In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
from scipy.integrate import complex_ode
```

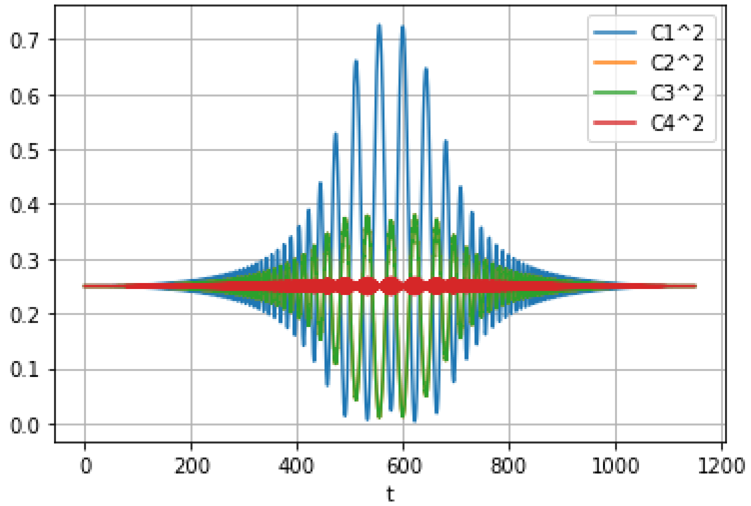
In [2]:

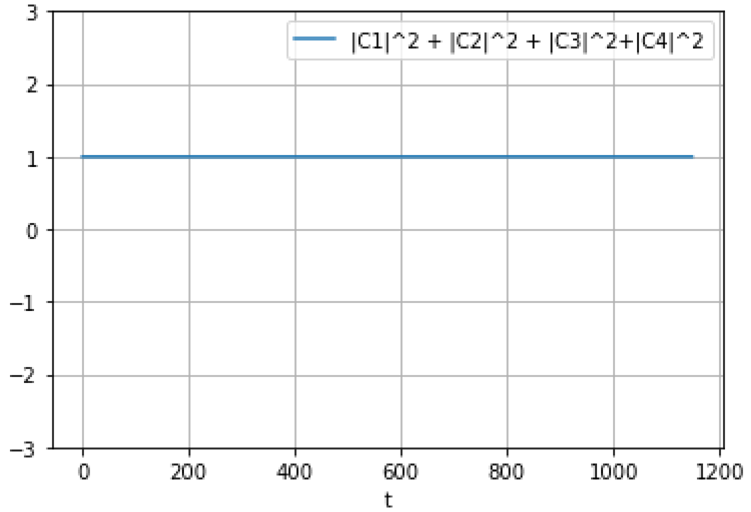
```
def odeintz(func, z0, t, **kwargs):
    """An odeint-like function for complex valued differential equations."""

    # Disallow Jacobian-related arguments.
    _unsupported_odeint_args = ['Dfun', 'col_deriv', 'ml', 'mu']
    bad_args = [arg for arg in kwargs if arg in _unsupported_odeint_args]
    if len(bad_args) > 0:
        raise ValueError("The odeint argument %r is not supported by "
                         "odeintz." % (bad_args[0],))

    # Make sure z0 is a numpy array of type np.complex128.
    z0 = np.array(z0, dtype=np.complex128, ndmin=1)

    def realfunc(x, t, *args):
        z = x.view(np.complex128)
        dzdt = func(z, t, *args)
        # func might return a python list, so convert its return
        # value to an array with type np.complex128, and then return
        # a np.float64 view of that array.
        return np.asarray(dzdt, dtype=np.complex128).view(np.float64)

    result = odeint(realfunc, z0.view(np.float64), t, **kwargs)

    if kwargs.get('full_output', False):
        z = result[0].view(np.complex128)
        infodict = result[1]
        return z, infodict
    else:
        z = result.view(np.complex128)
        return z

#This fuction is from stack overlflow as it's difficult for odeint to handle complex numb
```

In [93]:

