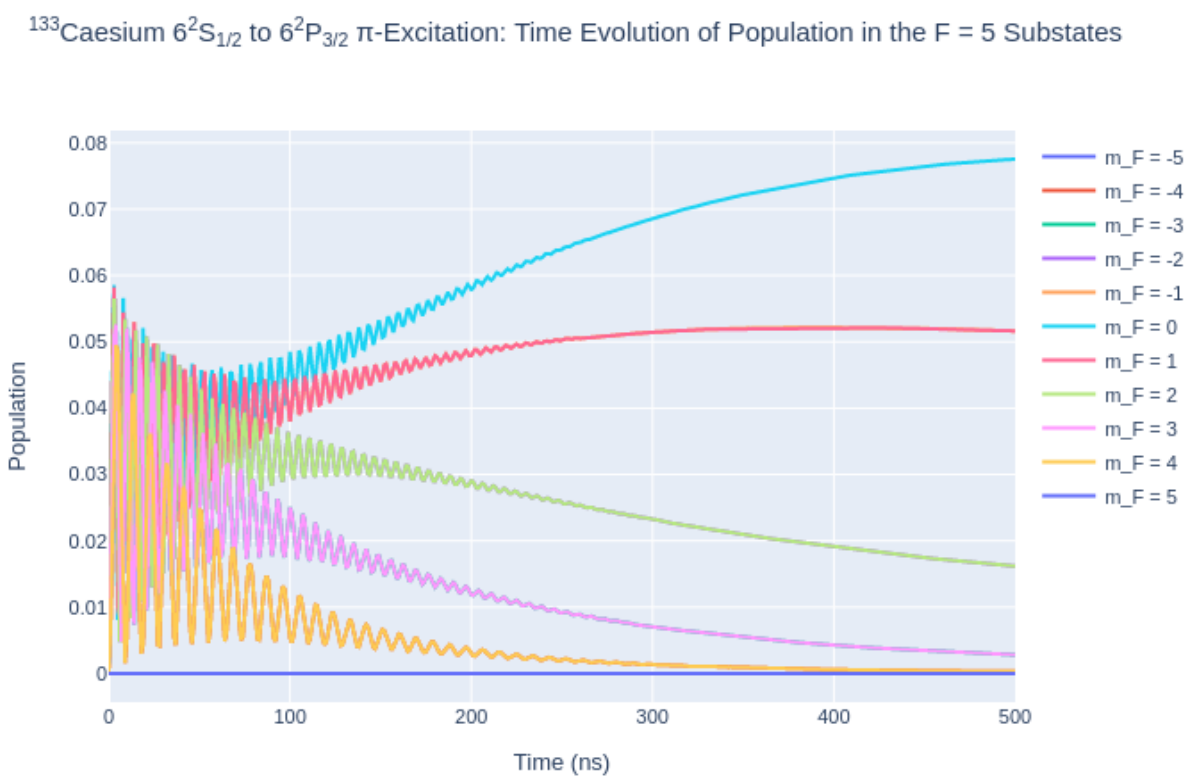
```
Populating ground states equally as the initial condition.
The code finished in 8568.3967 seconds
```

2.6.2 Saving and Plotting

[4]:
```
cs_system.saveToCSV(f"cs133piExcitationI={intensity_cs}.csv")
```

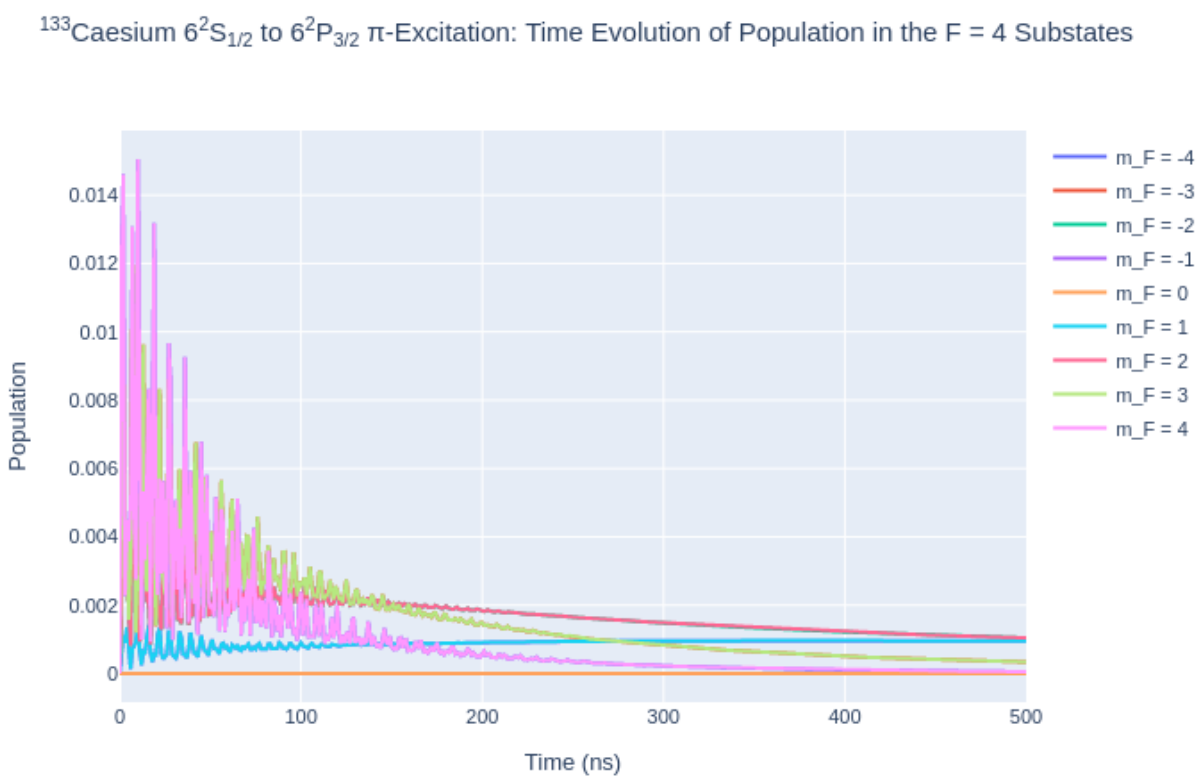
[5]:
```
rho_to_plot = [ [abs(rho) for rho in cs_system.Rho_t(s, s)] for s in F5]

fig_csF5 = go.Figure()

for i, rho in enumerate(rho_to_plot):
    fig_csF5.add_trace(go.Scatter(x = time_cs,
                                  y = rho,
                                  name = f"m_F = {F5[i].m}",
                                 mode = 'lines'))

fig_csF5.update_layout(title = "<sup>133</sup>Caesium 6<sup>2</sup>S<sub>1/2</sub> to 6
↪<sup>2</sup>P<sub>3/2</sub> -Excitation: Time Evolution of Population in the F = 5␣
↪Substates",
                   xaxis_title = "Time (ns)",
                   yaxis_title = "Population",
                  font = dict(
                      size = 11))
fig_csF5.write_image(f"SavedPlots/CsF=5I={intensity_cs}.png")
Image(f"SavedPlots/CsF=5I={intensity_cs}.png")
```

[5]:



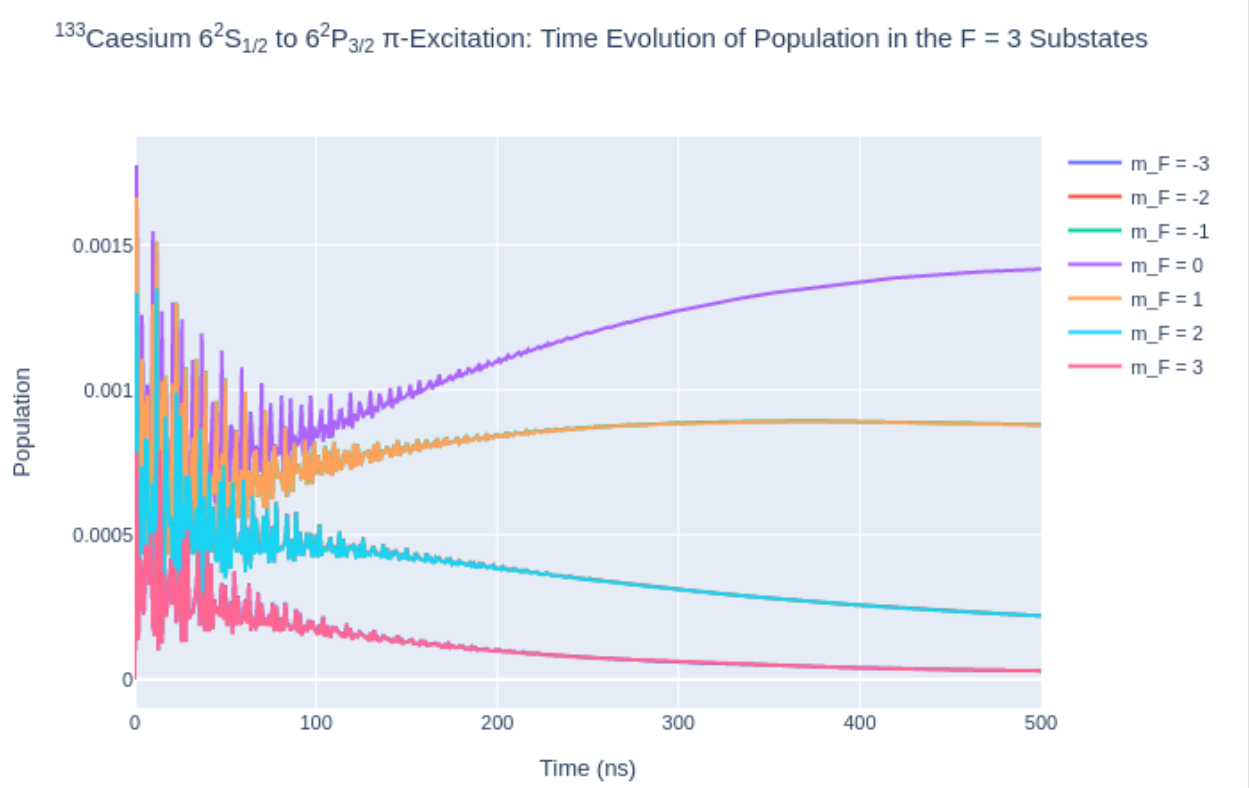
[6]:
```
rho_to_plot = [ [abs(rho) for rho in cs_system.Rho_t(s, s)] for s in F4]

fig_csF4 = go.Figure()

for i, rho in enumerate(rho_to_plot):
    fig_csF4.add_trace(go.Scatter(x = time_cs,
                                  y = rho,
                                  name = f"m_F = {F4[i].m}",
                                 mode = 'lines'))

fig_csF4.update_layout(title = "<sup>133</sup>Caesium 6<sup>2</sup>S<sub>1/2</sub> to 6
↪<sup>2</sup>P<sub>3/2</sub> -Excitation: Time Evolution of Population in the F = 4␣
↪Substates",
                   xaxis_title = "Time (ns)",
                   yaxis_title = "Population",
                  font = dict(
                      size = 11))
fig_csF4.write_image(f"SavedPlots/CsF=4I={intensity_cs}.png")
Image(f"SavedPlots/CsF=4I={intensity_cs}.png")
```

[6]:



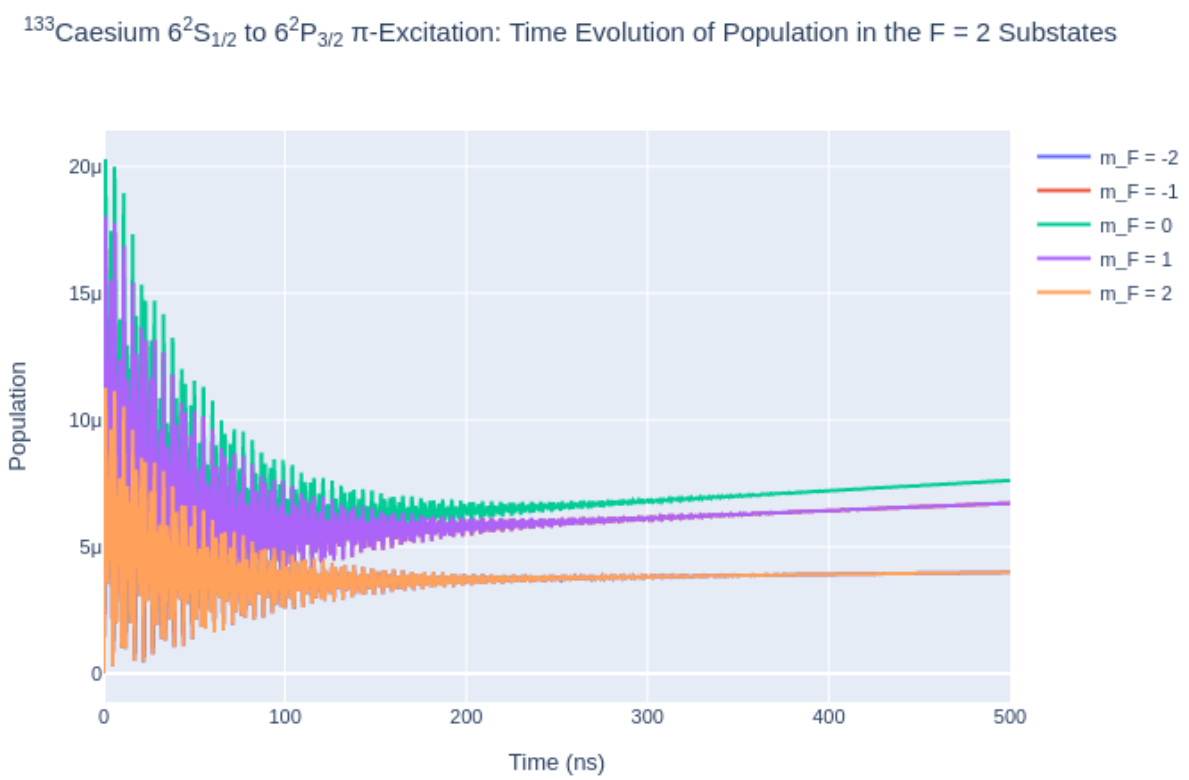
[7]:
```
rho_to_plot = [ [abs(rho) for rho in cs_system.Rho_t(s, s)] for s in F3]

fig_csF3 = go.Figure()

for i, rho in enumerate(rho_to_plot):
    fig_csF3.add_trace(go.Scatter(x = time_cs,
                                  y = rho,
                                  name = f"m_F = {F3[i].m}",
                                 mode = 'lines'))

fig_csF3.update_layout(title = "<sup>133</sup>Caesium 6<sup>2</sup>S<sub>1/2</sub> to 6
↪<sup>2</sup>P<sub>3/2</sub> -Excitation: Time Evolution of Population in the F = 3␣
↪Substates",
                  xaxis_title = "Time (ns)",
                  yaxis_title = "Population",
                 font = dict(
                     size = 11))
fig_csF3.write_image(f"SavedPlots/CsF=3I={intensity_cs}.png")
Image(f"SavedPlots/CsF=3I={intensity_cs}.png")
```

[7]:



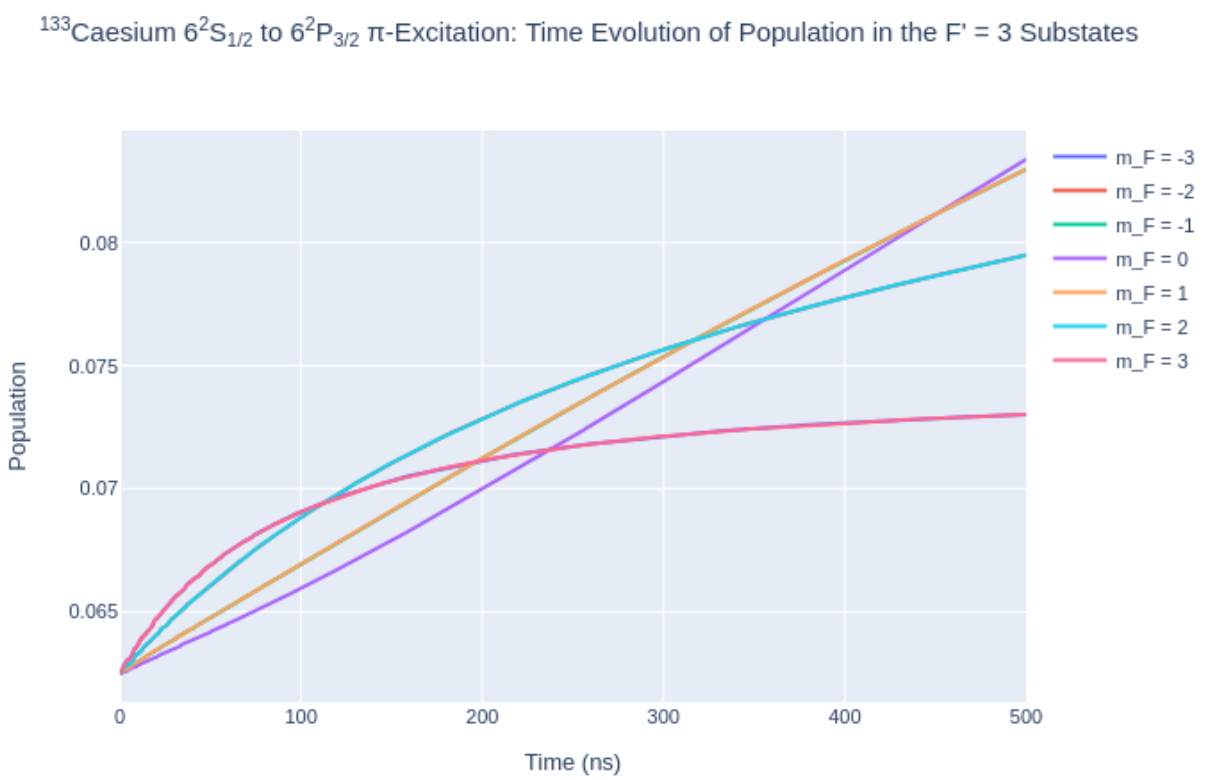
[8]:
```
rho_to_plot = [ [abs(rho) for rho in cs_system.Rho_t(s, s)] for s in F2]

fig_csF2 = go.Figure()

for i, rho in enumerate(rho_to_plot):
    fig_csF2.add_trace(go.Scatter(x = time_cs,
                                  y = rho,
                                  name = f"m_F = {F2[i].m}",
                                 mode = 'lines'))

fig_csF2.update_layout(title = "<sup>133</sup>Caesium 6<sup>2</sup>S<sub>1/2</sub> to 6
↪<sup>2</sup>P<sub>3/2</sub> -Excitation: Time Evolution of Population in the F = 2␣
↪Substates",
                   xaxis_title = "Time (ns)",
                   yaxis_title = "Population",
                  font = dict(
                      size = 11))
fig_csF2.write_image(f"SavedPlots/CsF=2I={intensity_cs}.png")
Image(f"SavedPlots/CsF=2I={intensity_cs}.png")
```

[8]:



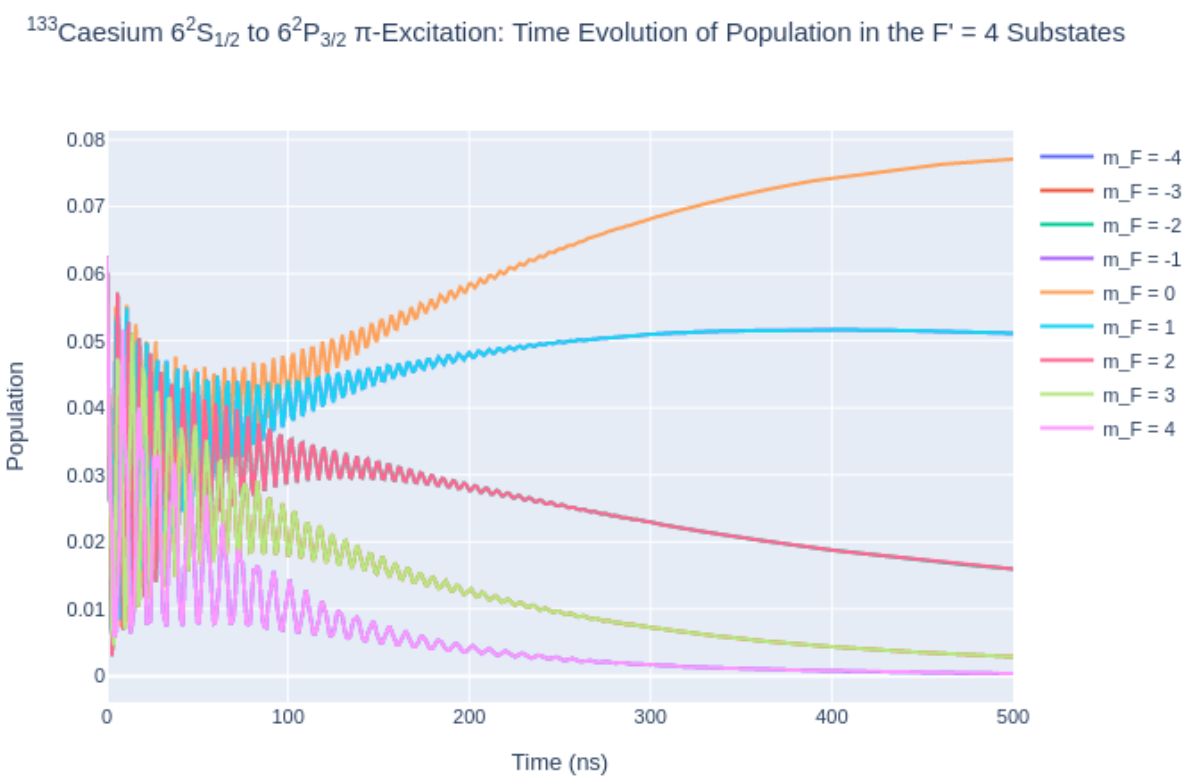
[9]:
```
rho_to_plot = [ [abs(rho) for rho in cs_system.Rho_t(s, s)] for s in Fp3]

fig_csFp3 = go.Figure()

for i, rho in enumerate(rho_to_plot):
    fig_csFp3.add_trace(go.Scatter(x = time_cs,
                                   y = rho,
                                   name = f"m_F = {Fp3[i].m}",
                                  mode = 'lines'))

fig_csFp3.update_layout(title = "<sup>133</sup>Caesium 6<sup>2</sup>S<sub>1/2</sub> to 6
↪<sup>2</sup>P<sub>3/2</sub> -Excitation: Time Evolution of Population in the F' = 3␣
↪Substates",
                  xaxis_title = "Time (ns)",
                  yaxis_title = "Population",
                 font = dict(
                     size = 11))
fig_csFp3.write_image(f"SavedPlots/CsFp=3I={intensity_cs}.png")
Image(f"SavedPlots/CsFp=3I={intensity_cs}.png")
```

[9]:



[10]:
```
rho_to_plot = [ [abs(rho) for rho in cs_system.Rho_t(s, s)] for s in Fp4]

fig_csFp4 = go.Figure()

for i, rho in enumerate(rho_to_plot):
    fig_csFp4.add_trace(go.Scatter(x = time_cs,
                                   y = rho,
                                   name = f"m_F = {Fp4[i].m}",
                                  mode = 'lines'))

fig_csFp4.update_layout(title = "<sup>133</sup>Caesium 6<sup>2</sup>S<sub>1/2</sub> to 6
↪<sup>2</sup>P<sub>3/2</sub> -Excitation: Time Evolution of Population in the F' = 4␣
↪Substates",
                 xaxis_title = "Time (ns)",
                 yaxis_title = "Population",
                font = dict(
                    size = 11))
fig_csFp4.write_image(f"SavedPlots/CsFp=4I={intensity_cs}.png")
Image(f"SavedPlots/CsFp=4I={intensity_cs}.png")
```

[10]:


^{133}Caesium $6^2S_{1/2}$ to $6^2P_{3/2}$ π-Excitation: Time Evolution of Population in the F' = 4 Substates
Population
Time (ns)
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0
100
200
300
400
500
m_F = -4
m_F = -3
m_F = -2
m_F = -1
m_F = 0
m_F = 1
m_F = 2
m_F = 3
m_F = 4

CHAPTER

THREE

DETAILED API

3.1 Detailed API

3.1.1 Submodules

3.1.2 LASED.constants module

Contains definitions of useful fundamental constants. Author: Manish Patel Last Edited: 25/08/2021

3.1.3 LASED.decay_constant module

This is a file to define functions to calculate the generalised decay constants Author: Manish Patel Date created: 27/07/2021

LASED.decay_constant.**generalisedDecayConstant**(*ep*, *epp*, *g*, *G*, *Q_decay*)

Calculates the branching ratio for the generalised decay constant.

This ratio must be multiplied by 1/tau to get the generalised decay constant in Grad/s. This is then used for evaluating vertical coherences.

Parameters

- **ep** (State) – Excited State object
- **epp** (State) – Excited State object
- **g** (State) – Ground State object
- **G** (*list*) – :List of all ground State objects
- **tau** (*float*) – Lifetime of state in ns
- **Q_decay** (*list*) – List of decay channel polarisations allowed by transition rules, usually [-1, 0, 1]

Returns Value of the branching ratio for the generalised decay constant gamma_{ep, epp, g}.

3.1.4 LASED.density_matrix module

Density matrix operations Author: Manish Patel Date Created: 11/06/2021

LASED.density_matrix.**JNumber**(*state_list*)
Calculate the angular momentum from the number of sub-states in the list.

Parameters **state_list** (*list*) – List of sub-states of a single angular momentum state.

Returns int: The total angular momentum of the state.

LASED.density_matrix.**appendDensityMatrixToFlatCoupledMatrix**(*flatrho*, *density_rho*, *sub_states*, *n*)
Adds density matrix elements to a flat, coupled matrix.

Parameters

- **flat_rho** (*list*) – array of arrays with one column of all density matrix elements of coupled E and G states
- **n** (*int*) – number of states in total laser-coupled system
- **sub-states** (*list*) – a list of the excited or ground states
- **density_rho** (*ndarray*) – either the excited or ground state density matrix with the convention that the upper left-hand of the matrix is state population for m_J = -J if the state has angular momentum J

LASED.density_matrix.**getSingleStateMatrix**(*flat_rho*, *n*, *sub_states*)
Obtain an angular momentum state density matrix from the flattened coupled state density rho vector.

Parameters

- **flat_rho** (*list of list*) – flattened 2D density matrix of coupled E & G states with the laser.
- **n** (*int*) – number of states in total laser-coupled system.
- **sub_states** (*list*) – a list of the excited or ground states, E or G respectively.

Returns A square matrix of size length of sub_states. Elements are ordered from left to right according to the order of sub_states e.g. if state labelled 1 is first then first element would correspond to rho_11 i.e. population of state 1.

Return type ndarray

3.1.5 LASED.detuning module

Define functions for the detuning of an atomic system

LASED.detuning.**angularFreq**(*wavelength*)
Calculates the angular frequency in Grad/s from a given wavelength.

Parameters **wavelength** (*float*) – A wavelength in nm

Returns The angular frequency in Grad/s

Return type float

LASED.detuning.**delta**(*e*, *g*)
Detunings between substates.

Parameters: e (State): State object g (State): State object

Returns Difference in angular frequency of states (Grad/s).

Return type float

LASED.detuning.**dopplerDelta**(*e*, *g*, *w_q*, *lambda_q*, *v_z*)
The detuning between excited and ground states.

Accounts for a fixed motion of the atoms. Used between excited and ground states.

Parameters

- **e** (State) – State object for excited state.
- **g** (State) – State object for ground state.
- **w_q** (*float*) – Angular frequency of exciting laser in rad/s.
- **lambda_q** (*float*) – Wavelength of exciting laser in m.
- **v_z** (*float*) – Velocity component of atoms in direction of laser in m/s.

Returns The detuning between ground and excited states including the doppler detuning due to a given atomic velocity.

Return type float

3.1.6 LASED.generate_sub_states module

Generate Sub States. Generates sub-states of a quantum state

LASED.generate_sub_states.**generateSubStates**(*label_from*, *w*, *L*, *S*, *J=None*, *I=None*, *F=None*)
Generates a vector of sub-states given quantum numbers.

Parameters

- **label_from** (*int*) – the number assigned to the first state generated - goes from -m_F to +m_F so the number given will be assigned to -m_F
- **w** (*float*) – the angular frequency (in Grad/s) given to the states generated
- **L** (*int*) – orbital angular momentum
- **S** (*int*) – spin quantum number
- **J** (*int*) – total angular momentum
- **I** (*int*) – nuclear isospin
- **F** (*int*) – total angular momentum + isospin

Returns A list of State objects from -m_F to +m_F with labels beginning at label_from

Return type list

3.1.7 LASED.half_rabi_freq module

Definition of the half-Rabi frequency and all other functions needed to calculate it

LASED.half_rabi_freq.**coupling**(*e*, *g*, *q*)
Coupling constants between states via laser radiation

Parameters

- **e** (State) – the excited state
- **g** (State) – the ground state
- **q** (*int*) – the laser polarisation coupling e and g. This can be -1 for LH circular, +1 for RH circular, or 0 for linear polarisation.
- **Returns** – float: the coupling constant between an excited and ground state

LASED.half_rabi_freq.**gaussianIntensity**(*P_las*, *r_sigma*, *r*)
Intensity of Gaussian TEM_00 laser beam profile.

Parameters

- **P_las** (*float*) – Power of laser given by power meter in mW
- **r_sigma** (*float*) – Radius at 2D standard deviation of a Gaussian in mm
- **r** (*float*) – Radius in mm at which the laser intensity is evaluated

Returns Intensity in mW/mm^2 at radius r from beam axis given a Gaussian intensity profile

Return type float

LASED.half_rabi_freq.**halfRabiFreq**(*intensity*, *lifetime*, *wavelength*)
Calculates the half-Rabi frequency in Grad/s.

Parameters

- **intensity** – intensity of laser in mW/mm^2
- **lifetime** – lifetime of the excited state to the ground state transition in nanoseconds
- **wavelength** – wavelength (in metres) corresponding to the resonant transition from the ground to excited state.

Returns The half-Rabi frequency in Grad/s.

Return type float

3.1.8 LASED.index module

Definition of the index function

LASED.index.**getStateLabelsFromLineNo**(*line_no*, *n*)
Gets the indeces for labelling rho [i, j] from the line number of an array.

Parameters

- **line_no** (*int*) – line number of an array e.g. position of a row in a matrix or column in rho_t
- **n** – the number of states in the system

Returns A tuple of the indeces for the matrix position

Return type tuple

LASED.index.**index**(*i*, *j*, *n*)

Function to return the index of an element of a 2D density matrix if flattened to a 1D array

Parameters

- **i** (State) – To get the row number of the element to be selected
- **j** (State) – To get the column number of the element to be selected
- **n** (*int*) – dimension of the square array i.e. number of states in the system

Returns Index of the element in a 1D array

Return type int

Example

To get the index of rho_44 in a 4-level system call index(four, four, 4) where "four" is the variable which stores the State object labelled 4.

3.1.9 LASED.laser_atom_system module

The LaserAtomSystem class definition.

class LASED.laser_atom_system.**LaserAtomSystem**(*E*, *G*, *tau*, *Q*, *laser_wavelength*, *laser_intensity=None*, *laser_power=None*, *tau_f=None*, *tau_b=None*, *rabi_scaling=None*, *rabi_factors=None*)

Bases: object

A physical system composed of a laser field acting on an atomic system.

Q_decay

List of ints describing the selection rules of the decay. Selection rules are set to +1, -1, and 0.

Type list

rho_t

List of flattened 2D arrays over the time interval simulated for. This is initialised as empty as no time evolution has taken place.

Type list

E

List of State objects which are the excited states of the system.

Type list

G

List of State objects which are the ground states of the system.

Type list

tau

Lifetime of transition in nanoseconds between excited and ground state.

Type float

Q

List of laser polarisations. This can be +1, 0, -1 for right-hand circular, left-hand circular, and linear polarisation. If more than one polarisation is in the list then the system will be excited with a linear combination of the polarisations.

Type list

laser_wavelength
Wavelength of transition from ground to excited state in metres.

Type float

laser_intensity
Intensity of the laser in mW/mm^2.

Type float

laser_power
Power of the laser in mW. This is needed for Gaussian averaging of beam profile.

Type float

tau_f
Upper state lifetime to states outside of laser coupling in nanoseconds.

Type float

tau_b
Ground state lifetime to states outside of laser coupling in nanoseconds.

Type float

rho_0
2D array creating the density matrix at t = 0.

Type list of list

rabi_scaling
The normalisation of the Rabi frequency. The half-Rabi frequency is divided by this number. Use this if there are more than one polarisations to normalise the population.

Type float

rabi_factors
Elements of this list are multiplies by the half-Rabi frequency for each polarisation. This can be used to obtain elliptical polarisation.

Type list

Q_decay = [1, 0, -1]

Rho_0(*i*, *j*)
Accessor for an element in rho_0.

Parameters

- **i** (State) – First state index.
- **j** (State) – Second state index.

Returns element of the laser-atom system density matrix at t=0.

Return type complex

Example

print(Rho_0(one, two))

Rho_t(*i*, *j*)
Accessor for an element in rho_t.

Parameters

- **i** (State) – First state index
- **j** (State) – Second state index

Returns Array of an element in laser-atom system for all of the simulation time

Return type list

Example

print(Rho_t(one, two)) prints element rho_12 if one and two are State objects corresponding to label 1 and 2 respectively.

angularShape_0(*state*, *theta*, *phi*)
Gets the angular shape (radius) of the atomic state given at t = 0.

Parameters

- **state** (*char*) – Either "e" or "g" specifying the excited or lower state.
- **theta** (*array_like*) – Azimuthal (longitudinal) coordinate in [0, 2*pi].
- **phi** (*array_like*) – Polar (colatitudinal) coordinate in [0, pi].

Returns A list of lists with the radius of the angular shape of the atomic state given at t = 0.

Return type (2D array)

Example

calcium.angularShape("g", theta, phi) for a pre-defined calcium system gives the angular shape of the lower state density matrix.

angularShape_t(*state*, *theta*, *phi*)
Gets the time evolution of the angular shape (radius) of the atomic state given.

Parameters

- **state** (*char*) – Either "e" or "g" specifying the excited or lower state.
- **theta** (*array_like*) – Azimuthal (longitudinal) coordinate in [0, 2*pi].
- **phi** (*array_like*) – Polar (colatitudinal) coordinate in [0, pi].

Returns List of 2D arrays of the radius of the angular shape of the atomic state for a given time in the evolution of the laser-atom system.

Return type (List of 2D array)

appendDensityMatrixToRho_0(*density_rho*, *state_type*)
Sets the laser-atom system density matrix at t=0 to the matrix given.

Parameters

- **density_rho** (*ndarray*) – 2D array of the system density matrix.

- **state_type** (*char*) – Defines whether the density matrix is for the ground or excited state. Either an "e" or "g" for excited and ground state density matrices respectively.

Note: Density matrix input must be square and the size of the matrix must match with E or G.

clearRho_0()
Makes all values of rho_0 zero.

property n
Total number of substates.

Returns Number of substates.

Return type int

property rho_e0
Upper state density matrix for the initial condition.

Returns Upper state density matrix composed of all states defined in E.

Return type ndarray

property rho_et
Upper state density matrix for all of the time evolution.

Returns A list of upper state density matrices for the time evolution.

Return type list of ndarray

property rho_g0
Lower state density matrix for the initial condition.

Returns Lower state density matrix composed of all states defined in G.

Return type ndarray

property rho_gt
Lower state density matrix for all of the time evolution.

Returns A list oflower state density matrices for the time evolution.

Return type list of ndarray

rho_t = []

rotateRho_0(*alpha*, *beta*, *gamma*)
Rotate rho_0 by the Euler angles alpha, beta, and gamma.

Parameters

- **alpha** (*float*) – rotation (in radians) around z-axis
- **beta** (*float*) – rotation (in radians) about the y'-axis
- **gamma** (*float*) – rotation (in radians) about the z"-axis

rotateRho_t(*alpha*, *beta*, *gamma*)
Rotate rho_0 by the Euler angles alpha, beta, and gamma.

Parameters

- **alpha** (*float*) – rotation (in radians) around z-axis
- **beta** (*float*) – rotation (in radians) about the y'-axis
- **gamma** (*float*) – rotation (in radians) about the z"-axis

saveToCSV(*filename*, *precision=None*)
Saves rho_t as a csv file.

Parameters

- **filename** (*string*) – Name of the csv file created.
- **precision** (*int*) – Precision of numbers in decimal places.

Returns Void.

setRho_0(*i*, *j*, *value*)
Sets a value to an element of rho_0.

Parameters

- **i** (State) – First state index
- **j** (State) – Second state index
- **value** (*np.complex*) – Sets the value of rho_ij to the complex value here

time = []

timeEvolution(*time*, *beam_profile_averaging=None*, *doppler_averaging=None*, *print_eq=None*, *detuning=None*, *atomic_velocity=None*, *r_sigma=None*, *n_beam_averaging=None*, *doppler_width=None*, *doppler_detunings=None*, *pretty_print_eq=None*, *pretty_print_eq_tex=None*, *pretty_print_eq_pdf=None*, *pretty_print_eq_filename=None*)
Evolves the laser-atom system over time.

Produces a list of flattened 2D density matrices which correspond to the time array. This is stored in rho_t.

Parameters

- **time** (*list*) – Array of the times in the time evolution in nanoseconds e.g. time = [0, 1, 2] to simulate up to 2 ns
- **beam_profile_averaging** (*bool*) – Turn on averaging over a Gaussian TEM00 laser beam profile. Must have n_beam_averaging to state the number of averages to take over the beam profile and r_sigma to characterise the standard debviation of the Gaussian beam.
- **doppler_averaging** (*bool*) – Turn on averaging over the doppler profile of the atoms. Must have doppler_width and doppler_detunings as well.
- **print_eq** (*bool*) – Turn on printing the coupled differential equatiuons numerically.
- **atomic_velocity** (*float*) – The velocity component of the atoms in the direction of the laser beam in metres/second. This is used for doppler shifting the energy levels.
- **r_sigma** (*float*) – The radial distance to the 2D standard deviation in millimetres of the Gaussian beam profile of the laser.
- **n_beam_averaging** (*int*) – The number of times the beam profile will be split to average over. The higher the number, the more accurate the beam profile averaging but it will be slower.
- **doppler_width** (*float*) – The doppler width of the beam profile in Grad/s.
- **doppler_detunings** (*list*) – List of the doppler detunings creating a doppler profile. This list will be averaged over. Must go from a -ve detuning to a +ve detuning. All detunings are in Grad/s.
- **pretty_print_eq** (*bool*) – Turn on printing of the coupled differential equations symbolically using Sympy. Only available using an IPython environment e.g. Jupyter.

- **pretty_print_eq_tex** (*bool*) – Produces a .tex file with the equations of motion printed symbolically using Sympy. Must input a filename for the .tex file with the keyword "pretty_print_eq_filename".
- **pretty_print_eq_pdf** (*bool*) – Produces a .pdf file with the equations of motion printed symbolically using Sympy and pdflatex. Must have pdflatex installed on your system. Must input a filename for the .tex file with the keyword "pretty_print_eq_filename".
- **Note** – Must have laser_power attribute for Gaussian averaging of the beam and must have laser_intensity attribute when not Gaussian averaging. Also, cannot print equations in doppler or Gaussian averaging.

3.1.10 LASED.matrix_methods module

Defines functions to manipulate matrices.

LASED.matrix_methods.**printNonZeroMatrixElements**(*A*)
Prints non-zero matrix elements

Parameters **A** (*ndarray*) – A matrix

3.1.11 LASED.rotation module

Define functions for rotating density matrices in the QED simulation of a laser-atom system. Author: Manish Patel Date created: 12/05/2021

LASED.rotation.**createDictionaryOfSubStates**(*E*, *G*)
Creates a dictionary of sub-states with (F, m) as the key and the State object as the value.

Parameters

- **E** (*list of States*) – List of excited sub-states in the laser-atom system.
- **G** (*list of States*) – List of ground sub-states in the laser-atom system

Returns A dictionary with (F, m) as the keys and the corresponding State object as the value

Return type dictionary

LASED.rotation.**rotateElement**(*rho*, *i*, *j*, *n*, *sub_state_dict*, *alpha*, *beta*, *gamma*)
Rotates an element of a density matrix rho_ij by the euler angles given.

Parameters

- **rho** (*complex*) – The density matrix to be rotated.
- **i** (*state*) – A sub-state of the laser-atom system.
- **j** (*state*) – A sub-state of the laser-atom system.
- **sub_state_dict** (*dict of State*) – Dictionary of States with (F,m) tuple as the keys and the corresponding State object as the value.
- **n** (*int*) – Total number of sub-states in the system.
- **alpha** (*float*) – rotation around z-axis in radians.
- **beta** (*float*) – rotation about the y'-axis in radians.
- **gamma** (*float*) – rotation about the z"-axis in radians.

Returns The rotated density matrix element.

Return type complex

LASED.rotation.**rotateFlatDensityMatrix**(*flat_rho*, *n*, *E*, *G*, *alpha*, *beta*, *gamma*)
Rotate the excited and ground state populations by the Euler angles.

Parameters

- **flat_rho** (*list*) – A flattened 2D density matrix
- **n** (*int*) – Number of substates which compose the density matrix flat_rho
- **E** (*list of States*) – list of excited State objects which compose flat_rho
- **G** (*list of States*) – list of ground State objects which compose flat_rho
- **alpha** (*float*) – rotation around z-axis in radians.
- **beta** (*float*) – rotation about the y'-axis in radians.
- **gamma** (*float*) – rotation about the z"-axis in radians.

Returns A rotated flattened 2D density matrix

Return type list of lists

LASED.rotation.**small_Wigner_D**(*J*, *beta*, *mp*, *m*)
Calculates the small Wigner D-matrix elements for rotation.

Parameters

- **J** (*int*) – total angular momentum quantum number of the state which will be rotated with the resulting D-matrix.
- **beta** (*float*) – rotation about the y'-axis in radians.
- **mp** (*int*) – row number of element in the Wigner D-matrix
- **m** (*int*) – column number oif element in the Wigner D-matrix

Returns The small Wigner D-matrix elements

Return type float

LASED.rotation.**wigner_D**(*J*, *mu*, *m*, *alpha*, *beta*, *gamma*)
Calculates the Wigner D-matrix for rotation by Eueler angles (alpha, beta, gamma).

Parameters

- **J** (*int*) – total angular momentum quantum number of the state which will be rotated with the
- **D-matrix.** (*resulting*) –
- **alpha** (*float*) – rotation around z-axis in radians.
- **beta** (*float*) – rotation about the y'-axis in radians.
- **gamma** (*float*) – rotation about the z"-axis in radians.

Returns A square matrix of size 2J+1.

Return type ndarray

3.1.12 LASED.save_to_csv module

This is a file to define the function to save rho_t as a csv file.

LASED.save_to_csv.**saveRhotAsCSV**(*n*, *E*, *G*, *time*, *rho_t*, *filename*, *precision=None*)
Saves rho_t to a csv file.

Creates a csv file with filename and saves each element's time evolution as a column.

Parameters

- **n** (*int*) – Number of substates in the system.
- **E** (*list of States*) – List of State objects in excited state of system.
- **G** (*list of States*) – List of State objects in ground state of system.
- **rho_t** (*list of lists*) – List of flattened 2D density matrices representing time evolution of density matrix.
- **filename** (*string*) – Name of the csv file created.
- **precision** (*int*) – Precision of numbers in decimal places.

Returns Void

LASED.save_to_csv.**stateLabel**(*s*, *state_type*)
Returns the short-hand label of the state.

Parameters

- **s** (State) – a State object.
- **state_type** (*string*) – either 'e' or 'g' for excited state or ground state. The ground state is primed.

Returns "J=k;m_J=l" if k and l are J and m quantum numbers. If the state has isospin then it is "F=k;m_F=l"

Return type string

LASED.save_to_csv.**writeCSV**(*filename*, *headers*, *data*, *precision=None*)
Creates a csv file using pandas dataframe.

Parameters

- **headers** – Array of strings used as column headers.
- **data** – Array of arrays of data used for the columns, must be same size as headers
- **filename** – String of the name of the csv file output
- **precision** (*int*) – Precision of numbers in decimal places.

Returns Void.

3.1.13 LASED.state module

Class definition for an atomic state

class LASED.state.**State**(*label*, *w*, *m*, *L*, *S*, *J=None*, *I=None*, *F=None*)

Bases: object

An atomic state.

label

Number labelling of state e.g. state |2> would have label 2.

Type int

w

Angular frequency corresponding to energy of state in Grad/s.

Type float

L

Orbital angular momentum quantum number.

Type int

S

Spin quantum number.

Type int

m

Degeneracy of the total angular momentum.

Type int

J

Resultant of L coupling to S.

Type int

I

Nuclear spin quantum number.

Type int

F

Total angular momentum.

Type int

3.1.14 LASED.symbolic_print module

LASED.symbolic_print.**appendEqToTexFile**(*file*, *input_str*)

Appends an equation to a .tex file given.

Parameters

- **file** (*file*) – file object to write to
- **input_str** (*string*) – The string (equation) to be written to the .tex file

LASED.symbolic_print.**callPdfLatex**(*filename*)

Calls pdflatex on the system to convert the .tex file to a .pdf file. Must have pdflatex installed.

Parameters **filename** (*string*) – Name of the .tex file to be converted.

LASED.symbolic_print.**closeTexFile**(*file*)
Closes a .tex file which has been written to

Parameters **file** (*file*) – file object to be closed

LASED.symbolic_print.**createTexFile**(*filename*)
Creates a .tex file with the filename given and returns a file

Parameters **filename** (*string*) – Name of the .tex file created.

Returns .tex file with filename

Return type file

LASED.symbolic_print.**symbolicPrintRhoee**(*n*, *E*, *G*, *Q*, *Q_decay*, *tau_f*, *rabi_scaling*, *rabi_factors*, *pretty_print_eq=None*, *pretty_print_eq_file=None*)
Prints the density matrix elements rho_ee" for the motion of the laser-atom system using Sympy.

LASED.symbolic_print.**symbolicPrintRhoeg**(*n*, *E*, *G*, *Q*, *Q_decay*, *tau_f*, *tau_b*, *detuning*, *laser_wavelength*, *atomic_velocity*, *rabi_scaling*, *rabi_factors*, *pretty_print_eq=None*, *pretty_print_eq_file=None*)
Prints the density matrix elements rho_eg for the motion of the laser-atom system using Sympy.

LASED.symbolic_print.**symbolicPrintRhoge**(*n*, *E*, *G*, *Q*, *Q_decay*, *tau_f*, *tau_b*, *detuning*, *laser_wavelength*, *atomic_velocity*, *rabi_scaling*, *rabi_factors*, *pretty_print_eq=None*, *pretty_print_eq_file=None*)
Prints the density matrix elements rho_ge for the motion of the laser-atom system using Sympy.

LASED.symbolic_print.**symbolicPrintRhogg**(*n*, *E*, *G*, *Q*, *Q_decay*, *tau_b*, *rabi_scaling*, *rabi_factors*, *pretty_print_eq=None*, *pretty_print_eq_file=None*)
Prints the density matrix elements rho_gg" for the motion of the laser-atom system using Sympy.

LASED.symbolic_print.**symbolicPrintSystem**(*n*, *E*, *G*, *Q*, *Q_decay*, *tau_f*, *tau_b*, *detuning*, *laser_wavelength*, *atomic_velocity*, *rabi_scaling*, *rabi_factors*, *pretty_print_eq=None*, *pretty_print_eq_tex=None*, *pretty_print_eq_pdf=None*, *pretty_print_eq_filename=None*)
Prints the equations of motion of the laser-atom system in full using Sympy.

3.1.15 LASED.time_evolution module

This file contains the function to calculate the time evolution of the density matrix for an atomic system interacting with a laser.

LASED.time_evolution.**timeEvolution**(*n*, *E*, *G*, *Q*, *Q_decay*, *tau*, *laser_intensity*, *laser_wavelength*, *time*, *rho0*, *rho_output*, *tau_f=None*, *tau_b=None*, *detuning=None*, *rabi_scaling=None*, *rabi_factors=None*, *print_eq=None*, *atomic_velocity=None*, *pretty_print_eq=None*, *pretty_print_eq_tex=None*, *pretty_print_eq_pdf=None*, *pretty_print_eq_filename=None*)
Calculates the time evolution of a laser-atom system.

Uses a flattened density matrix rho0 and calculates the time evolution over the time specified. The density matrix at each time step is stored in rho_output.

LASED.time_evolution.**timeEvolutionDopplerAveraging**(*n*, *E*, *G*, *Q*, *Q_decay*, *tau*, *laser_intensity*, *laser_wavelength*, *doppler_width*, *doppler_detunings*, *time*, *rho0*, *rho_output*, *tau_f=None*, *tau_b=None*, *detuning=None*, *rabi_scaling=None*, *rabi_factors=None*, *print_eq=None*, *atomic_velocity=None*, *pretty_print_eq=None*, *pretty_print_eq_tex=None*, *pretty_print_eq_pdf=None*, *pretty_print_eq_filename=None*)

Calculates the time evolution of a laser-atom system with a Gaussian doppler profile for the atoms.

Uses a flattened density matrix rho0 and calculates the time evolution over the time specified. The density matrix at each time step is stored in rho_output.

LASED.time_evolution.**timeEvolutionGaussianAndDopplerAveraging**(*n*, *E*, *G*, *Q*, *Q_decay*, *tau*, *laser_power*, *r_sigma*, *n_intensity*, *laser_wavelength*, *doppler_width*, *doppler_detunings*, *time*, *rho0*, *rho_output*, *tau_f=None*, *tau_b=None*, *detuning=None*, *rabi_scaling=None*, *rabi_factors=None*, *print_eq=None*, *atomic_velocity=None*, *pretty_print_eq=None*, *pretty_print_eq_tex=None*, *pretty_print_eq_pdf=None*, *pretty_print_eq_filename=None*)

Calculates the time evolution of a laser-atom system with a Gaussian doppler profile for the atoms and a Gaussian laser beam profile.

Uses a flattened density matrix rho0 and calculates the time evolution over the time specified. The density matrix at each time step is stored in rho_output.

LASED.time_evolution.**timeEvolutionGaussianAveraging**(*n*, *E*, *G*, *Q*, *Q_decay*, *tau*, *laser_power*, *r_sigma*, *n_intensity*, *laser_wavelength*, *time*, *rho0*, *rho_output*, *tau_f=None*, *tau_b=None*, *detuning=None*, *rabi_scaling=None*, *rabi_factors=None*, *print_eq=None*, *atomic_velocity=None*, *pretty_print_eq=None*, *pretty_print_eq_tex=None*, *pretty_print_eq_pdf=None*, *pretty_print_eq_filename=None*)

Calculates the time evolution of a laser-atom system with a Gaussian laser beam profile.

Uses a flattened density matrix rho0 and calculates the time evolution over the time specified. The density matrix at each time step is stored in rho_output.

3.1.16 LASED.time_evolution_matrix module

This is a file to define a function to populate the time evolution matrix for a laser-atom system Author: Manish Patel Date created: 12/05/2021

LASED.time_evolution_matrix.**rho_eepp**(*A*, *n*, *E*, *G*, *Q*, *Q_decay*, *tau*, *rabi*, *rabi_factors*, *tau_f=None*, *numeric_print=None*)

Function to populate the matrix A with coefficients for populations and atomic coherences of the excited states.

LASED.time_evolution_matrix.**rho_eg**(*A*, *n*, *E*, *G*, *Q*, *Q_decay*, *tau*, *rabi*, *rabi_factors*, *laser_wavelength*, *atomic_velocity*, *tau_f=None*, *tau_b=None*, *detuning=None*, *numeric_print=None*)

Function to populate the matrix A with coefficients for optical coherences between excited and ground states.

LASED.time_evolution_matrix.**rho_ge**(*A*, *n*, *E*, *G*, *Q*, *Q_decay*, *tau*, *rabi*, *rabi_factors*, *laser_wavelength*, *atomic_velocity*, *tau_f=None*, *tau_b=None*, *detuning=None*, *numeric_print=None*)

Function to populate the matrix A with coefficients for optical coherences between ground and excited states.

LASED.time_evolution_matrix.**rho_ggpp**(*A*, *n*, *E*, *G*, *Q*, *Q_decay*, *tau*, *rabi*, *rabi_factors*, *tau_b=None*, *numeric_print=None*)

Function to populate the matrix A with coefficients for populations and atomic coherences of the ground states.

LASED.time_evolution_matrix.**timeEvolutionMatrix**(*n*, *E*, *G*, *Q*, *Q_decay*, *tau*, *laser_wavelength*, *laser_intensity*, *tau_f=None*, *tau_b=None*, *detuning=None*, *numeric_print=None*, *rabi_scaling=None*, *rabi_factors=None*, *atomic_velocity=None*, *pretty_print_eq=None*, *pretty_print_eq_tex=None*, *pretty_print_eq_pdf=None*, *pretty_print_eq_filename=None*)

Function to create and populate the coupled differential equation matrix A for the laser-atom system.

Returns Matrix which contains all thera coefficients for the set of coupled differential equations describing a laser-atom system.

Return type ndarray

3.1.17 Module contents

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

L

INDEX

M

N

P

Q

R

S

T

W