final

April 28, 2025

[379]:
```
import numpy as np
import matplotlib.pyplot as plt
```

[380]:
```
# Fig Size for display
figX = 6.4
figY = 4.8
```

1 Introduction

2 Method

2.1 Part A

We will solve the Hamiltonian analytically,

$$\hat{H} = \begin{bmatrix} E & t \\ t^* & -E \end{bmatrix}.$$

Taking the determinant of the Hamiltonian less the identity matrix multiplied by the energy states and setting it to 0,

$$\begin{aligned} \hat{H} - \epsilon I &= \begin{bmatrix} E-\epsilon & t \\ t & -E-\epsilon \end{bmatrix} \\ |\hat{H} - \epsilon I| &= (E-\epsilon)(-E-\epsilon) - t^2 = 0 \\ &= -E^2 + \epsilon^2 - t^2 \\ \epsilon &= \pm\sqrt{E^2+t^2}. \end{aligned}$$

We obtain the ground energy level as

$$\epsilon_0 = -\sqrt{E^2+t^2}$$

and the energy of the first excited state as

$$\epsilon_1 = +\sqrt{E^2+t^2}.$$

The energy difference is then,

$$\Delta\epsilon = \epsilon_1 - \epsilon_0 = 2\sqrt{E^2 + t^2}.$$

By inspection, we find the minimum energy difference occurs when $E = t = 0$, where $\Delta\epsilon = 0$. To find the probability of measuring a '0' state or a '1' state, we first need to solve the equation,

$$\hat{H}|\psi_{0/1}>=\epsilon|\psi_{0/1}>$$

where $\psi_0 = c_1|0> + c_2|1>$ and $\psi_1 = c_3|0> + c_4|1>$. Through an excessive amount of alegbra, we were able to obtain for ψ_0

$$c_1 = \frac{t}{\sqrt{t^2 + (\sqrt{E^2+t^2}+E)^2}}$$
$$c_2 = \frac{\sqrt{E^2+t^2}+E}{\sqrt{t^2 + (\sqrt{E^2+t^2}+E)^2}}$$

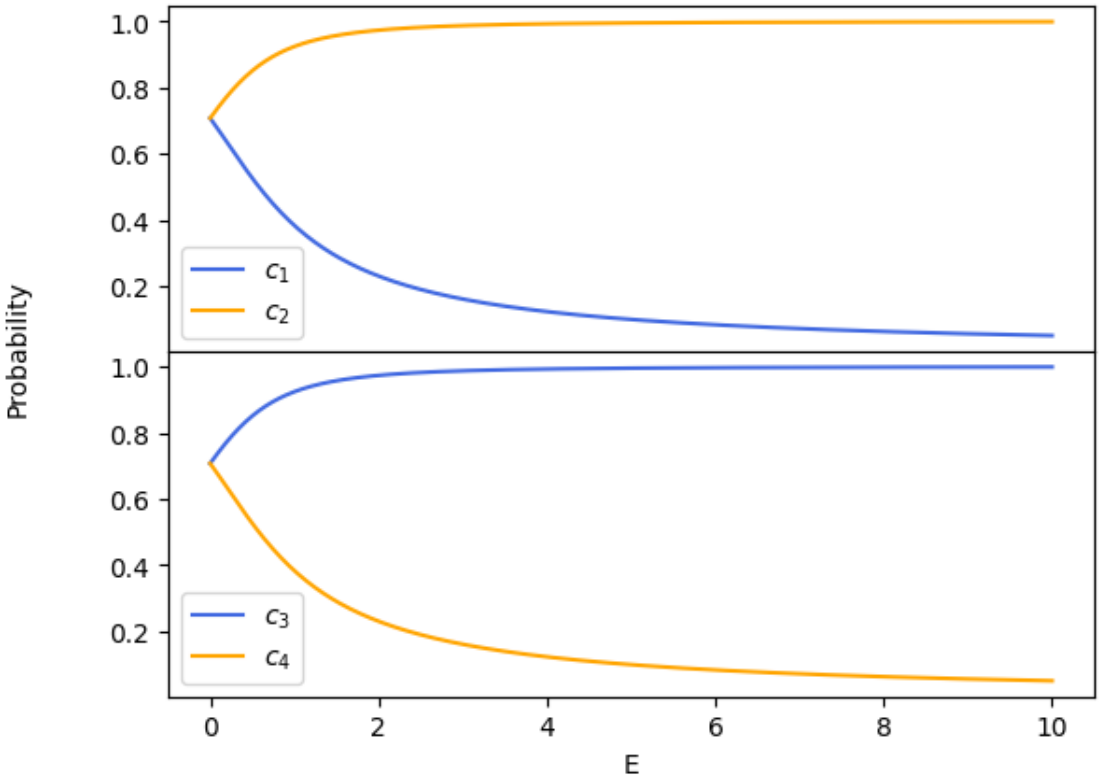
and for ψ_1

$$c_3 = \frac{t}{\sqrt{t^2 + (\sqrt{E^2+t^2}-E)^2}}$$
$$c_4 = \frac{\sqrt{E^2+t^2}-E}{\sqrt{t^2 + (\sqrt{E^2+t^2}-E)^2}}$$

Let us now plot these probabilities for some fixed value of t.

[381]:
```
# Plotting code for probabilities above
t = 1
E = np.linspace(0,10,500)

c1 = t/np.sqrt(t**2+(np.sqrt(E**2+t**2)+E)**2)
c2 = (np.sqrt(E**2+t**2)+E)/np.sqrt(t**2+(np.sqrt(E**2+t**2)+E)**2)
c3 = t/np.sqrt(t**2+(np.sqrt(E**2+t**2)-E)**2)
c4 = (np.sqrt(E**2+t**2)-E)/np.sqrt(t**2+(np.sqrt(E**2+t**2)-E)**2)

f, axes = plt.subplots(2,1,sharex=True,figsize=(figX,figY))
plt.setp(axes,label='Time (s)')
axes[0].plot(E,c1,label=r'$c_1$',c='royalblue')
axes[0].plot(E,c2,label=r'$c_2$',c='orange')
axes[1].plot(E,c3,label=r'$c_3$',c='royalblue')
axes[1].plot(E,c4,label=r'$c_4$',c='orange')
axes[-1].set_xlabel('E')
```

```
axes[0].legend()
axes[1].legend()
f.text(0, 0.5, 'Probability', ha='center', va='center', rotation='vertical')
f.subplots_adjust(hspace=0)
```



2.2 Code constant across all plots

[382]:
```
# Constants
hbar = 1.055e-34
q = 1.602e-19
m = 9.109e-31 * 0.2

# Generating the mesh
Np = 300
a = 1e-10 #in metres
X = a*np.linspace(-Np//2,Np//2,Np) #in metres

# Define Hamiltonian as a tridiagonal matrix
def H_build(Potential):
    t0=(hbar*hbar)/(2*m*a*a) #working in Joules
    on=2.0*t0*np.ones(Np)
```

```
    off=-t0*np.ones(Np-1)
    return np.diag(on+Potential) + np.diag(off,1) + np.diag(off,-1)

# Return probability distributions for ground and first excited state
def eigenstates(H):
    eigval,eigvec=np.linalg.eig(H)
    idx = eigval.argsort()[::1]
    eigval = eigval[idx]
    eigvec = eigvec[:,idx]
    dE = eigval[1] - eigval[0]
    Psi_0 = eigvec[:,0]
    Psi_1 = eigvec[:,1]
    Psi_0_squared = np.abs(Psi_0)**2
    Psi_1_squared = np.abs(Psi_1)**2
    Psi_0_squared /= np.sum(Psi_0_squared)
    Psi_1_squared /= np.sum(Psi_1_squared)
    return Psi_0, Psi_1, dE, Psi_0_squared, Psi_1_squared, eigval[0], eigval[1]
```

2.2.1 Different potentials

[383]:
```
# Single quantum dot
def V_SQD(frequency): # frequency on order of 1e15 - frequency of harmonic␣
 ↪oscillator
    alpha = 0.5*m*frequency**2
    return alpha*(X**2)

# Double quantum dot (static)
def V_DQD(separation_distance,frequency): # r on order of 1e-9 - input as nm -␣
 ↪separation distance between quantum dot centres
    alpha = 0.5*m*frequency**2
    r = separation_distance / 2 * 1e-9
    V1 = alpha * (X + r)**2
    V2 = alpha * (X - r)**2
    return np.minimum(V1, V2)

# Double quantum dot (with detuning bias)
def V_DQDF(F,frequency,separation_distance,t=0): #F on order of 1e11 - ␣
 ↪electric field strength
    return V_DQD(separation_distance,frequency) + F*q*X*np.cos(frequency*t)
```

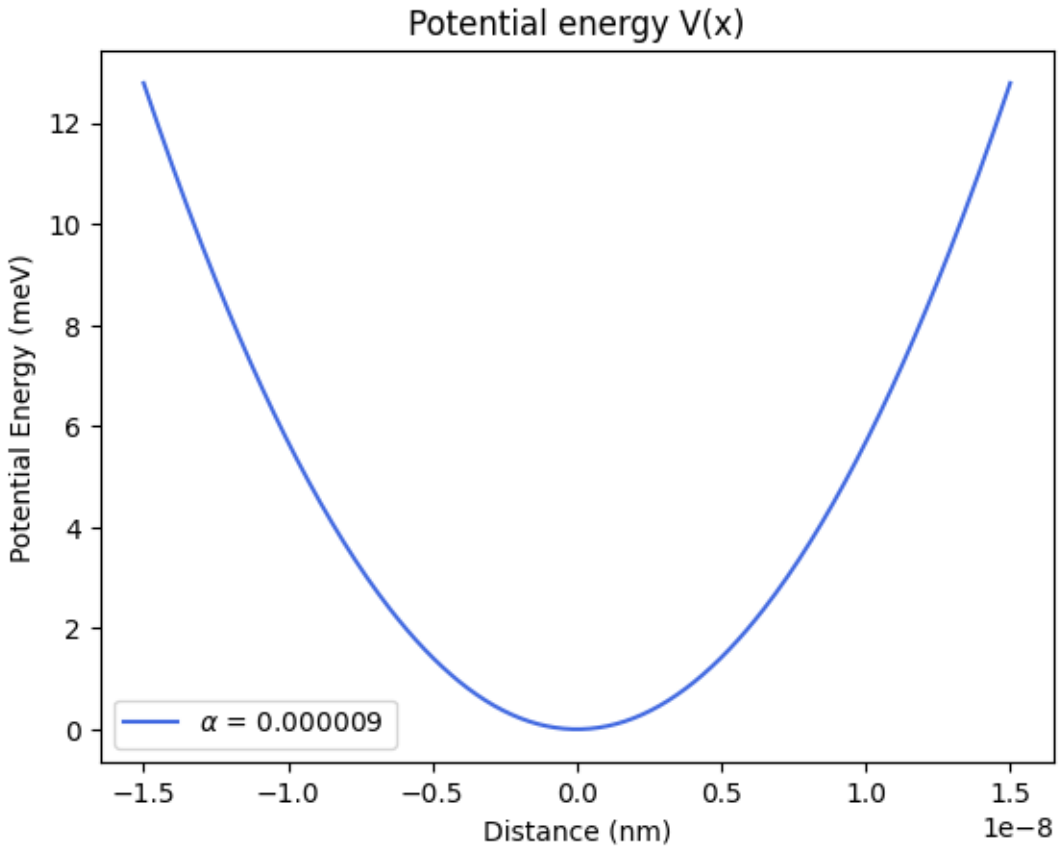
2.3 Results

2.3.1 Part B

(i)

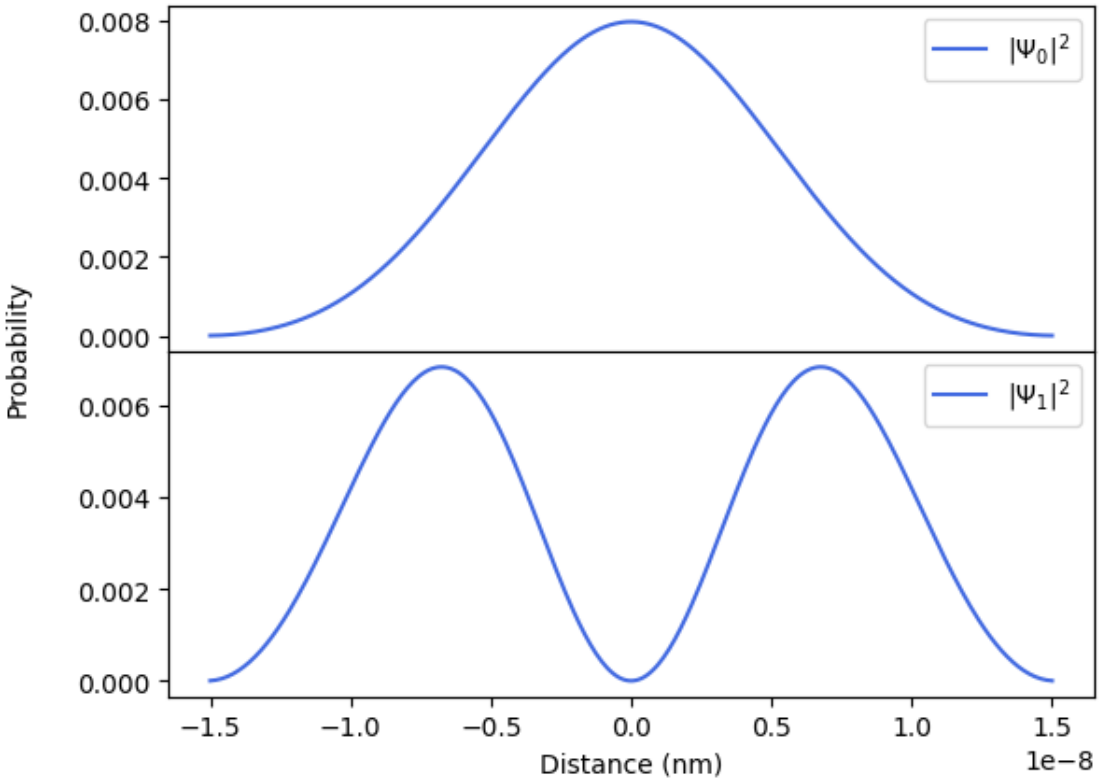
[384]:
```
omega = 1e13
alpha = 0.5*m*omega**2
```

```
plt.figure(figsize=(figX,figY))
plt.plot(
    X,
    1e3*V_SQD(omega)/q,
    c='royalblue',
    label=rf'$\alpha$ = {alpha:.6f}'
)

plt.title('Potential energy V(x)')
plt.xlabel('Distance (nm)')
plt.ylabel('Potential Energy (meV)')
plt.legend()
plt.show()
```



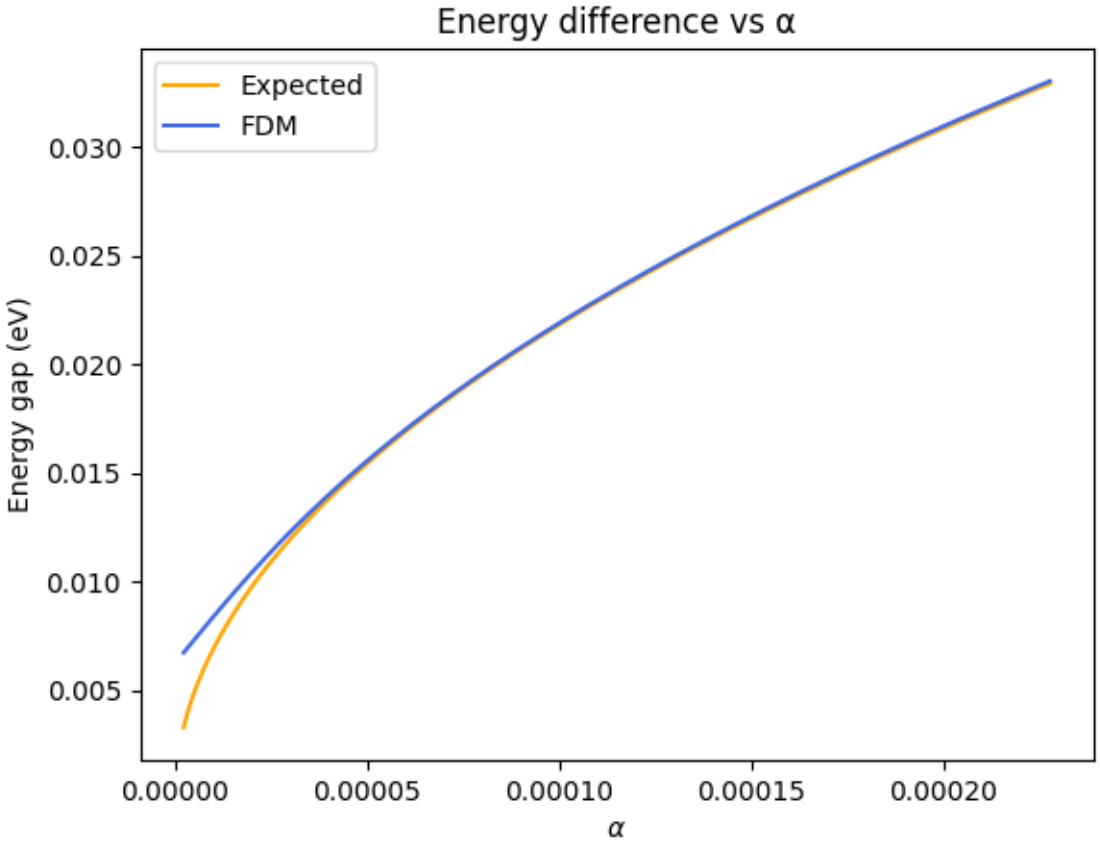
[385]:
```
omega = 1e13
alpha = 0.5*m*omega**2

f, axes = plt.subplots(2,1,sharex=True,figsize=(figX,figY))
```

```
plt.setp(axes,label='Time (s)')
axes[0].
  ↪plot(X,eigenstates(H_build(V_SQD(omega)))[3],c='royalblue',label=r'$|\Psi_0|^2$')
axes[1].
  ↪plot(X,eigenstates(H_build(V_SQD(omega)))[4],c='royalblue',label=r'$|\Psi_1|^2$')
axes[-1].set_xlabel('Distance (nm)')
axes[0].legend()
axes[1].legend()
f.text(0, 0.5, 'Probability', ha='center', va='center', rotation='vertical')
f.subplots_adjust(hspace=0)
```



[386]:
```
omega = np.linspace(5e12,5e13,Np)
alpha = 0.5 * m * omega**2
energy_diff = []

for i in alpha:
    V = i * (X**2)
    H = H_build(V)
    energy_diff.append(eigenstates(H)[2])

energy_diff = np.array(energy_diff)
```

```
def dE(alpha):
    return hbar*np.sqrt(2*alpha/m)

plt.figure(figsize=(figX,figY))
plt.plot(alpha, dE(alpha)/q,c='orange',label='Expected')
plt.plot(alpha, energy_diff/q,c='royalblue',label='FDM')

plt.xlabel(r'$\alpha$')
plt.ylabel('Energy gap (eV)')
plt.title('Energy difference vs  ')
plt.legend(loc='upper left')
plt.show()
```



To obtain an energy splitting of 4 meV, we would need to solve the following pair of equations for α, the energy difference in a harmonic oscillator is given by

$$\Delta E = \bar{H}\omega$$

and the frequency of the oscillator is related to the parameter α by

$$\alpha = \frac{1}{2}m\omega^2$$
$$\omega = \sqrt{\frac{2\alpha}{m}}.$$

This leads us to

$$\alpha = \frac{(\Delta E)^2 m}{2\bar{h}^2}.$$

Subtituting in the energy splitting of 4 meV, we obtain

$$\alpha = 3.361 \times 10^{-6}\ \mathrm{Nm}^{-1}.$$

To engineer lower values of α, we want the electrons to have less energy spacing. The frequency of the oscillator needs to be lower, in a classical analogy, the restoring force of the particle needs to be weak. This means that the electron is less tightly confined and its wavefunction is wider. This means that increasing the quantum dot size will lower the value of α. By decreasing the effective mass of the electron by changing the material of the quantum dot then alpha would also decrease however this can be achieved relatively easily. Furthermore, additional energy states beyond the first excited state may have greater spacing so limiting the electron to only populating the ground and first excited state would be ideal.

From Boltzmann distribution, the mean energy of an electron is given as

$$\bar{E} = \frac{1}{2}\hbar\omega + \frac{\hbar\omega}{e^{\beta\hbar\omega} - 1}$$

where $\beta = 1/k_B T$

In the classical limit, $k_B T \gg \hbar\omega$, $\beta\hbar\omega \to 0$ and so

$$\bar{E} = \frac{1}{2}\hbar\omega + k_B T.$$

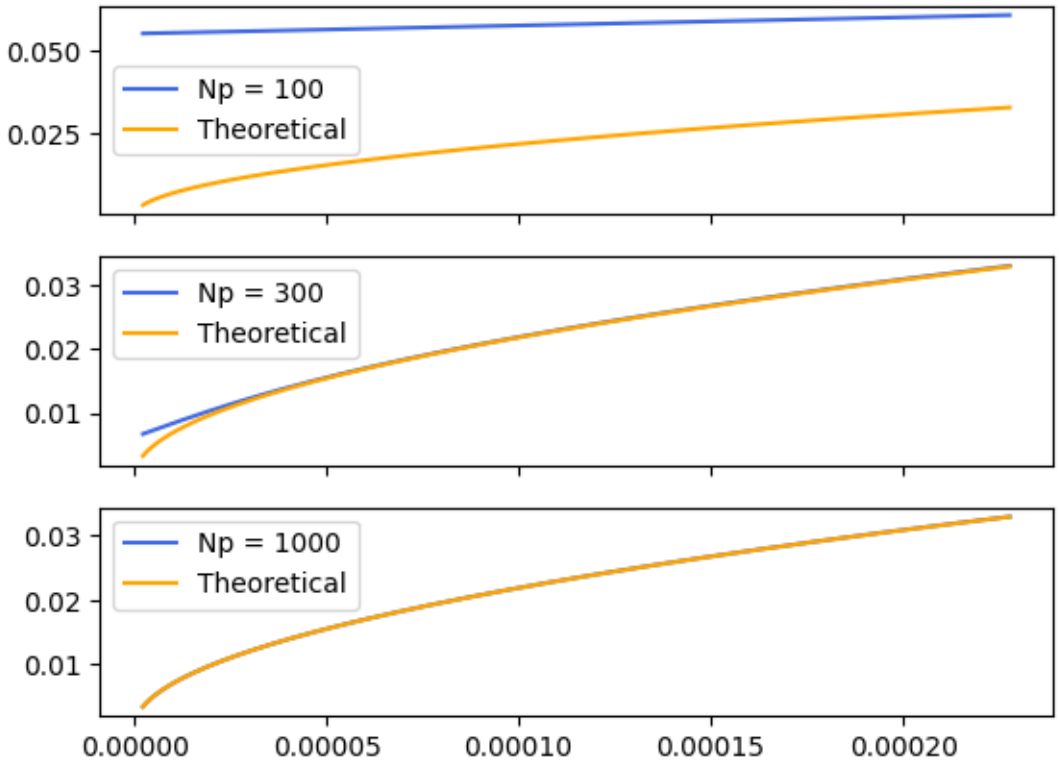
This is unideal for our situation as we want the mean energy of the electron to be as low as possible so if $kT \ll \hbar\omega$,

$$\bar{E} = \frac{1}{2}\hbar\omega.$$

We find that now the mean energy is independent of T. This is a quantum effect known as 'Freeze out', where modes of the gas are frozen, i.e the degrees of freedom of the gas is lowered. This means that the higher energy states of the electron are restricted. Therefore, for the quantum dot, we want to work in the regime of $kT \ll \hbar\omega$, or just close to zero at cryogenic temperatures.

(ii) To ensure our results do not suffer from numerical artifacts, the amount of points in the mesh was steadily increased where the plot of energy difference versus alpha was matching theoretical results.

[415]:
```
Np_array = [100,300,1000]
a = 1e-10 #in metres

def dE(alpha):
    return hbar*np.sqrt(2*alpha/m)

def H_build_test(Potential,Np):
    t0=(hbar*hbar)/(2*m*a*a) #working in Joules
    on=2.0*t0*np.ones(Np)
    off=-t0*np.ones(Np-1)
    return np.diag(on+Potential) + np.diag(off,1) + np.diag(off,-1)

f,axes = plt.subplots(3,1,sharex=True,figsize=(figX,figY))
plt.setp(axes,label=r'$\alpha$')

for i in range(len(Np_array)):
    Np = Np_array[i]
    X = a*np.linspace(-Np//2,Np//2,Np)
    omega = np.linspace(5e12,5e13,Np)
    alpha = 0.5 * m * omega**2
    energy_diff = []

    for j in alpha:
        V = j * (X**2)
        H = H_build_test(V,Np)
        energy_diff.append(eigenstates(H)[2])

    energy_diff = np.array(energy_diff)
    axes[i].plot(alpha,energy_diff/q,label=f'Np = {Np_array[i]}',c='royalblue')
    axes[i].plot(alpha,dE(alpha)/q,label='Theoretical',c='orange')
    axes[i].legend()
```



(iii) The energy of a particle in a 3D cube in the ground state is

$$E_0 = \frac{3\pi^2\hbar^2}{2mL^2}$$

where L is the length of the cube. In this scenario, the 'cube' can be thought of as the confinement of the electron, defined by its wavefunction. From Part B(i), we know that the size of the quantum dot and its material is what defines this confinement length. The energy of the first excited state is

$$E_1 = \frac{3\pi^2\hbar^2}{mL^2}$$

and so the energy difference is simply equal to the ground state energy as $E_1 = 2E_0$. Rearranging the equation for the energy difference to obtain the characteristic length of the quantum dot,

$$L = \sqrt{\frac{3\pi^2\hbar^2}{2m\Delta E}}.$$

For a 4 meV energy split, $L = 3.76 \times 10^{-8}$ m. Therefore, if atoms in a silicon crystal along a 1D chain are spaced by 0.543 nm, the ground state of a quantum dot wavefunction will span for approximately 69 atoms.

The main reason why quantum dots are often called 'artificial atoms' are because of the discrete energy spectrum of the system. This occurs when the size of the quantum dot is comparable to the wavelength of the electrons that occupy it. This can be seen in hydrogen, where its Bohr radius is 5.29×10^{-11} m or 0.529A. The ground state energy of hydrogen is -13.6 eV and so the wavelength of the electrons that occupy it can be found using de Broglie's relation,

$$\lambda = \frac{2\pi\hbar}{\sqrt{2mE}}.$$

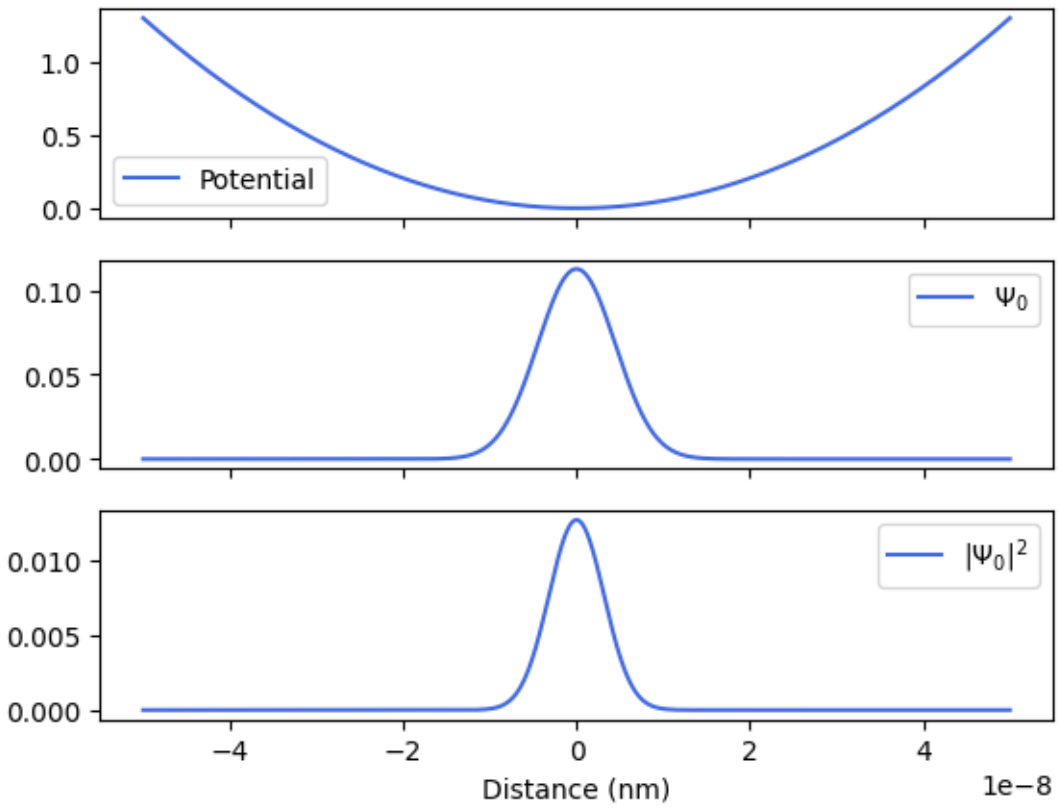
For an electron at ground state, the wavelength would be 3.33A $\sim$ 0.529A.

2.3.2 Part C

(i)

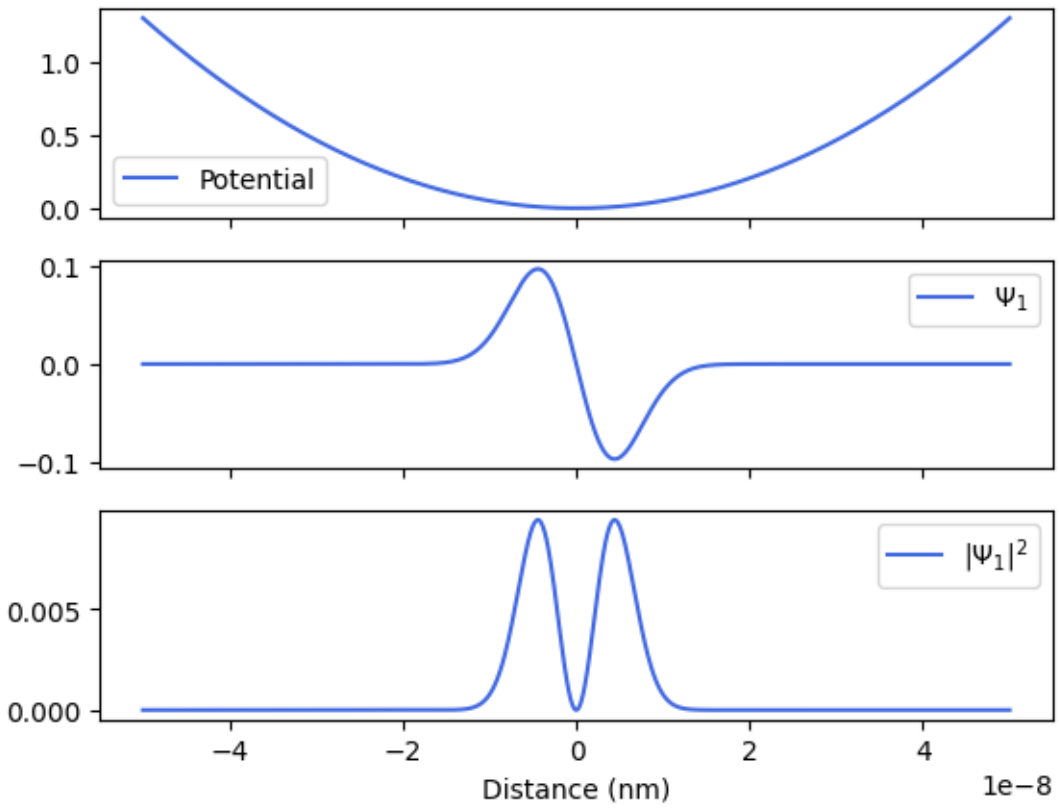
[419]:
```
alpha = (2e-2*q)**2*m/(2*hbar**2)
omega = np.sqrt(2*m*alpha)/m
r = 0.5

f, axes = plt.subplots(3,1,sharex=True,figsize=(figX,figY))
plt.setp(axes,label='Time (s)')
axes[0].plot(X,V_DQD(r,omega)/q,c='royalblue',label='Potential')
axes[1].
 ↪plot(X,eigenstates(H_build(V_DQD(r,omega)))[0],c='royalblue',label=r'$\Psi_0$')
axes[2].
 ↪plot(X,eigenstates(H_build(V_DQD(r,omega)))[3],c='royalblue',label=r'$|\Psi_0|^2$')
axes[-1].set_xlabel('Distance (nm)')
axes[0].legend()
axes[1].legend()
axes[2].legend()
```

[419]:
```
<matplotlib.legend.Legend at 0x30337dfd0>
```



[420]:
```
alpha = (2e-2*q)**2*m/(2*hbar**2)
omega = np.sqrt(2*m*alpha)/m
r = 0.5

f, axes = plt.subplots(3,1,sharex=True,figsize=(figX,figY))
plt.setp(axes,label='Time (s)')
axes[0].plot(X,V_DQD(r,omega)/q,c='royalblue',label='Potential')
axes[1].
  ↪plot(X,eigenstates(H_build(V_DQD(r,omega)))[1],c='royalblue',label=r'$\Psi_1$')
axes[2].
  ↪plot(X,eigenstates(H_build(V_DQD(r,omega)))[4],c='royalblue',label=r'$|\Psi_1|^2$')
axes[-1].set_xlabel('Distance (nm)')
axes[0].legend()
axes[1].legend()
axes[2].legend()
```

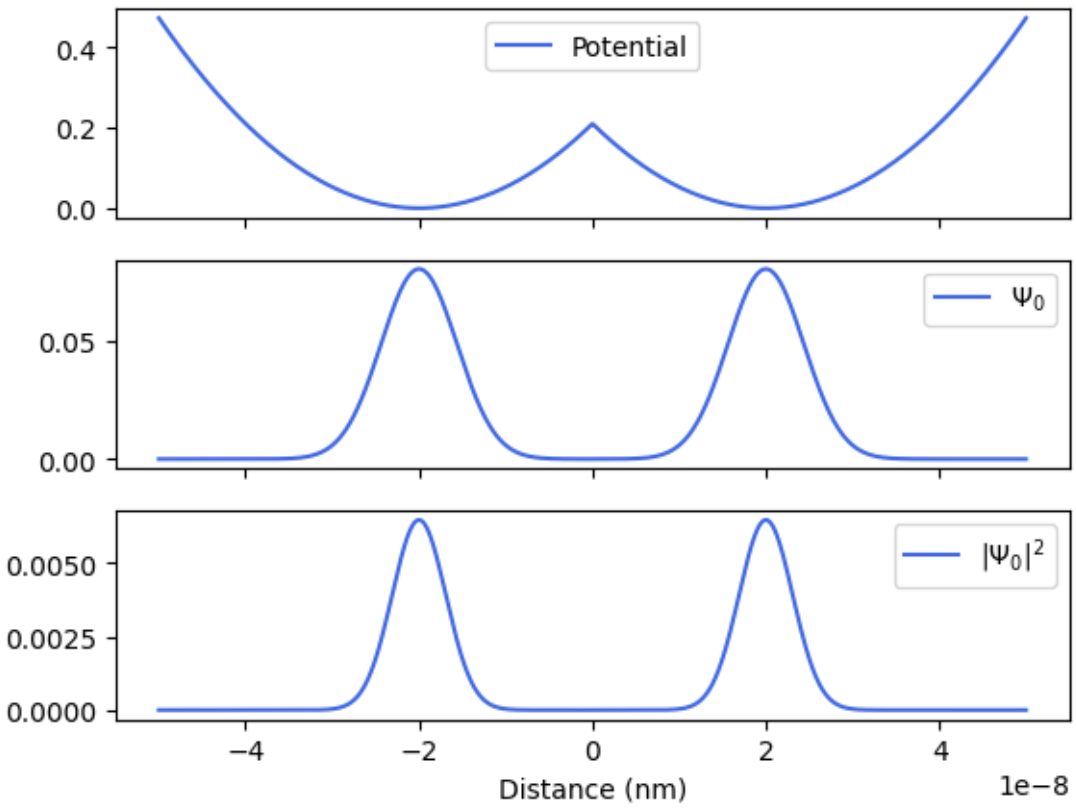
[420]:
```
<matplotlib.legend.Legend at 0x303468650>
```



As we can see, when the dot separation is small enough at 0.5 nm, the double quantum dot is effectively a single quantum dot as its wavefunction is identical for both dots. This is a strong coupling.

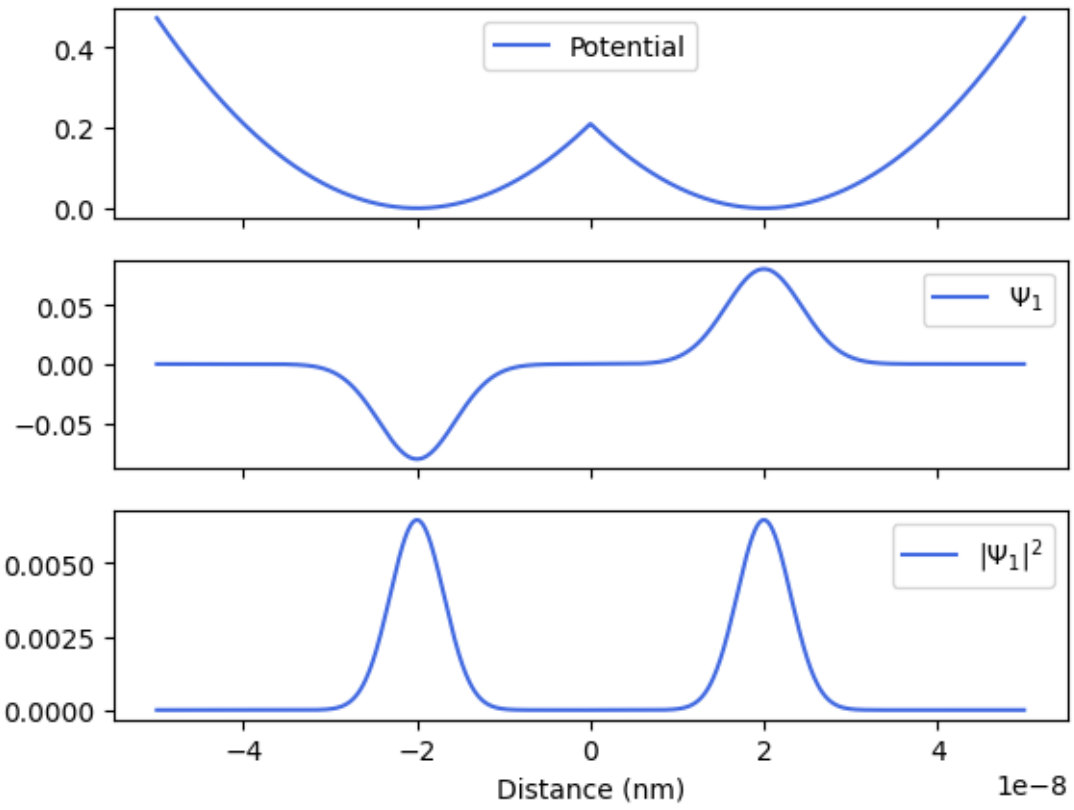
[447]:
```
alpha = (2e-2*q)**2*m/(2*hbar**2)
omega = np.sqrt(2*m*alpha)/m
r = 40

f, axes = plt.subplots(3,1,sharex=True,figsize=(figX,figY))
plt.setp(axes,label='Time (s)')
axes[0].plot(X,V_DQD(r,omega)/q,c='royalblue',label='Potential')
axes[1].
  ↪plot(X,eigenstates(H_build(V_DQD(r,omega)))[0],c='royalblue',label=r'$\Psi_0$')
axes[2].
  ↪plot(X,eigenstates(H_build(V_DQD(r,omega)))[3],c='royalblue',label=r'$|\Psi_0|^2$')
axes[-1].set_xlabel('Distance (nm)')
axes[0].legend()
axes[1].legend()
axes[2].legend()
```

[447]:
```
<matplotlib.legend.Legend at 0x30184e0f0>
```



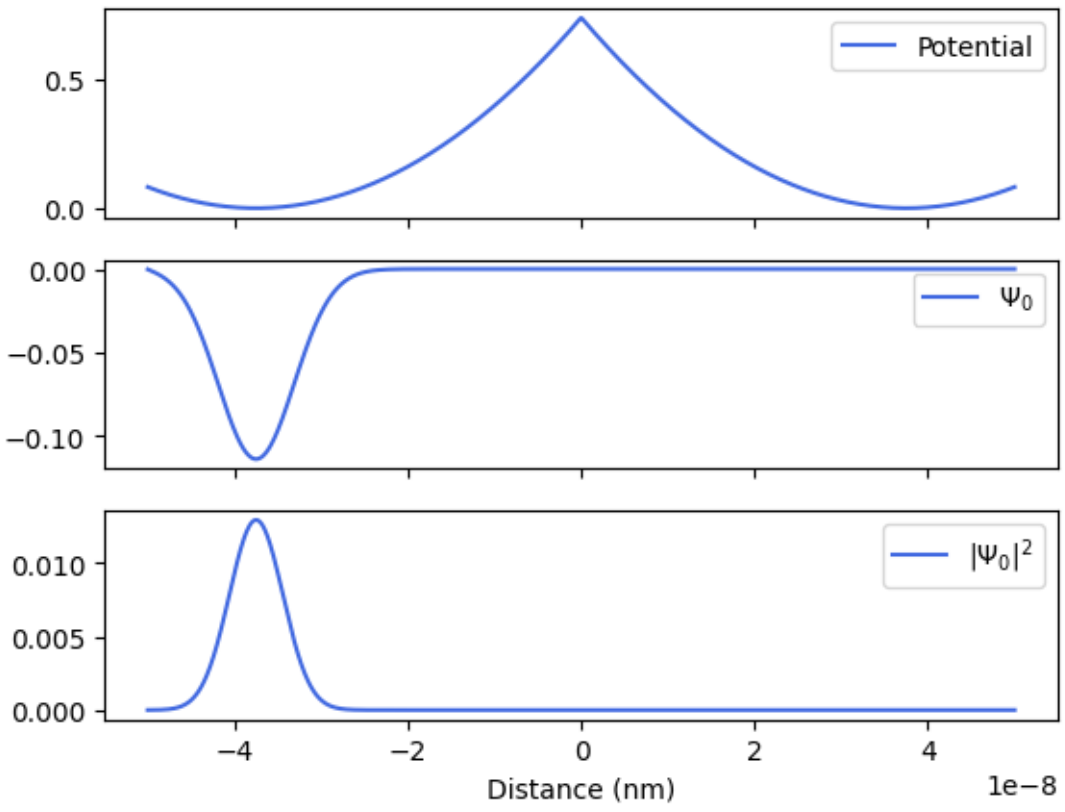
[448]:
```
alpha = (2e-2*q)**2*m/(2*hbar**2)
omega = np.sqrt(2*m*alpha)/m
r = 40

f, axes = plt.subplots(3,1,sharex=True,figsize=(figX,figY))
plt.setp(axes,label='Time (s)')
axes[0].plot(X,V_DQD(r,omega)/q,c='royalblue',label='Potential')
axes[1].
  ↪plot(X,eigenstates(H_build(V_DQD(r,omega)))[1],c='royalblue',label=r'$\Psi_1$')
axes[2].
  ↪plot(X,eigenstates(H_build(V_DQD(r,omega)))[4],c='royalblue',label=r'$|\Psi_1|^2$')
axes[-1].set_xlabel('Distance (nm)')
axes[0].legend()
axes[1].legend()
axes[2].legend()
```

[448]:
```
<matplotlib.legend.Legend at 0x3019767b0>
```



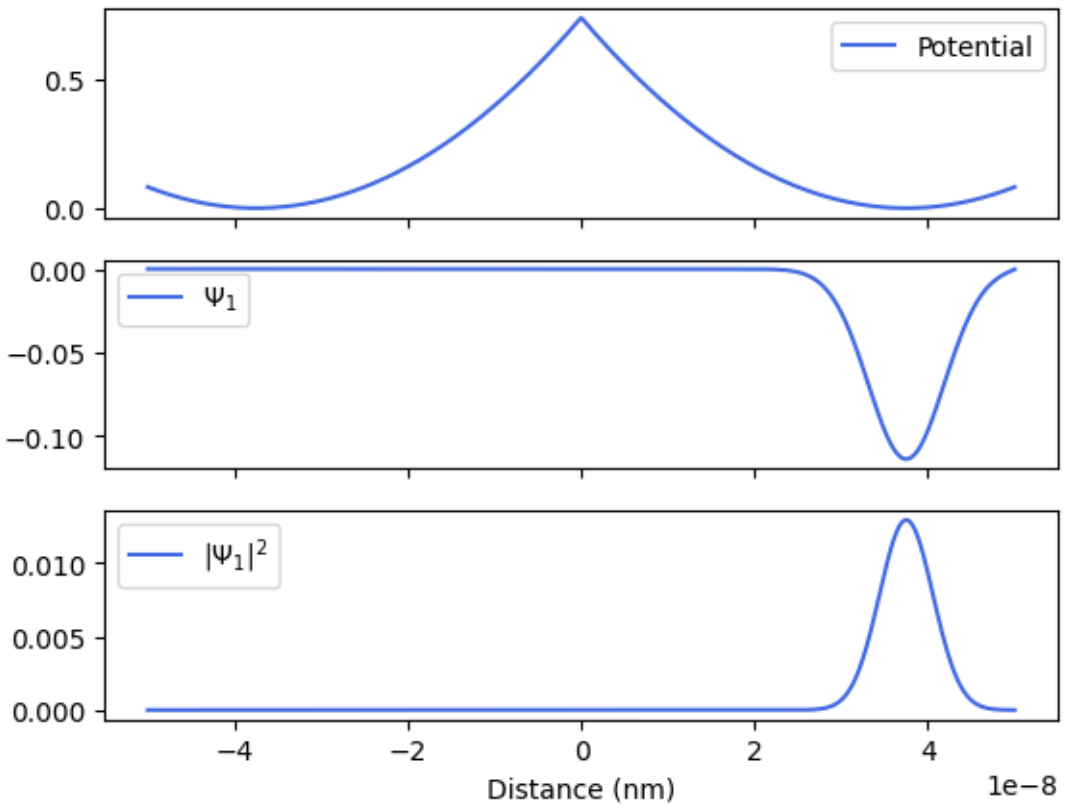
[438]:
```
alpha = (2e-2*q)**2*m/(2*hbar**2)
omega = np.sqrt(2*m*alpha)/m
r = 75

f, axes = plt.subplots(3,1,sharex=True,figsize=(figX,figY))
plt.setp(axes,label='Time (s)')
axes[0].plot(X,V_DQD(r,omega)/q,c='royalblue',label='Potential')
axes[1].
  ↪plot(X,eigenstates(H_build(V_DQD(r,omega)))[0],c='royalblue',label=r'$\Psi_0$')
axes[2].
  ↪plot(X,eigenstates(H_build(V_DQD(r,omega)))[3],c='royalblue',label=r'$|\Psi_0|^2$')
axes[-1].set_xlabel('Distance (nm)')
axes[0].legend()
axes[1].legend()
axes[2].legend()
```

[438]:
```
<matplotlib.legend.Legend at 0x17de948f0>
```



[439]:
```
alpha = (2e-2*q)**2*m/(2*hbar**2)
omega = np.sqrt(2*m*alpha)/m
r = 75

f, axes = plt.subplots(3,1,sharex=True,figsize=(figX,figY))
plt.setp(axes,label='Time (s)')
axes[0].plot(X,V_DQD(r,omega)/q,c='royalblue',label='Potential')
axes[1].
  ↪plot(X,eigenstates(H_build(V_DQD(r,omega)))[1],c='royalblue',label=r'$\Psi_1$')
axes[2].
  ↪plot(X,eigenstates(H_build(V_DQD(r,omega)))[4],c='royalblue',label=r'$|\Psi_1|^2$')
axes[-1].set_xlabel('Distance (nm)')
axes[0].legend()
axes[1].legend()
axes[2].legend()
```

[439]:
```
<matplotlib.legend.Legend at 0x17df6f1d0>
```



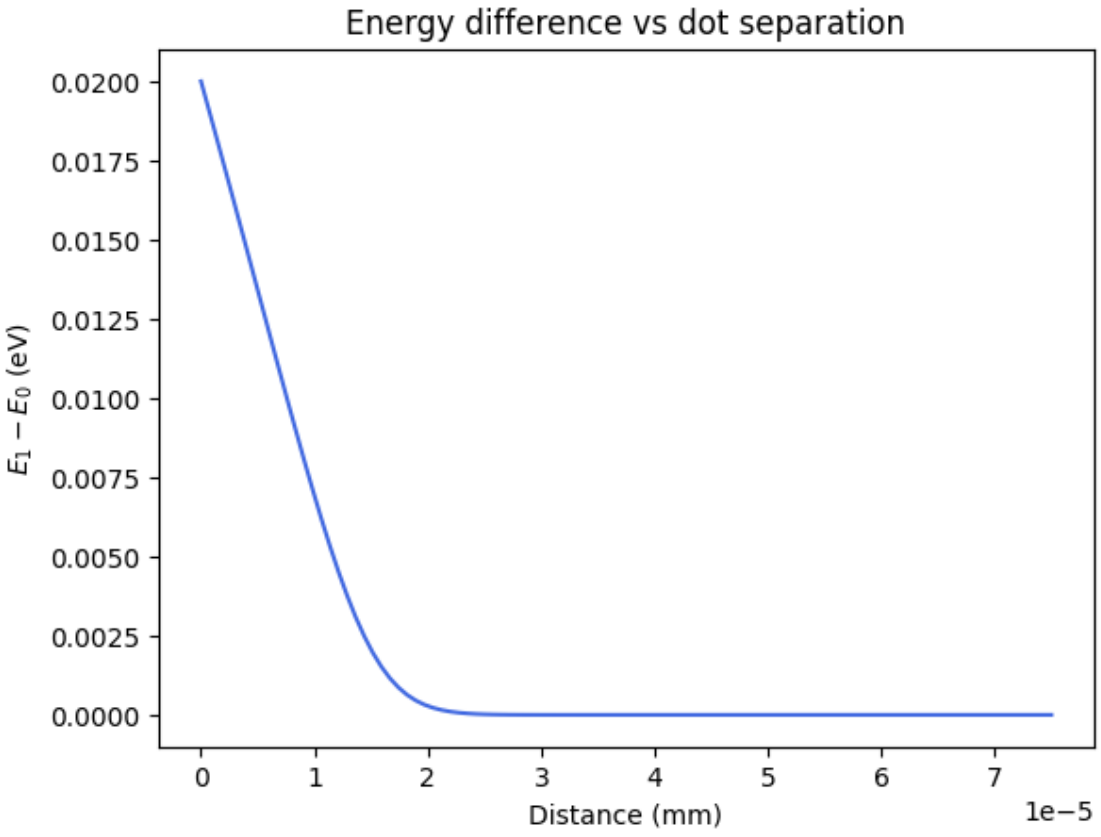
For the moderate and weak coupling, we find that their wavefunctions are less overlapping, indicating weaker quantum tunneling between the dots. The further away the two quantum dots, the more the energy level splits, breaking the initial degeneracy. For the moderately coupled double quantum dot, the lowest two wavefunctions display similar characteristics, being the two peaks at the same two locations. However they differ as in the ground state, the two peaks are in phase with each other whereas in the first excited state there is a clear π phase shift. As we raise the tunnel barrier to a height where there is no more tunneling, we find that the states have localised in each quantum dot. We find that the ground state electrons only exist and are the only existing electrons in the left quantum dot and likewise for the right quantum dot and the first excited state. This is likely due to the imperfections of numerical methods. Theoretically, the wavefunction should be delocalised across both wells, producing a symmetric ground state and an antisymmetric first excited state as we saw before. However, any slight asymmetry, which would easily arise in computational methods, there would be a preference for each state.

(ii)

[443]:
```
alpha = (2e-2*q)**2*m/(2*hbar**2)
omega = np.sqrt(2*m*alpha)/m
r_range = np.linspace(0,75,100)

dE = []
```

```
for i in r_range:
    V = V_DQD(i,omega)
    H = H_build(V)
    dE.append(eigenstates(H)[2])

dE = np.array(dE)

plt.figure(figsize=(figX,figY))
plt.title('Energy difference vs dot separation')
plt.xlabel('Distance (mm)')
plt.ylabel(r'$E_1 - E_0$ (eV)')
plt.plot(r_range*1e-6,dE/q,c='royalblue')
```

[443]: [<matplotlib.lines.Line2D at 0x17e27a030>]



We find that the energy level separation decays exponentially as the distance between the quantum dots increase. This is expected as the tunnel coupling strength is dependent on the distance,

$$t = e^{-\frac{d}{d_0}}$$

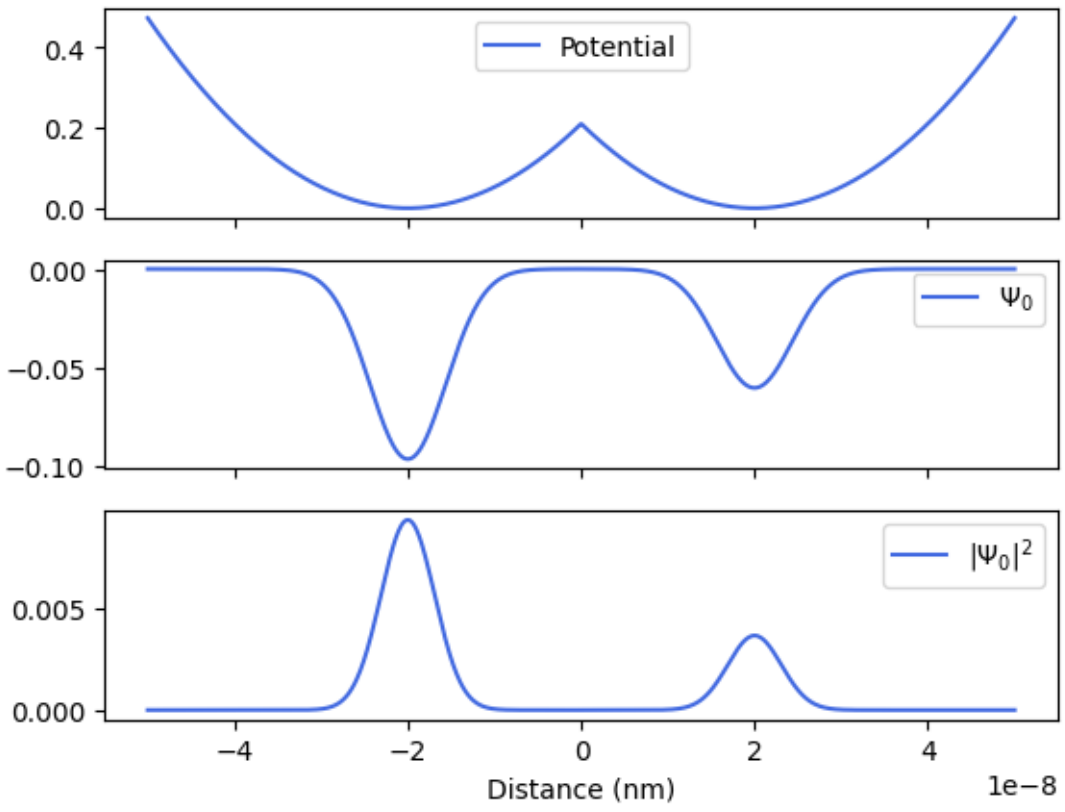
where d_0 is the characteristic length scale that depends on the height of the potential barrier and the electron's effective mass in the material. The energy splitting is directly proportional to tunnel coupling. This is because the localised states in each dot hybridise into bonding and antibonding combinations. Since the bonding states are lower in energy and the antibonding state is higher in energy, the overlap in the wavefunction makes the energy splitting between these two states larger. The bonding states occur when the two wavefunctions combine constructively whereas the antibonding states combine destructively. This was seen in the previous plots of the wavefunction, where the moderately coupled wavefunctions possess antisymmetric and symmetric wavefunctions for the ground and first excited state however this characteristic is lost when the distance between the dots is large enough. This tunnel coupling strength parameter is denoted by the t found in Part A. Therefore, this can not be present in the classical case as it relies on quantum effects. Physically, this coupling parameter is dependent on the height of the potential well which is determined by barrier height and width, effective mass of the electron and the energy of the electron relative to the barrier.

A smaller dot to dot separation is desirable but not too small as the energy levels become too degenerate. There is an optimal separation which is found by balancing the trade offs between suppressed tunneling and too much hybridisation. We want to be able to distinguish between two quantum dots whilst also being able to move electrons via tunneling between them.

(iii)

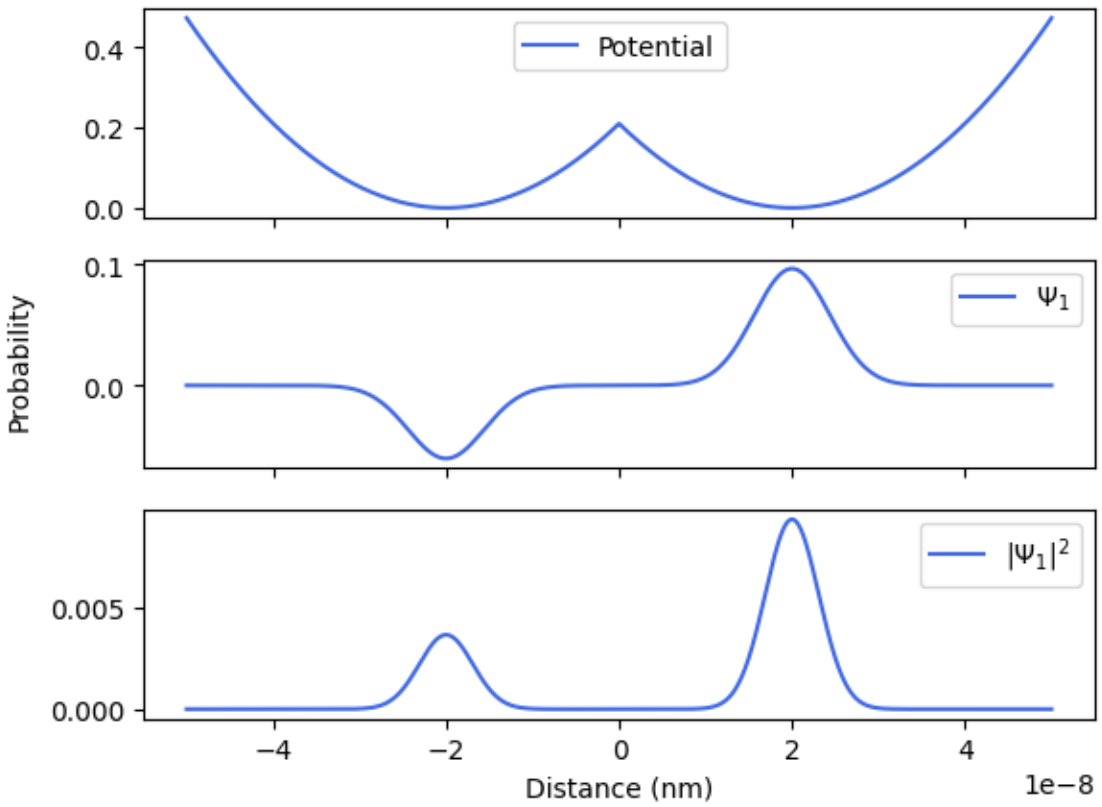
[454]:
```
alpha = (2e-2*q)**2*m/(2*hbar**2)
omega = np.sqrt(2*m*alpha)/m
r = 40
F = 1e-3

f, axes = plt.subplots(3,1,sharex=True,figsize=(figX,figY))
plt.setp(axes,label='Time (s)')
axes[0].plot(X,V_DQDF(F,omega,r)/q,c='royalblue',label='Potential')
axes[1].
 ↪plot(X,eigenstates(H_build(V_DQDF(F,omega,r)))[0],c='royalblue',label=r'$\Psi_0$')
axes[2].
 ↪plot(X,eigenstates(H_build(V_DQDF(F,omega,r)))[3],c='royalblue',label=r'$|\Psi_0|^2$')
axes[-1].set_xlabel('Distance (nm)')
axes[0].legend()
axes[1].legend()
axes[2].legend()
```

[454]:
```
<matplotlib.legend.Legend at 0x305b61100>
```



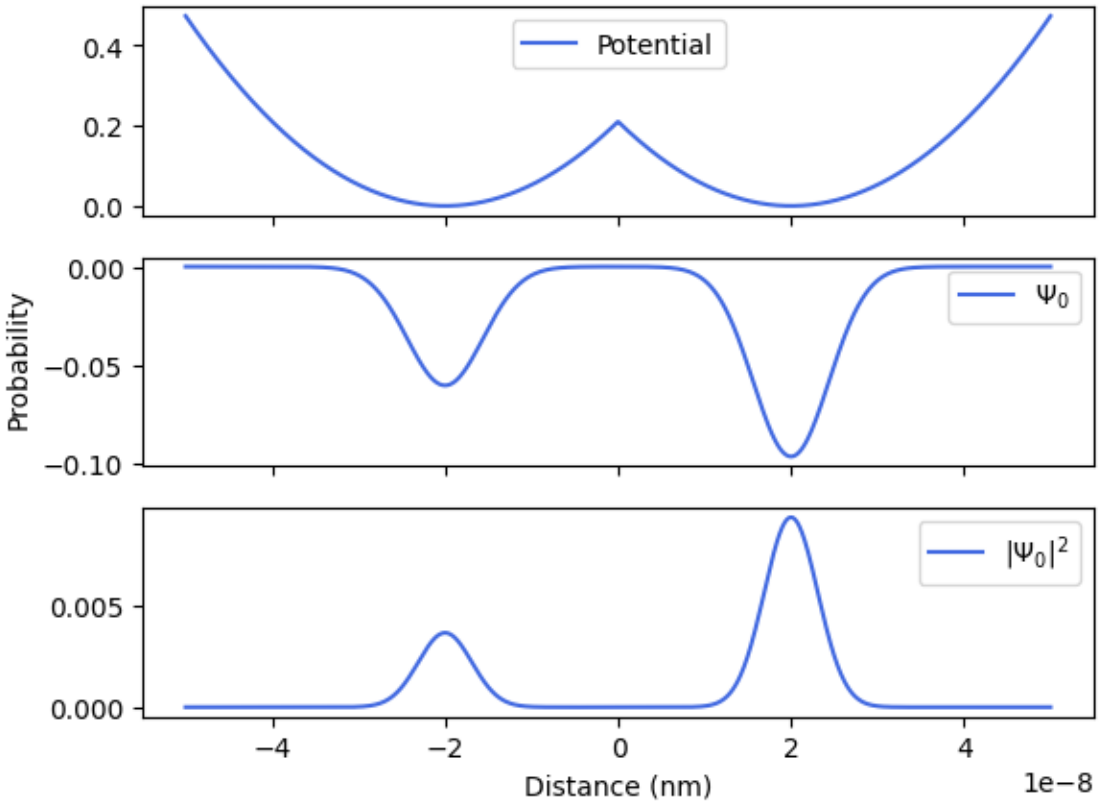
[455]:
```
alpha = (2e-2*q)**2*m/(2*hbar**2)
omega = np.sqrt(2*m*alpha)/m
r = 40
F = 1e-3

f, axes = plt.subplots(3,1,sharex=True,figsize=(figX,figY))
plt.setp(axes,label='Time (s)')
axes[0].plot(X,V_DQDF(F,omega,r)/q,c='royalblue',label='Potential')
axes[1].
  ↪plot(X,eigenstates(H_build(V_DQDF(F,omega,r)))[1],c='royalblue',label=r'$\Psi_1$')
axes[2].
  ↪plot(X,eigenstates(H_build(V_DQDF(F,omega,r)))[4],c='royalblue',label=r'$|\Psi_1|^2$')
axes[-1].set_xlabel('Distance (nm)')
axes[0].legend()
axes[1].legend()
axes[2].legend()
f.text(0.025, 0.5, 'Probability', ha='center', va='center', rotation='vertical')
```

[455]:
```
Text(0.025, 0.5, 'Probability')
```



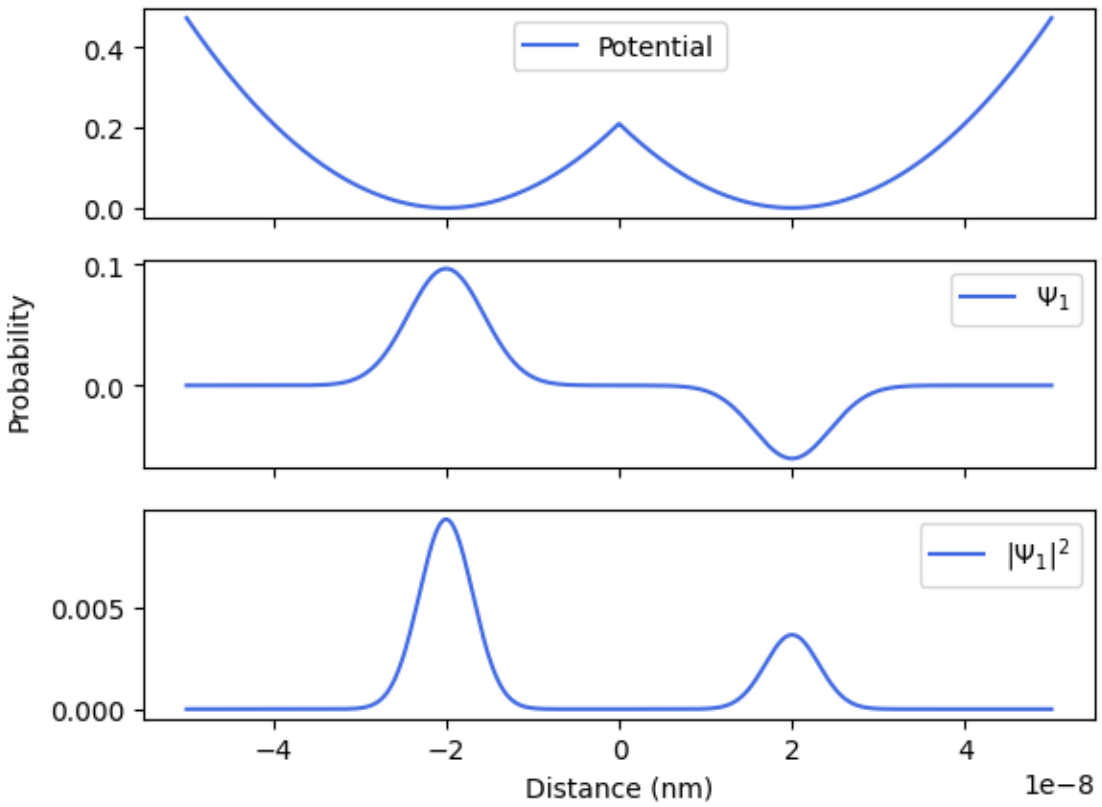
[456]:
```
alpha = (2e-2*q)**2*m/(2*hbar**2)
omega = np.sqrt(2*m*alpha)/m
r = 40
F = -1e-3

f, axes = plt.subplots(3,1,sharex=True,figsize=(figX,figY))
plt.setp(axes,label='Time (s)')
axes[0].plot(X,V_DQDF(F,omega,r)/q,c='royalblue',label='Potential')
axes[1].
  ↪plot(X,eigenstates(H_build(V_DQDF(F,omega,r)))[0],c='royalblue',label=r'$\Psi_0$')
axes[2].
  ↪plot(X,eigenstates(H_build(V_DQDF(F,omega,r)))[3],c='royalblue',label=r'$|\Psi_0|^2$')
axes[-1].set_xlabel('Distance (nm)')
axes[0].legend()
axes[1].legend()
axes[2].legend()
f.text(0.025, 0.5, 'Probability', ha='center', va='center', rotation='vertical')
```

[456]:
```
Text(0.025, 0.5, 'Probability')
```



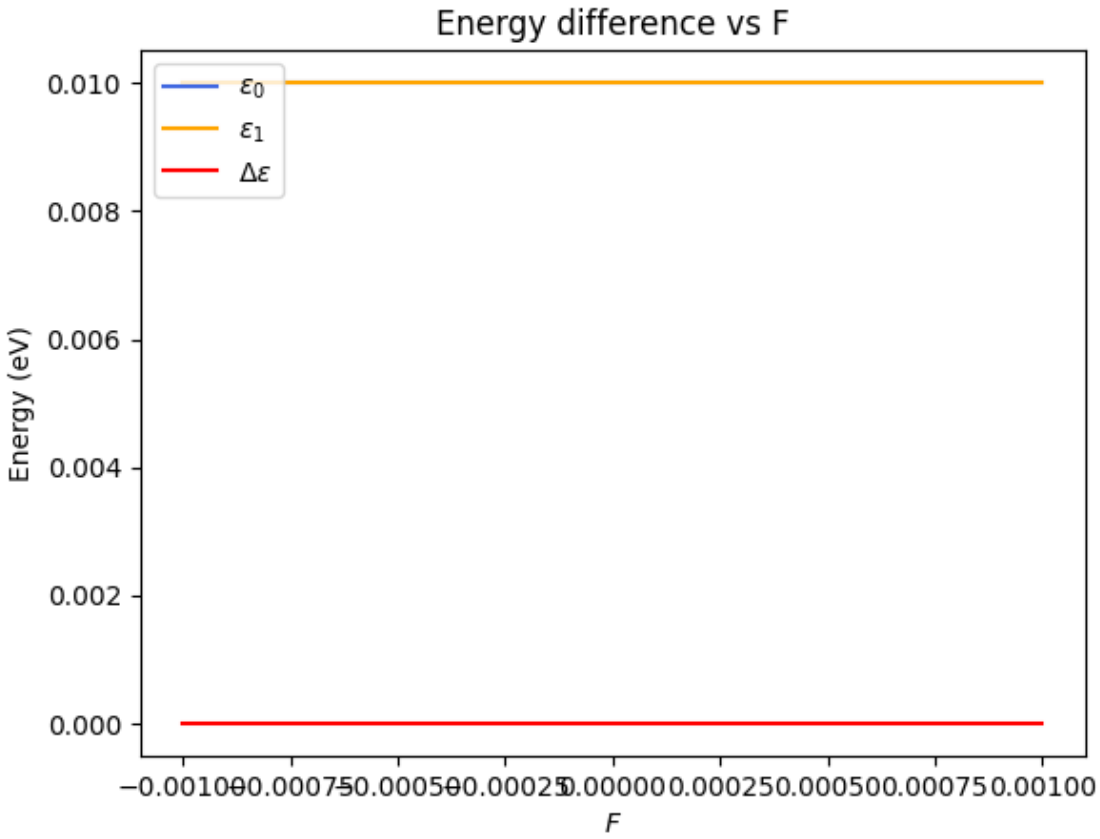
[458]:
```
alpha = (2e-2*q)**2*m/(2*hbar**2)
omega = np.sqrt(2*m*alpha)/m
r = 40
F = -1e-3

f, axes = plt.subplots(3,1,sharex=True,figsize=(figX,figY))
plt.setp(axes,label='Time (s)')
axes[0].plot(X,V_DQDF(F,omega,r)/q,c='royalblue',label='Potential')
axes[1].
 ↪plot(X,eigenstates(H_build(V_DQDF(F,omega,r)))[1],c='royalblue',label=r'$\Psi_1$')
axes[2].
 ↪plot(X,eigenstates(H_build(V_DQDF(F,omega,r)))[4],c='royalblue',label=r'$|\Psi_1|^2$')
axes[-1].set_xlabel('Distance (nm)')
axes[0].legend()
axes[1].legend()
axes[2].legend()
f.text(0.025, 0.5, 'Probability', ha='center', va='center', rotation='vertical')
```

[458]:
```
Text(0.025, 0.5, 'Probability')
```



[ ]:
```
alpha = (2e-2*q)**2*m/(2*hbar**2)
omega = np.sqrt(2*m*alpha)/m
r = 40

F_range = np.linspace(-1e-3,1e-3,100)

E0 = []
E1 = []

for i in F_range:
    V = V_DQDF(i,omega,r)
    H = H_build(V)
    E0.append(eigenstates(H)[5])
    E1.append(eigenstates(H)[6])

E0 = np.array(E0)
E1 = np.array(E1)

plt.figure(figsize=(figX,figY))
plt.plot(F_range*1e3,E0/q,c='royalblue',label=r'$\epsilon_0$')
plt.plot(F_range*1e3,E1/q,c='orange',label=r'$\epsilon_1$')
```

```
plt.plot(F_range*1e3,(E1-E0)/q,c='red',label=r'$\Delta \epsilon$')

plt.xlabel(r'$F$')
plt.ylabel('Energy (eV)')
plt.title('Energy difference vs F')
plt.legend(loc='upper left')
plt.show()
```



2.4 Part D

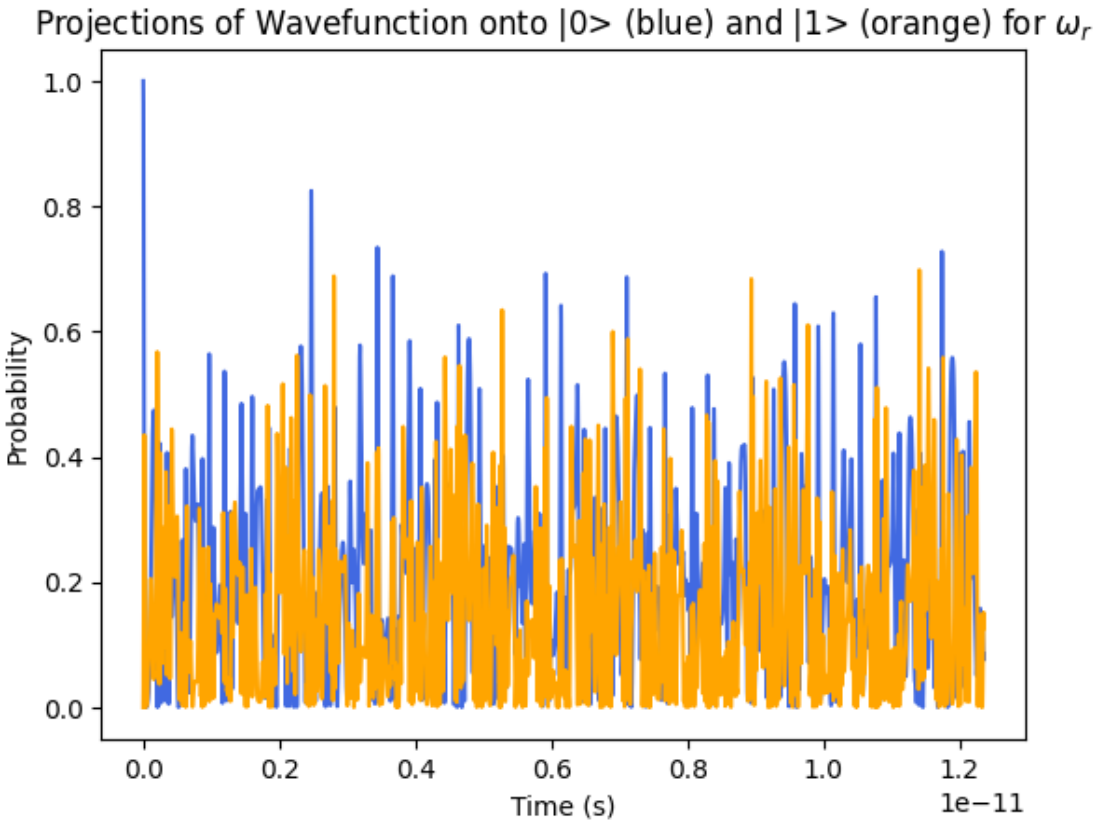
(i)

[408]:
```
r = 4
alpha = (4e-3*q)**2*m/(2*hbar**2)
omega = np.sqrt(2*m*alpha)/m

# No detuning bias for resonant frequency
omega_r = eigenstates(H_build(V_DQD(r,omega)))[2]/hbar

# With detuning
```

```
psi_0 = eigenstates(H_build(V_DQD(r,omega_r)))[0]
psi_1 = eigenstates(H_build(V_DQD(r,omega_r)))[1]
F = 1e7

t_min = 0
t_max = 2 * np.pi / (omega_r)
print(t_max)
dt = t_max/500
t_range = np.arange(t_min,20*t_max,dt)

proj_0_t = []
proj_1_t = []

psi = psi_0.copy()

for t in t_range:
    Vt = V_DQDF(F,omega_r,r,t)
    Ht = H_build(Vt)

    c0 = np.vdot(psi_0, psi)
    c1 = np.vdot(psi_1, psi)

    p0 = np.abs(c0)**2
    p1 = np.abs(c1)**2

    proj_0_t.append(p0)
    proj_1_t.append(p1)

    Vt_next = V_DQDF(F,omega_r,r,t+dt)
    Ht_next = H_build(Vt_next)

    A = np.eye(Np) + 1j * dt/(2*hbar) * Ht_next
    B = np.eye(Np) - 1j * dt/(2*hbar) * Ht

    psi_new = np.linalg.solve(A,B @ psi)
    psi = psi_new / np.linalg.norm(psi_new)

proj_0_t = np.array(proj_0_t)
proj_1_t = np.array(proj_1_t)

plt.figure(figsize=(figX,figY))
plt.plot(t_range, proj_0_t, label="Projection onto |0> (blue)",c='royalblue')
plt.plot(t_range, proj_1_t, label="Projection onto |1> (orange)",c='orange')
plt.xlabel("Time (s)")
plt.ylabel("Probability")
plt.title(r"Projections of Wavefunction onto |0> (blue) and |1> (orange) for␣
 ↪$\omega_r$")
```

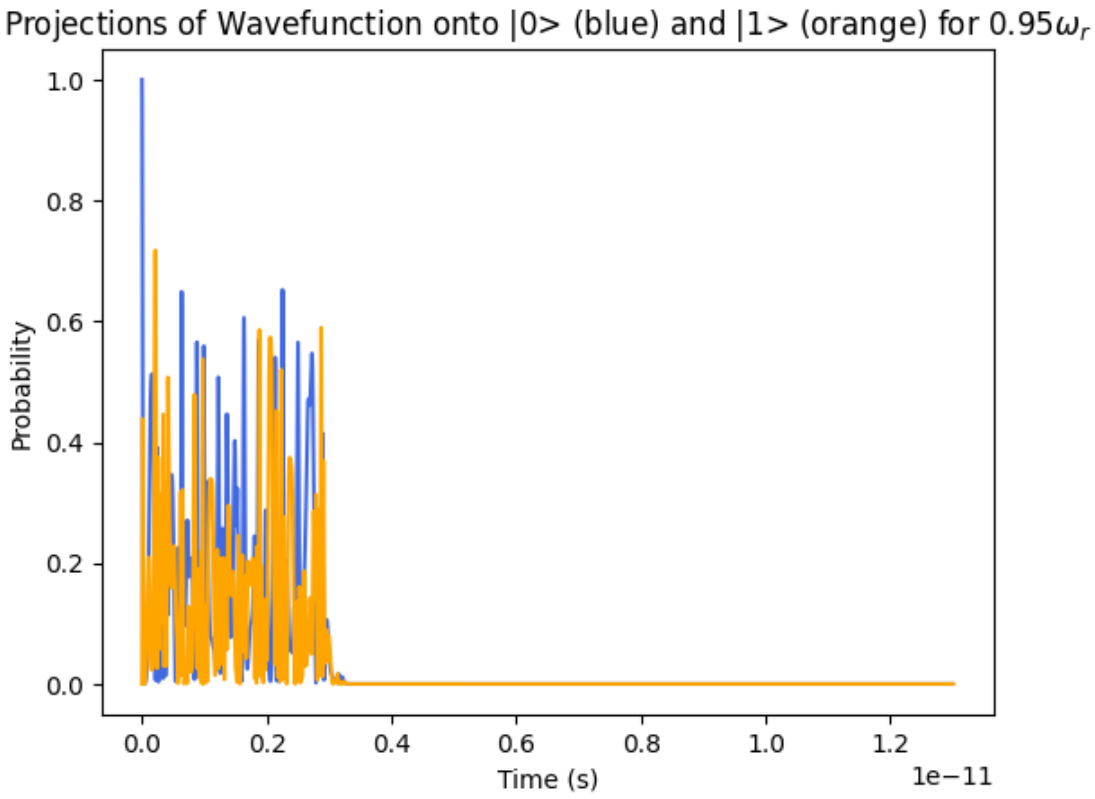
```
plt.show()
```

```
6.177331259529757e-13
```



[400]:
```
r = 4
alpha = (4e-3*q)**2*m/(2*hbar**2)
omega = np.sqrt(2*m*alpha)/m

# No detuning bias for resonant frequency
omega_r_095 = 0.95*eigenstates(H_build(V_DQD(r,omega)))[2]/hbar

# With detuning
psi_0 = eigenstates(H_build(V_DQD(r,omega_r_095)))[0]
psi_1 = eigenstates(H_build(V_DQD(r,omega_r_095)))[1]
F = 1e7

t_min = 0
t_max = 2 * np.pi // (omega_r_095)
dt = t_max/500
t_range = np.arange(t_min,20*t_max,dt)
```

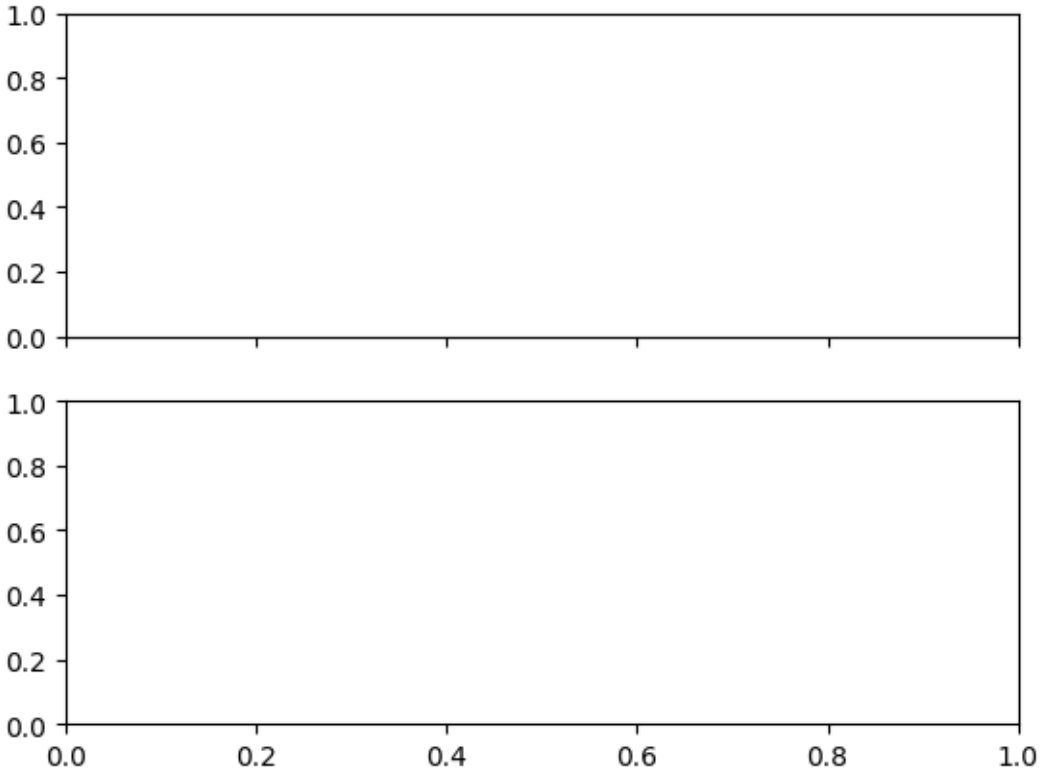
```
proj_0_t_095 = []
proj_1_t_095 = []

psi = psi_0.copy()

for t in t_range:
    Vt = V_DQDF(F,omega_r_095,r,t)
    Ht = H_build(Vt)

    c0 = np.vdot(psi_0, psi)
    c1 = np.vdot(psi_1, psi)

    p0 = np.abs(c0)**2
    p1 = np.abs(c1)**2

    proj_0_t_095.append(p0)
    proj_1_t_095.append(p1)

    Vt_next = V_DQDF(F,omega_r,r,t+dt)
    Ht_next = H_build(Vt_next)

    A = np.eye(Np) + 1j * dt/(2*hbar) * Ht_next
    B = np.eye(Np) - 1j * dt/(2*hbar) * Ht

    psi_new = np.linalg.solve(A,B @ psi)
    psi = psi_new / np.linalg.norm(psi_new)

proj_0_t_095 = np.array(proj_0_t_095)
proj_1_t_095 = np.array(proj_1_t_095)

plt.figure(figsize=(figX,figY))
plt.plot(t_range, proj_0_t_095, label="Projection onto |0>␣
 ↪(blue)",c='royalblue')
plt.plot(t_range, proj_1_t_095, label="Projection onto |1> (orange)",c='orange')
plt.xlabel("Time (s)")
plt.ylabel("Probability")
plt.title(r"Projections of Wavefunction onto |0> (blue) and |1> (orange) for $0.
 ↪95\omega_r$")
plt.show()
```



[403]:
```
f, axes = plt.subplots(2,1,sharex=True,figsize=(figX,figY))
plt.setp(axes,label='Time (s)')
axes[0].plot(t_range,proj_0_t,label=r'$p_0$ for $\omega_r$',c='royalblue')
axes[0].plot(t_range,proj_1_t,label=r'$p_1$ for $\omega_r$',c='orange')
axes[1].plot(t_range,proj_0_t_095,label=r'$p_0$ for $0.
 ↪95\omega_r$',c='royalblue')
axes[1].plot(t_range,proj_1_t_095,label=r'$p_1$ for $0.95\omega_r$',c='orange')
axes[-1].set_xlabel('Time (s)')
f.text(0.05, 0.5, 'Probability', ha='center', va='center', rotation='vertical')
f.subplots_adjust(hspace=0)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[403], line 3
      1 f, axes = plt.subplots(2,1,sharex=True,figsize=(figX,figY))
      2 plt.setp(axes,label='Time (s)')
----> 3 axes[0].plot(t_range,proj_1_t,label=r'$p_1$ for $\omega_r$',c='orange')
      4 axes[1].plot(t_range,proj_0_t_095,label=r'$p_0$ for $0.
 ↪95\omega_r$',c='royalblue')
      5 axes[1].plot(t_range,proj_1_t_095,label=r'$p_1$ for $0.
 ↪95\omega_r$',c='orange')
```

```
File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/
 ↪site-packages/matplotlib/axes/_axes.py:1779, in Axes.plot(self, scalex,␣
 ↪scaley, data, *args, **kwargs)
   1536 """
   1537 Plot y versus x as lines and/or markers.
   1538
   (…)
   1776 (``'green'``) or hex strings (``'#008000'``).
   1777 """
   1778 kwargs = cbook.normalize_kwargs(kwargs, mlines.Line2D)
-> 1779 lines = [*self._get_lines(self, *args, data=data, **kwargs)]
   1780 for line in lines:
   1781     self.add_line(line)

File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/
 ↪site-packages/matplotlib/axes/_base.py:296, in _process_plot_var_args.
 ↪__call__(self, axes, data, *args, **kwargs)
    294     this += args[0],
    295     args = args[1:]
--> 296 yield from self._plot_args(
    297     axes, this, kwargs, ambiguous_fmt_datakey=ambiguous_fmt_datakey)

File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/
 ↪site-packages/matplotlib/axes/_base.py:486, in _process_plot_var_args.
 ↪_plot_args(self, axes, tup, kwargs, return_kwargs, ambiguous_fmt_datakey)
    483     axes.yaxis.update_units(y)
    485 if x.shape[0] != y.shape[0]:
--> 486     raise ValueError(f"x and y must have same first dimension, but "
    487                      f"have shapes {x.shape} and {y.shape}")
    488 if x.ndim > 2 or y.ndim > 2:
    489     raise ValueError(f"x and y can be no greater than 2D, but have "
    490                      f"shapes {x.shape} and {y.shape}")

ValueError: x and y must have same first dimension, but have shapes (10001,) an␣
 ↪(10000,)
```


1.0
0.8
0.6
0.4
0.2
0.0
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0