```
if __name__ == "__main__":
    # Generate a solution to:
    #     dC_1/dt = -i*(omega)*(C_2+C_3)
    #     dC_2/dt = -i*((omega)*(C_1+C_4)+delta*(C_2))
    #     dC_3/dt = -i*((omega)*(C_1+C_4)+delta*(C_3))
    #     dC_4/dt = -i*((omega)*(C_2+C_3)+(v+2*delta*(C_4))


    # Define the right-hand-side of the differential equation.
    #ODE's needed to be solved
    def zfunc(z, t, omega_0, delta_0,v):
        C_1,C_2,C_3,C_4=z
        tau=114.85/omega_0
        #corrected
        omega=omega_0*np.sin((np.pi*t)/tau)**2 / 2
        delta=delta_0*np.cos((np.pi*t)/tau)**2


        new_c1=-1j*(omega)*(C_2+C_3)
        new_c2=-1j*((omega)*(C_1+C_4)+delta*(C_2))
        new_c3=-1j*((omega)*(C_1+C_4)+delta*(C_3))
        new_c4= -1j*((omega)*(C_2+C_3)+((v+2*delta)*(C_4)))

        return [new_c1,new_c2,new_c3,new_c4]

    # Set up the inputs and call odeintz to solve the system.

    #coefficient of the system at the start.
    intial = np.array([0.5,0.5,0.5,0.5])
    #normalisation.
    intial_1=np.abs(intial)**2
    normalisation=np.sum(intial_1)
    #normalised coeffcient.
    intial=(intial)/(np.sqrt(normalisation))

    #values of the parameters.
    omega_0 = 0.1
    delta_0 = 1.8
    tau=114.85/omega_0
    #corrected
    v=4.0
    t = np.linspace(0, tau, 10000)
    #solution
    z, infodict = odeintz(zfunc, intial, t, args=(omega_0,delta_0,v),full_output=True)

    #Each coeffcient ode solved
    C_1= z[:,0]
    C_2= z[:,1]
    C_3= z[:,2]
    C_4= z[:,3]

    #check if normalised
    norm=(np.abs(C_1))**2 +(np.abs(C_2))**2 +(np.abs(C_3))**2 +(np.abs(C_4))**2

    #modulus squared of each coefficient
    c1_squared= np.abs(C_1)**2
    c2_squared= np.abs(C_2)**2
    c3_squared= np.abs(C_3)**2
    c4_squared= np.abs(C_4)**2
```

```
import matplotlib.pyplot as plt
#plot coefficient modulus squared against time
plt.clf()
plt.plot(t, c1_squared, label='C1^2')
plt.plot(t, c2_squared, label='C2^2')
plt.plot(t, c3_squared, label='C3^2')
plt.plot(t, c4_squared, label='C4^2')
plt.xlabel('t')
plt.grid(True)
plt.legend(loc='best')
plt.show()



plt.clf()
plt.plot(t, norm, label='|C1|^2 + |C2|^2 + |C3|^2+|C4|^2')
plt.xlabel('t')
plt.grid(True)
plt.legend(loc='best')
plt.ylim(-3,3)
plt.show()
```





In [101]:

```
#The coefficent of the state after the time period
c1_last= C_1[-1]
c2_last= C_2[-1]
c3_last= C_3[-1]
c4_last= C_4[-1]
print(c1_last,c2_last,c3_last,c4_last)
```

```
(0.07163154321263285+0.49483693878262586j) (-0.27458090752610537+0.4178612
1089609j) (-0.27458090752610537+0.41786121089609j) (-0.0281469677948874-0.
4992523659517184j)
```

In [102]:

```
#finding the fidelity of the system
#The intial coefficient for |psi> are a,b,c,d=0.5, after tau time the coefficient changes
#with the  state |psi>= a|11>+b|1r>+c|r1>+d|rr>)
#expected result |phi>= 0.5(|11>+|1r>+|r1>+e^(i pi)|rr>)
#The fidentility is given by F=|<phi|psi>|^2
#gives a measure of how accurate the system is
overlap=0.5*(c1_last+c2_last+c3_last)+ (-0.5)*c4_last
Fidelity=np.abs(overlap)**2
print(Fidelity)
```

```
0.8875390771220254
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: