Computational Essay- Qubit Simulation

S. Rager z5558988 UNSW PHYS3112

Abstract

The aim was to numerically solve the Schrodinger equation, understand semiconductor charge qubits based on simulations, assess quality of solutions, and identify numerical artifacts. It was concluded that a value of alpha=2.7e-5 produced an energy level splitting of 4 meV. It was determined that the theoretical qubits simulated in this experiment would function between the temperature range of 2.44 to 3.79 K, and that the magnitude of any detuning bias applied must be less than 7.5 meV for the qubits to remain operational.

Aim

Numerically solve the Schrodinger equation and understand semiconductor charge qubits based on simulations, assess quality of solutions, and identify numerical artifacts.

Introduction

A quantum bit or qubit is a two-level quantum system that forms the basis states for computation. Analogous to classical systems, these two-levels and their corresponding wavefunctions can be represented by '0' and '1'. The main difference between classical bits and qubits is the utilization of the quantum mechanical phenomenon, called superposition, which allows a qubit to simultaneously be in the '0' and '1' state. This is in contrast to a classical bit which can only be in the '0' or '1' state at any given moment in time. Mathematically, the qubit state |Ψ⟩ is writen as |Ψ⟩=a|0⟩+b|1⟩ where a and b are normalized complex numbers. A measurement on this state distinguishes between '0' and '1' with probabilities $|a|^2$ and $|b|^2$ resepctively. A quantum logic gate controls the superposition between |0⟩ and |1⟩. To use an example, a quantum X operation, the analogue of the classical NOT operation, flips the state of the qubit so that
|Ψ⟩=b|0⟩+a|1⟩. This operation can be denoted by the Pauli-X matrix σ_x=[[0,1],[1,0]] multplied by [[a],[b]] to get [[b],[a]]. A quantum S phase gate creates a phase between |0⟩ and |1⟩ so that S|Ψ⟩=a|0⟩+b $e^{i\theta}$|1⟩. A two-level quantum system mapped to the quibit can be modeled by the Hamiltonian H=[[E,t],[t*,-E]].

This Hamiltonian can be engineered in a metal-ozide-semiconductor used in silicon quantum computing. The electrostatic potential of each metal gate of a metal-oxide-semiconductor(MOS) traps an electron near the surface of silicon and forms a quantum dot. Using the discrete quantum states of the trapped electron, qubits can be implemented. To choose a basis of '0' and '1' one must select two energy levels whose superposition can be controlled. These energy levels must also be isolated from other energy levels in the system. Otherwise, decoherence and relaxation could occur, causing the loss of quantum information.

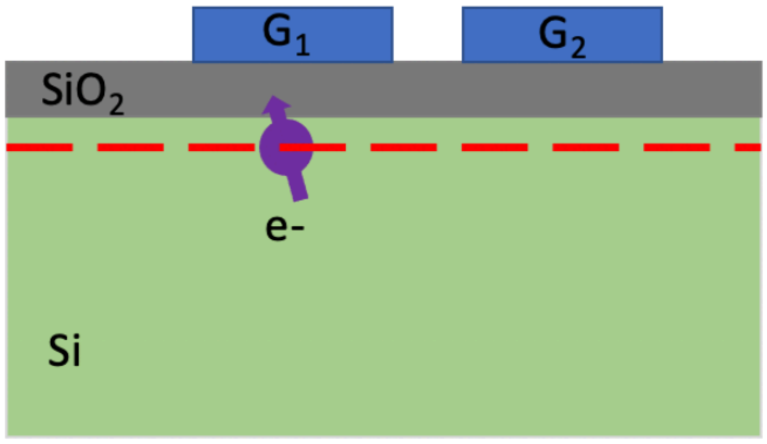


Figure 1 Double Quantum Dot device in silicon with a trapped electron under gate G₁. Either the charge or the spin of the electron can be used to encode quantum information. An insulating dielectric SiO₂ separates the metal gates from Si.

A "charge qubit" uses the localization of electronic charge to define a two-level quantum system. Whether an electron is localized in the left or right quantum dot can be measured with a charge sensor, typically through an electrostatic shift in current flowing between two metal contacts. In this case, the left corresponds to '0' and the right corresponds to '1'. This means one enery level, the ground state, can be used per quantum dot. By applying voltages to the gates, the electron can be moved from one dot to another. The spacing between the gates can also be controlled during fabrication, controlling the seperation distant between the dots. A simple model of this device can be solved using the finite difference method applied to the 1-dimensional time-independent Schrodinger equation. With energy eigenvalue E and wavefunction $\psi(x)$, the equation to be solved is: $-\frac{\hbar^2}{2m*}\frac{d^2\psi}{dx^2}+V(x)\psi(x)=E\psi(x)$, m*=0.2$m_0$ where m_0 is electron rest mass. For each quantum dot, the electrostatic potential from the gates along the separation axis can be approximated as a parabolic potential of the form V(x)=$\alpha(x-x_1)^2$ where x_1 is the center of the quantum dot and α is a constant that defines the curvature of the potential. x denotes the separation axis of the dots. For a double quantum dot, the potential can be approximated as V(x)=min[$\alpha(x-x_1)^2$,$\alpha(x-x_2)^2$] where x_1 and x_2 are the centers of the two quantum dots.

Results & Analysis

Part A

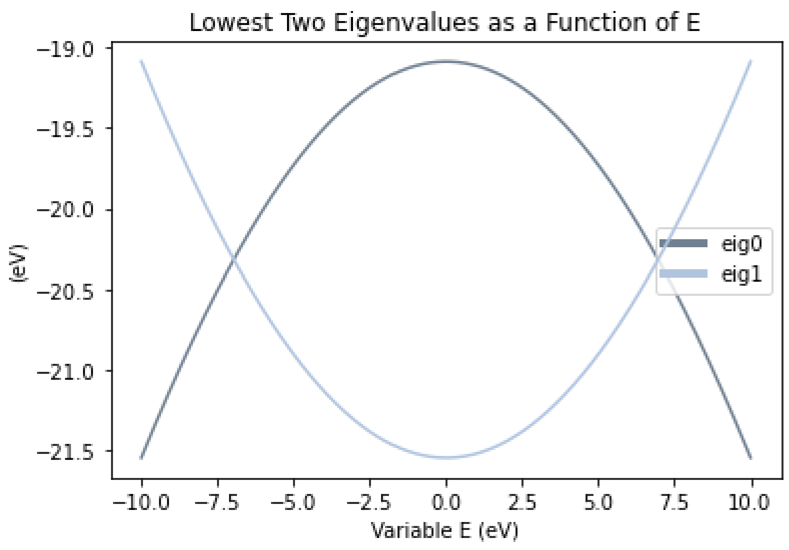
(i) Objectives:

1. Analytically solve for eigenvalues and eigenvectors of the 2 × 2 qubit Hamiltonian H=[[E, t], [t*, -E]].
2. Plot the two energy levels (eigenvalues) as a function of the variable E (for some reasonable choices of E & t, assuming t is real).
3. What is the minimum value of $E_2 - E_1$ in terms of t?
4. Plot the probability of measuring '0' and that of '1' for the two wavefunctions (eigenvectors) as a function of E.

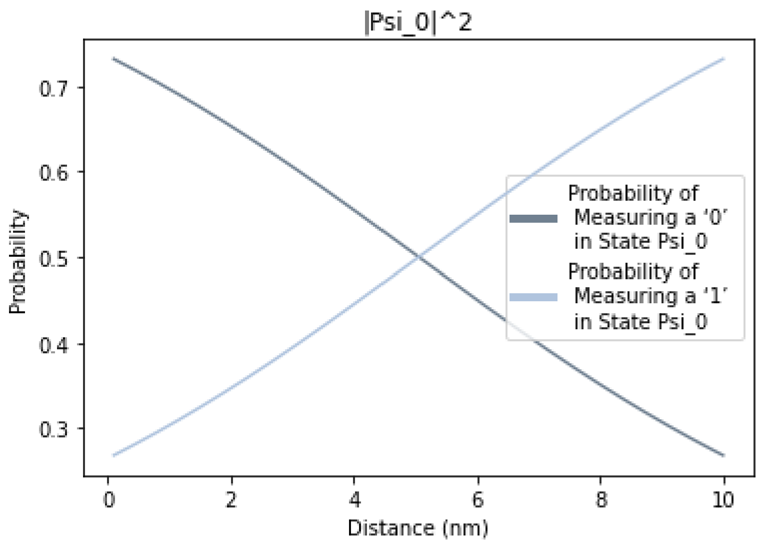
In [126…

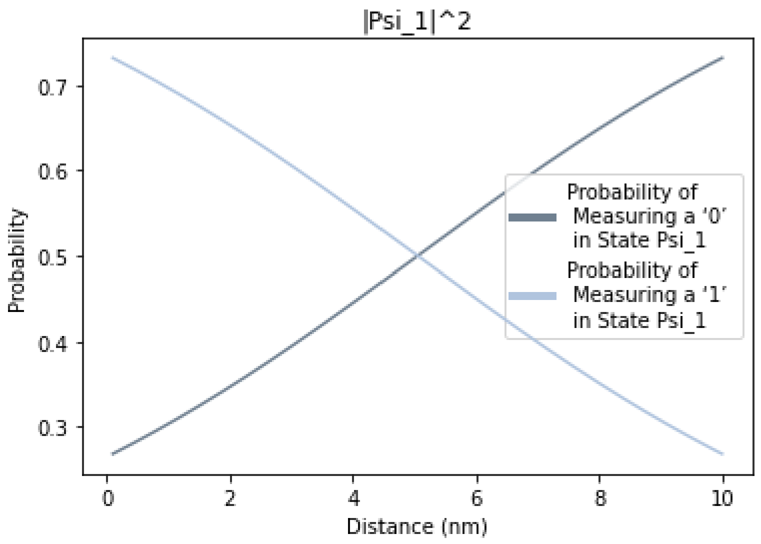
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
hbar=1.055e-34 #Js
q=1.602e-19    #C
m=0.2*9.1e-31      #kg
a=1e-10
X=a*np.linspace(1, 100, 1000)/1e-9
#t is effectively given by noting the mass that is given in the statement of the problem
t=(hbar*hbar)/(2*m*a*a)/q
E=np.linspace(-10,10,1000)
#Plotting for this choice of E clearly shows the symmetry of the eigenvalues
#H=[[E,t],[t*,-E]]
#Assume t is real
#lambda = eigenvalues
lambda0=-np.sqrt(E**2+t**2)
lambda1=np.sqrt(E**2+t**2)
#eig0=[e00,e01], eig1=[e10,e11]
e00=(E-np.sqrt(E**2+t**2))/t
e10=(E+np.sqrt(E**2+t**2))/t
e01=1
e11=1
#n stands for normalized
ne00=e00/(np.sqrt(((E-np.sqrt(E**2+t**2))/t)**2+1))
ne10=e10/(np.sqrt(((E+np.sqrt(E**2+t**2))/t)**2+1))
ne01=e01/(np.sqrt(((E-np.sqrt(E**2+t**2))/t)**2+1))
ne11=e11/(np.sqrt(((E+np.sqrt(E**2+t**2))/t)**2+1))

fig=plt.figure(1)
ax=fig.add_subplot(111, label="1")
ax2=fig.add_subplot(111, label="2", frame_on=False)
ax.plot(E, lambda0, color='slategrey')
ax2.plot(E, lambda1, color='lightsteelblue')
ax.set_xlabel('Variable E (eV)')
ax.set_ylabel('(eV)')
plt.title('Lowest Two Eigenvalues as a Function of E')
custom_lines = [Line2D([0], [0], color='slategrey', lw=4),
                Line2D([0], [0], color='lightsteelblue', lw=4)]
ax.legend(custom_lines, ['eig0', 'eig1'], loc='center right')
ax2.set_yticks([])
ax2.set_xticks([])
plt.show()
print('Minimum Difference Between Two Lowest Eigenvalues in terms of t: min(eig0-eig1)=

fig=plt.figure(2)
ax=fig.add_subplot(111, label="1")
ax2=fig.add_subplot(111, label="2", frame_on=False)
ax.plot(X, ne00**2*np.ones(1000), color='slategrey')
ax2.plot(X, ne01**2*np.ones(1000), color='lightsteelblue')
ax.set_xlabel('Distance (nm)')
ax.set_ylabel('Probability')
plt.title('|Psi_0|^2')
custom_lines = [Line2D([0], [0], color='slategrey', lw=4),
                Line2D([0], [0], color='lightsteelblue', lw=4)]
ax.legend(custom_lines, ['Probability of \n Measuring a ‘0’ \n in State Psi_0', 'Probab
ax2.set_yticks([])
ax2.set_xticks([])
plt.show()
```

```
fig=plt.figure(3)
ax=fig.add_subplot(111, label="1")
ax2=fig.add_subplot(111, label="2", frame_on=False)
ax.plot(X, ne10**2*np.ones(1000), color='slategrey')
ax2.plot(X, ne11**2*np.ones(1000), color='lightsteelblue')
ax.set_xlabel('Distance (nm)')
ax.set_ylabel('Probability')
plt.title('|Psi_1|^2')
custom_lines = [Line2D([0], [0], color='slategrey', lw=4),
                Line2D([0], [0], color='lightsteelblue', lw=4)]
ax.legend(custom_lines, ['Probability of \n Measuring a ‘0’ \n in State Psi_1', 'Probab
ax2.set_yticks([])
ax2.set_xticks([])
plt.show()
print('Note that at any point the proability of measuring the wavefunction in either sta
```



```
Minimum Difference Between Two Lowest Eigenvalues in terms of t: min(eig0-eig1)=min(2*sq
rt(E**2+t**2)
```





```
Note that at any point the proability of measuring the wavefunction in either state ‘0’
or ‘1’ sums to one
```

Part B

(i) Objectives:

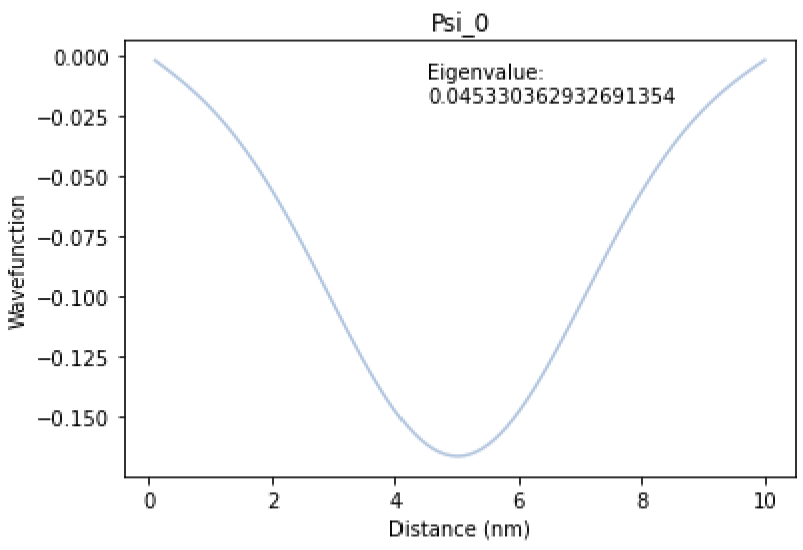
1. Solve the single qunatum dot and obtain the lowest two eigenvalues and wavefunctions.
2. Plot the two probability distributions as a function of x.
3. Plot the difference between the two eigenvalues as a function of alpha.
4. If one wants to engineer an energy splitting of about 4 meV (approximately) between these two states, what value of alpha should one choose?
5. Discuss how alpha can be engineered in a real device. What temperature ranges would these qubits be expected to operate at and why?

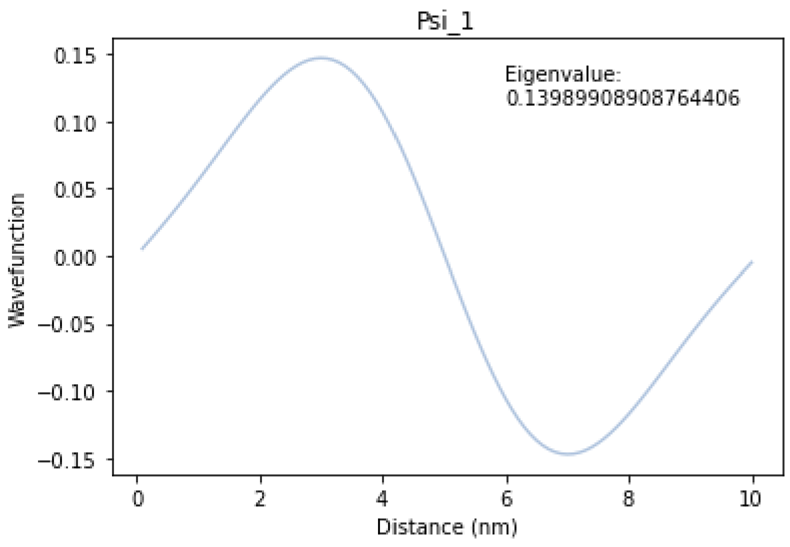
In [262…

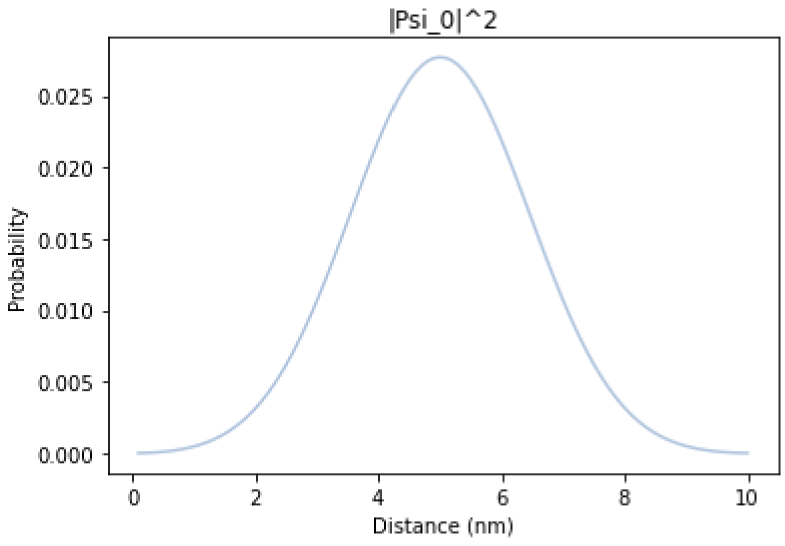
```
import numpy as np
import matplotlib.pyplot as plt
#constants
hbar=1.055e-34 #Js
q=1.602e-19    #C
m=0.2*9.1e-31      #kg
#set up grid
Np=100
#sample code has a=1e-10, but this gives probability distributions
#that integrate to 0.1 instead of 1, so a has been updated so
#the probability distributions integrate to 1 as they should
a=1e-10    #m
X=a*np.linspace(1,Np, Np)/1e-9  #nm
#Define Hamiltonian as a tridiagonal matrix
t0=(hbar*hbar)/(2*m*a*a)/q #divide by q to convert to eV
on=2.0*t0*np.ones(Np)
off=-t0*np.ones(Np-1)
#Define single quantum dot potential
n1=50
x1=n1*a/1e-9
alpha=0.0105
V=alpha*(X-x1)**2; #eV
U=np.array(V*np.ones(Np))
#Define Hamiltonian
```

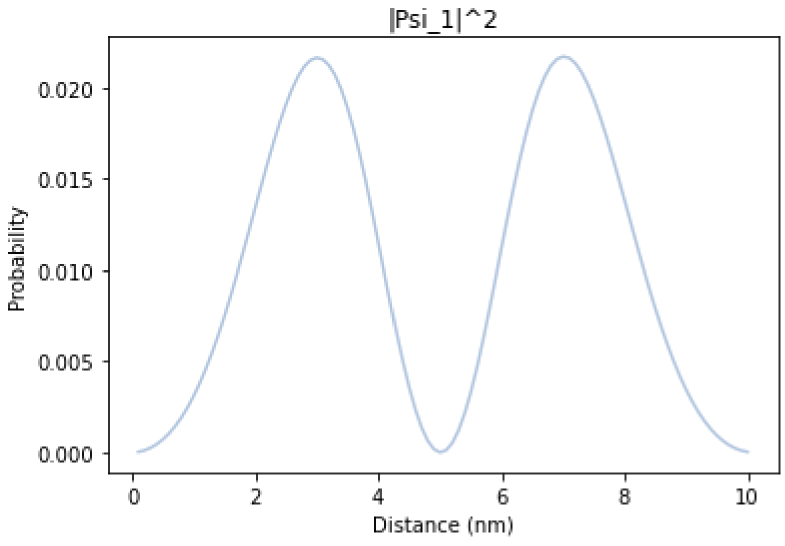
```
H=np.diag(on+U)+np.diag(off,1)+np.diag(off,-1)
#solve for eigenvalues and vectors
W,V=np.linalg.eig(H)
idx = W.argsort()[::1]
W = W[idx]
V = V[:,idx]

#display lowest two eigenvalues and wavefunctions
fig=plt.figure(4)
ax=fig.add_subplot(111, label="1")
ax.plot(X, V[:,0], color='lightsteelblue')
plt.xlabel('Distance (nm)')
plt.ylabel('Wavefunction')
plt.title('Psi_0')
ax.text(0.45, 0.95, 'Eigenvalue:\n'+str(W[0]), horizontalalignment='left', verticalalig
plt.show()
fig=plt.figure(5)
ax21=fig.add_subplot(111, label="1")
ax21.plot(X, V[:,1], color='lightsteelblue')
ax21.set_xlabel('Distance (nm)')
ax21.set_ylabel('Wavefunction')
plt.title('Psi_1')
ax21.text(0.6, 0.95, 'Eigenvalue:\n'+str(W[1]), horizontalalignment='left', verticalali
plt.show()

#plot two probability distributions as a function of X
probPsi0=np.multiply(V[:,0],V[:,0])
probPsi1=np.multiply(V[:,1],V[:,1])
plt.figure(6)
plt.plot(X, probPsi0, color='lightsteelblue')
plt.xlabel('Distance (nm)')
plt.ylabel('Probability')
plt.title('|Psi_0|^2')
plt.show()
plt.figure(7)
plt.plot(X, probPsi1, color='lightsteelblue')
plt.xlabel('Distance (nm)')
plt.ylabel('Probability')
plt.title('|Psi_1|^2')
plt.show()

integral=np.sum(probPsi0)
integral2=np.sum(probPsi1)
print('Note that both probability distributions integrate to 1 and are nonnegative. Int
      ' Integral |Psi_1|^2:'+str(np.round(integral2)))

#Define single quantum dot potential
n1=50
#Vary alpha
alpha=np.linspace(0.00001,0.0003,1000)
a=1e-8    #m
X=a*np.linspace(1,Np, Np)/1e-9  #nm
x1=n1*a/1e-9
t0=(hbar*hbar)/(2*m*a*a)/q #divide by q to convert to eV
on=2.0*t0*np.ones(Np)
off=-t0*np.ones(Np-1)
#Get energy gap values for varying alpha
Eg=[]
ac1=[]
ac2=[]
```

```
for i in alpha:
  V=i*(X-x1)**2; #eV
  U=np.array(V*np.ones(Np))
#Define Hamiltonian
  H=np.diag(on+U)+np.diag(off,1)+np.diag(off,-1)
#solve for eigenvalues and vectors
  W,e=np.linalg.eig(H)
  idx = W.argsort()[::1]
  W = W[idx]
  e = e[:,idx]
  Energy_gap=np.abs(W[0]-W[1])
  if np.round(Energy_gap,3) == 0.004:
        ac1.append(i)
  Eg.append(Energy_gap)
k=np.array(0.004*np.ones(1000))
plt.figure(7)
plt.plot(alpha, Eg, color='slategrey')
plt.plot(alpha, k, color='salmon')
plt.scatter(np.sum(ac1)/len(ac1),0.004)
plt.title('Difference Between the Two Eigenvalues as a Function of alpha')
plt.xlabel('Variable alpha (eV)')
plt.ylabel('Energy Gap (eV)')
plt.show()
print('alpha~'+str(np.round(np.sum(ac1)/len(ac1),6))+' creates an energy splitting of ~
print('The range of values used to determine this alpha was between ' + str((np.min(ac1
     str((np.max(ac1)))
k=8.617e-6
print('Using the well-known relation, E=kT with E='+str(np.round(np.min(ac1),6))+' and
     ', k=8.617e-6 eV*K^-1, T=|E|/k implies T='+str(np.abs(np.round(np.min(ac1)/k,2)))+
        +str(np.abs(np.round(np.min(ac1)/k,2)))+'-'+str(np.abs(np.round(np.max(ac1)/k,2
```

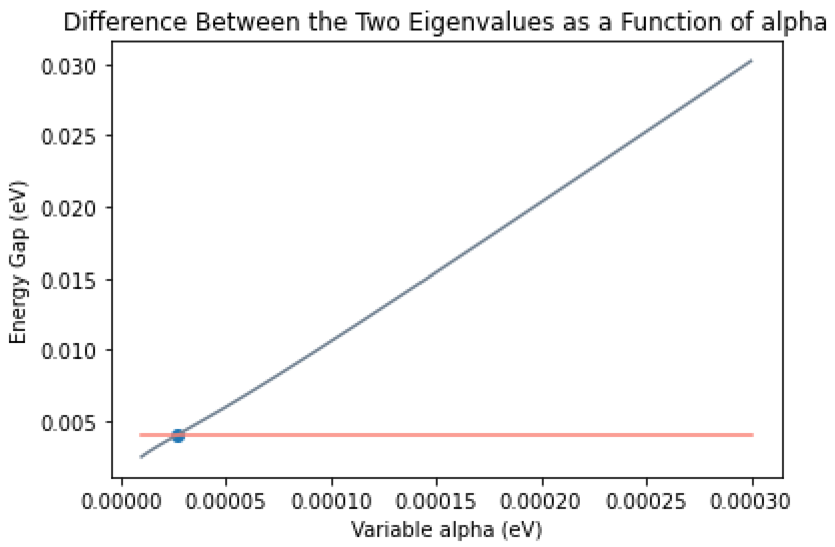








```
Note that both probability distributions integrate to 1 and are nonnegative. Integral |P
si_0|^2:1.0 Integral |Psi_1|^2:1.0
```



```
alpha~2.7e-05 creates an energy splitting of ~4meV
The range of values used to determine this alpha was between 2.1031031031031032e-05 and
3.2642642642642636e-05
Using the well-known relation, E=kT with E=2.1e-05 and 3.3e-05, k=8.617e-6 eV*K^-1, T=|E
|/k implies T=2.44 and 3.79K, respectively, for an operating range of 2.44-3.79K
```

To engineer alpha in an actual device it should be noted that a quantum dot in this simulation is essenitally a harmonic oscillator. Therefore, the potential V is of the form 1/2mw^2x^2, so α=mw^2 and it is known that E=(n+1/2)$\hbar$w. Therefore, w may be found from the energy and α may be engineered by adjusting the distances between the gates and the thicknesses of the SiO$_2$ and Si layers to engineer this desired energy. One should choose a value of α that creates a measurable difference between the lowest two eigenvalues but not one that creates a difference so large that an electron is excluded from moving between these two energy levels.

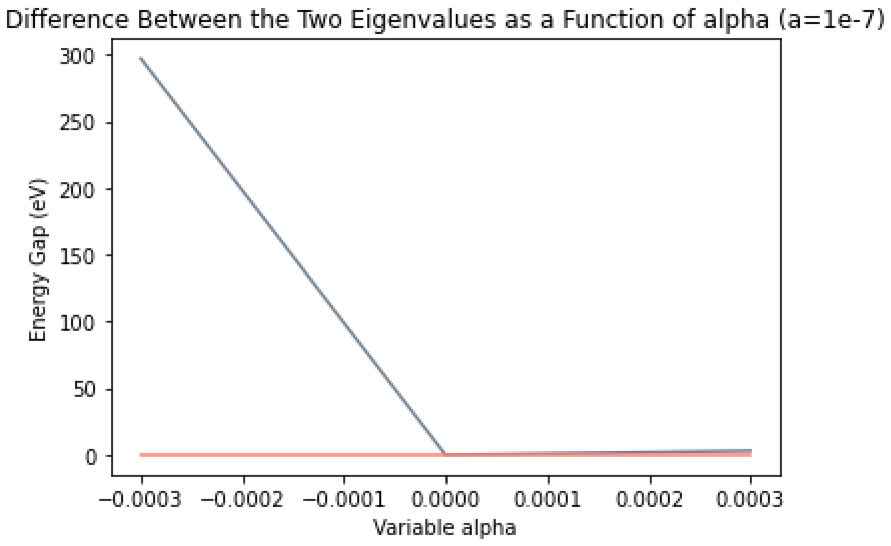
(ii) Objectives:

How was it ensured in (i) that the results do not suffer from various numerical artifacts, such as small simulation domain (artificial confinement), inaccurate meshing, and any other issues? Provide evidence of the tests that you performed. It was checked that the probability distributions integrated to 1 and were nonnnegative. A homogenous mesh was used because the potential was smooth and could be easily approximated linearly. There was also little variation in the potential over the meshing distance because V'=2α(X-x_1) with the distance on the order of nm and α<1 implies ΔV is of order less than -9. Artificial confinement was acounted for by modifying the value of a and adjusting the range of alpha to avoid numerical artefacts from the simulation runninng into the border of its domain. These adjustments were made after noting the unphysical tail of the energy gap plot as a function of alpha when alpha became positive for a=1e-9 and that the energy gap as plotted in this experiment never crossed 0.004 eV otherwise if a was not equal to 1e-8. Previous plots of the energy gap as a function of α before these modifications are below.

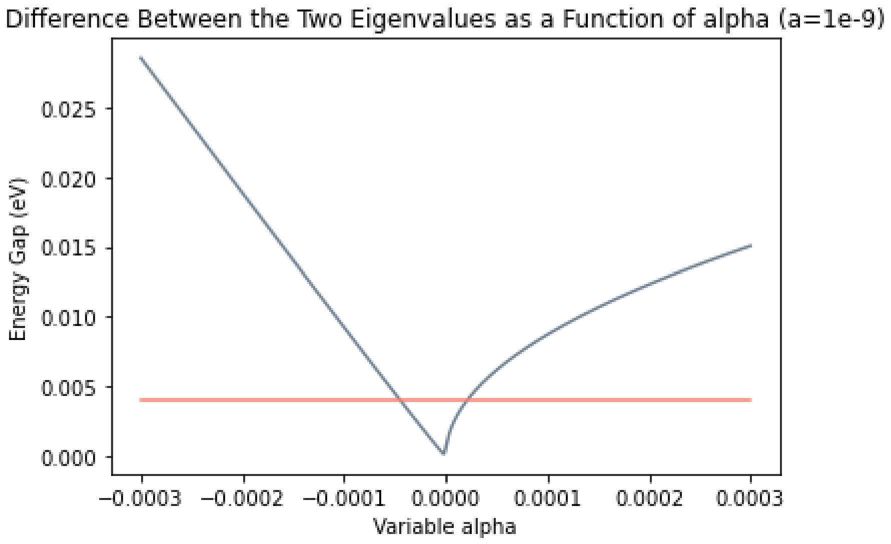
In [146…
```python
n1=50
#Vary alpha
alpha=np.linspace(-0.0003,0.0003,1000)
a=1e-7    #m
X=a*np.linspace(1,Np, Np)/1e-9  #nm
x1=n1*a/1e-9
t0=(hbar*hbar)/(2*m*a*a)/q #divide by q to convert to eV
```

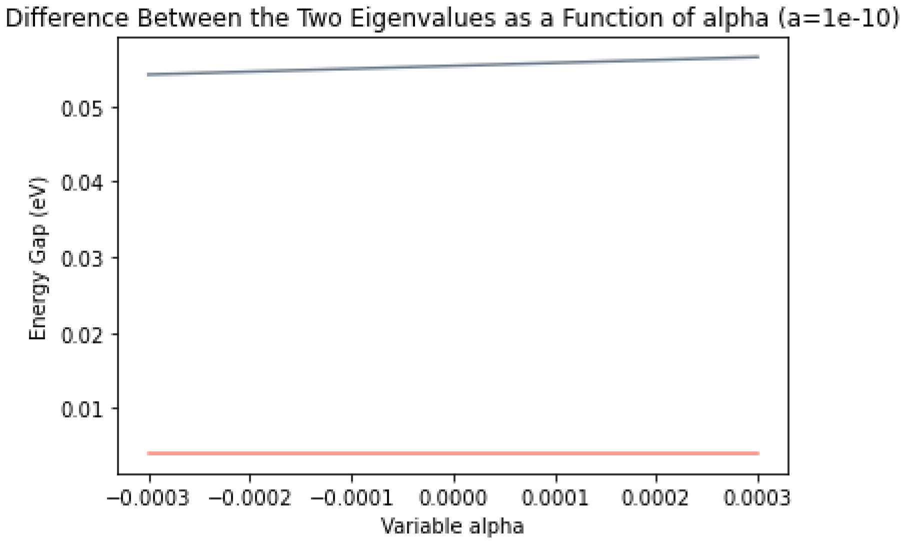
```
on=2.0*t0*np.ones(Np)
off=-t0*np.ones(Np-1)
#Get energy gap values for varying alpha
Eg=[]
for i in alpha:
  V=i*(X-x1)**2; #eV
  U=np.array(V*np.ones(Np))
#Define Hamiltonian
  H=np.diag(on+U)+np.diag(off,1)+np.diag(off,-1)
#solve for eigenvalues and vectors
  W,e=np.linalg.eig(H)
  idx = W.argsort()[::1]
  W = W[idx]
  e = e[:,idx]
  Energy_gap=np.abs(W[0]-W[1])
  Eg.append(Energy_gap)
k=np.array(0.004*np.ones(1000))
plt.figure(7)
plt.plot(alpha, Eg, color='slategrey')
plt.plot(alpha, k, color='salmon')
plt.title('Difference Between the Two Eigenvalues as a Function of alpha (a=1e-7)')
plt.xlabel('Variable alpha (eV)')
plt.ylabel('Energy Gap (eV)')
plt.show()
print('When run with code to find when the energy gap crosses 0.004eV a nan result is p

n1=50
#Vary alpha
alpha=np.linspace(-0.0003,0.0003,1000)
a=1e-9    #m
X=a*np.linspace(1,Np, Np)/1e-9  #nm
x1=n1*a/1e-9
t0=(hbar*hbar)/(2*m*a*a)/q #divide by q to convert to eV
on=2.0*t0*np.ones(Np)
off=-t0*np.ones(Np-1)
#Get energy gap values for varying alpha
Eg=[]
for i in alpha:
  V=i*(X-x1)**2; #eV
  U=np.array(V*np.ones(Np))
#Define Hamiltonian
  H=np.diag(on+U)+np.diag(off,1)+np.diag(off,-1)
#solve for eigenvalues and vectors
  W,e=np.linalg.eig(H)
  idx = W.argsort()[::1]
  W = W[idx]
  e = e[:,idx]
  Energy_gap=np.abs(W[0]-W[1])
  Eg.append(Energy_gap)
k=np.array(0.004*np.ones(1000))
plt.figure(7)
plt.plot(alpha, Eg, color='slategrey')
plt.plot(alpha, k, color='salmon')
plt.title('Difference Between the Two Eigenvalues as a Function of alpha (a=1e-9)')
plt.xlabel('Variable alpha (eV)')
plt.ylabel('Energy Gap (eV)')
plt.show()

n1=50
#Vary alpha
```

```
alpha=np.linspace(-0.0003,0.0003,1000)
a=1e-10   #m
X=a*np.linspace(1,Np, Np)/1e-9  #nm
x1=n1*a/1e-9
t0=(hbar*hbar)/(2*m*a*a)/q #divide by q to convert to eV
on=2.0*t0*np.ones(Np)
off=-t0*np.ones(Np-1)
#Get energy gap values for varying alpha
Eg=[]
for i in alpha:
  V=i*(X-x1)**2; #eV
  U=np.array(V*np.ones(Np))
#Define Hamiltonian
  H=np.diag(on+U)+np.diag(off,1)+np.diag(off,-1)
#solve for eigenvalues and vectors
  W,e=np.linalg.eig(H)
  idx = W.argsort()[::1]
  W = W[idx]
  e = e[:,idx]
  Energy_gap=np.abs(W[0]-W[1])
  Eg.append(Energy_gap)
k=np.array(0.004*np.ones(1000))
plt.figure(7)
plt.plot(alpha, Eg, color='slategrey')
plt.plot(alpha, k, color='salmon')
plt.title('Difference Between the Two Eigenvalues as a Function of alpha (a=1e-10)')
plt.xlabel('Variable alpha (eV)')
plt.ylabel('Energy Gap (eV)')
plt.show()
```



```
When run with code to find when the energy gap crosses 0.004eV a nan result is produced
because the energy gap does not
 in this plot
```





(iii) Objectives:

Why are quantum dots often called ‘artificial atoms’? Compare the length scale of the wavefunction to the Bohr radii of the ground state of a Hydrogen atom (by choosing a similar measure). If the atoms in a silicon crystal along a 1D chain are spaced by 0.543 nm, over how many atoms does a QD (ground state) wavefunction span for energy splitting of about 4 meV as in B(i)? Quantum dots are often called artifical atoms because they have discrete electronic energy levels like atoms. The Bohr radius is ~0.053 nm this is about 188 times smaller than the 10nm length scale of the ground state wavefunction Psi_0 from 0 to 0. For atoms in a silicon crystal along a 1D chain spaced by 0.543 nm, a single quantum dot would span 19 atoms.

Part C

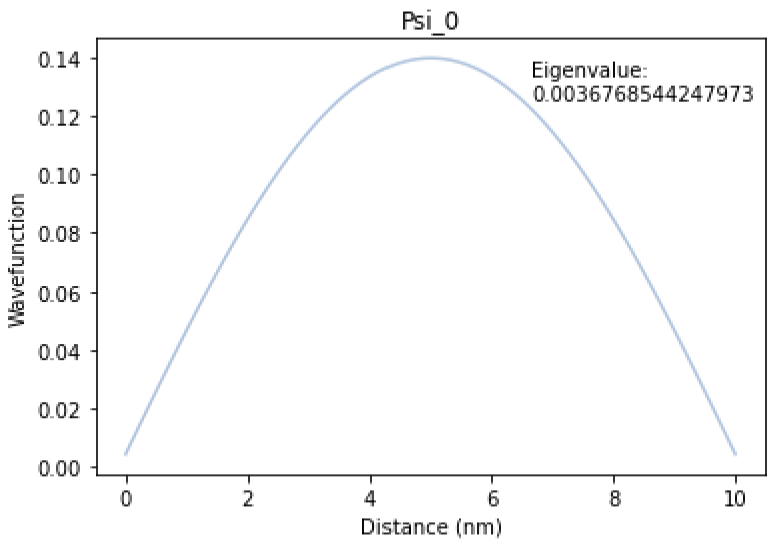
(i) Objectives:

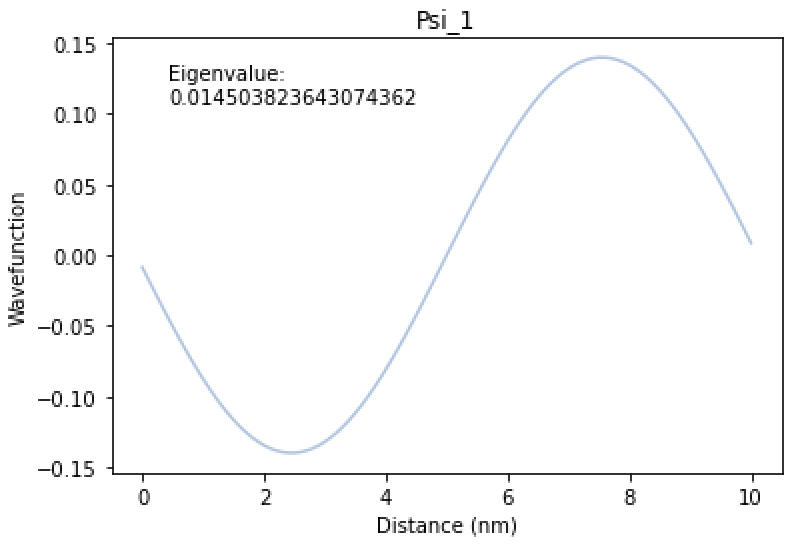
1. Plot the two lowest energy wavefunctions and their corresponding probability distributions as function of x for a double quantum dot and for choice of alpha that produces about 4 meV level splitting for a single quantum dot and separation distance R between the dots, chosen such that it covers two regimes of weak and strong interaction between the dots.

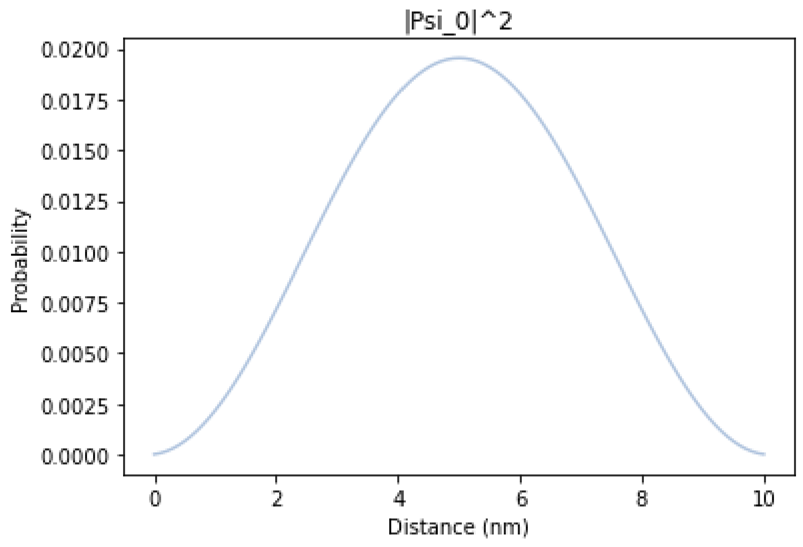
2. What differences are observed between the lowest two wavefunctions?

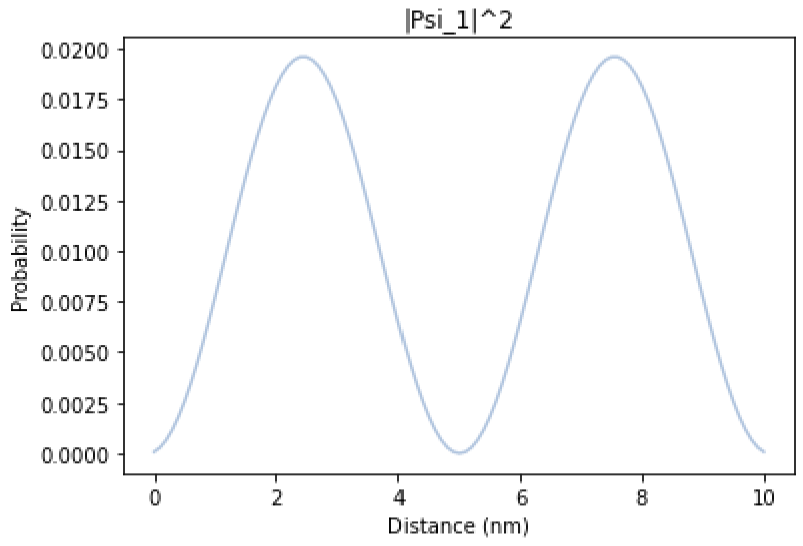
In [151…

```
import numpy as np
import matplotlib.pyplot as plt
#constants
hbar=1.055e-34 #Js
q=1.602e-19    #C
m=9.1e-31      #kg
#grid
Np=100
a=1e-10    #m
X=a*np.linspace(0, Np, Np+1)/1e-9  #nm
#Define Hamiltonian as a tridiagonal matrix
t0=(hbar*hbar)/(2*m*a*a)/q #divide by q to convert to eV
on=2.0*t0*np.ones(Np+1)
off=-t0*np.ones(Np)
#Define double quantum dot potential
n1=25
n2=75
x1=n1*a/1e-9
x2=n2*a/1e-9
alpha=2.7e-5
V=[]
for i in X:
  v=min(alpha*(i-x1)**2, alpha*(i-x2)**2); #eV
  V.append(v)
U=np.array(V*np.ones(Np+1))
#Define Hamiltonian
H=np.diag(on+U)+np.diag(off,1)+np.diag(off,-1)
#solve for eigenvalues and vectors
W,V=np.linalg.eig(H)
idx = W.argsort()[::1]
W = W[idx]
V = V[:,idx]
#calculate probablity
Psi0=np.multiply(V[:,0],V[:,0])
Psi1=np.multiply(V[:,1],V[:,1])
#display lowest two eigenvalues and wavefunctions
fig=plt.figure(4)
ax=fig.add_subplot(111, label="1")
ax.plot(X, V[:,0], color='lightsteelblue')
plt.xlabel('Distance (nm)')
plt.ylabel('Wavefunction')
plt.title('Psi_0')
ax.text(0.65, 0.95, 'Eigenvalue:\n'+str(W[0]), horizontalalignment='left', verticalalig
plt.show()
fig=plt.figure(5)
ax21=fig.add_subplot(111, label="1")
ax21.plot(X, V[:,1], color='lightsteelblue')
ax21.set_xlabel('Distance (nm)')
ax21.set_ylabel('Wavefunction')
plt.title('Psi_1')
ax21.text(0.1, 0.95, 'Eigenvalue:\n'+str(W[1]), horizontalalignment='left', verticalali
plt.show()
#corresponding probabilities
plt.figure(10)
plt.plot(X, Psi0, color='lightsteelblue')
plt.xlabel('Distance (nm)')
plt.ylabel('Probability')
plt.title('|Psi_0|^2')
plt.show()
plt.figure(11)
plt.plot(X, Psi1, color='lightsteelblue')
plt.xlabel('Distance (nm)')
plt.ylabel('Probability')
plt.title('|Psi_1|^2')
plt.show()
integral=np.sum(Psi0)
integral2=np.sum(Psi1)
print('Note that both probability distributions integrate to 1 and are nonnegative. Int
      ' Integral |Psi_1|^2:\n'+str(np.round(integral2)))
```









```
Note that both probability distributions integrate to 1 and are nonnegative. Integral |P
si_0|^2:1.0 Integral |Psi_1|^2:
1.0
```

The wavefunction Psi_0 is even while the wavefunction Psi_1 is odd. They both have the same maximum value, but Psi_0 only takes positive values while Psi_1 also takes negative values.

(ii) Objectives:

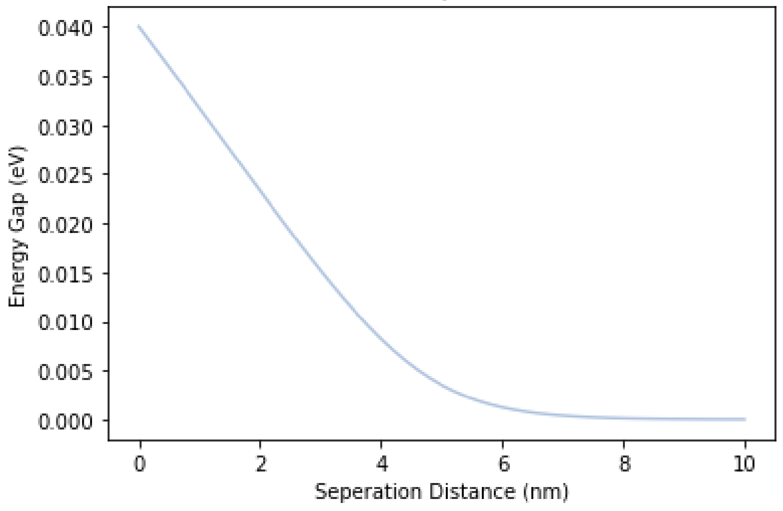
1. Plot the energy difference between the lowest two states of the double quantum dot as a function of the centre-to-centre dot separation.
2. Which Hamiltonian parameter of the 2 × 2 matrix from Part A is modified by engineering the dot separation? Physically, in this system, what does this parameter represent, and would this also be present in a classical system?
3. Is a large or small dot-to-dot separation desirable? Is there an optimal separation.

In [164…

```
import numpy as np
import matplotlib.pyplot as plt
#constants
hbar=1.055e-34 #Js
q=1.602e-19    #C
m=9.1e-31      #kg
```

```
#grid
Np=100
a=1e-10    #m
X=a*np.linspace(0, Np, Np+1)/1e-9  #nm
#Define Hamiltonian as a tridiagonal matrix
t0=(hbar*hbar)/(2*m*a*a)/q #divide by q to convert to eV
on=2.0*t0*np.ones(Np+1)
off=-t0*np.ones(Np)
#Define double quantum dot potential
n1=50
n2=50
x1=n1*a/1e-9
x2=n2*a/1e-9
alpha=0.0105
R=[]
Rn=[]
Eg=[]
while n1 >= 0:
  x1=n1*a/1e-9
  x2=n2*a/1e-9
  r=x2-x1
  rn=n2-n1
  R.append(r)
  Rn.append(rn)
  V=[]
  for i in X:
    v=min(alpha*(i-x1)**2, alpha*(i-x2)**2); #eV
    V.append(v)
  U=np.array(V*np.ones(Np+1))
#Define Hamiltonian
  H=np.diag(on+U)+np.diag(off,1)+np.diag(off,-1)
#solve for eigenvalues and vectors
  W,V=np.linalg.eig(H)
  idx = W.argsort()[::1]
  W = W[idx]
  V = V[:,idx]
  Energy_gap=np.abs(W[0]-W[1])
  Eg.append(Energy_gap)
  n1-=1
  n2+=1
plt.figure(12)
plt.plot(np.array(Rn)/10, Eg, color='lightsteelblue')
plt.xlabel('Seperation Distance (nm)')
plt.ylabel('Energy Gap (eV)')
plt.title('Energy Difference Between Lowest Two States of DQD as a Function of Centre-t
plt.show()
```

Energy Difference Between Lowest Two States of DQD as a Function of Centre-to-Centre Dot Separation



Engineering the dot seperation modifies E from the 2x2 Hamiltonian matrix of A. Modifying the dot seperation is a way of modifying the potential which in turn modifies the value of E. This parameter would be present in a classical system governed by some form of potential well, but it may be harder to engineer this specific double harmonic potential well classically than it is in this case quantum mechanically. A quantum dot seperation distance of about 2.5 nm would be optimal because the plot quickly decays to 0 eV once a seperation distance of 5 nm is exceded. There is an approximately linear trend from the value of the energy gap at 0 nm seperation distance to when the value of the energy gap is essentailly 0 eV at 5 nm. This suggests a quantum dot seperation of about 2.5 nm would be optimal because any closer and the potential wells would essentially be on top of each other, defeating the purpose of allowing an electron to be in the ground state of one well or the other with some tunneling probability.

(iii) Objectives:

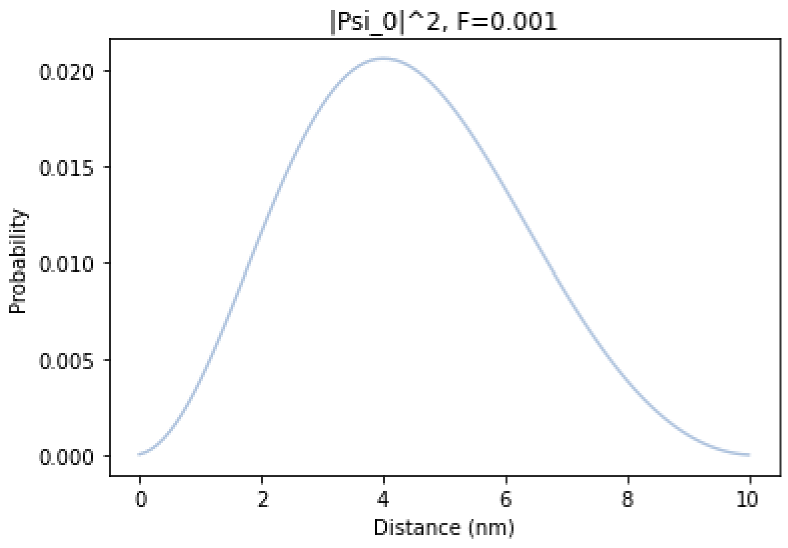
Voltages applied to the two gates G1 and G2 are used to create an electric field in the x direction. This controls the localization of the electron between the two dots, and is called a “detuning” bias. Model this by adding an extra potential energy term for a uniform electric F field V(x) = Fx to the double quantum dot Hamiltonian.

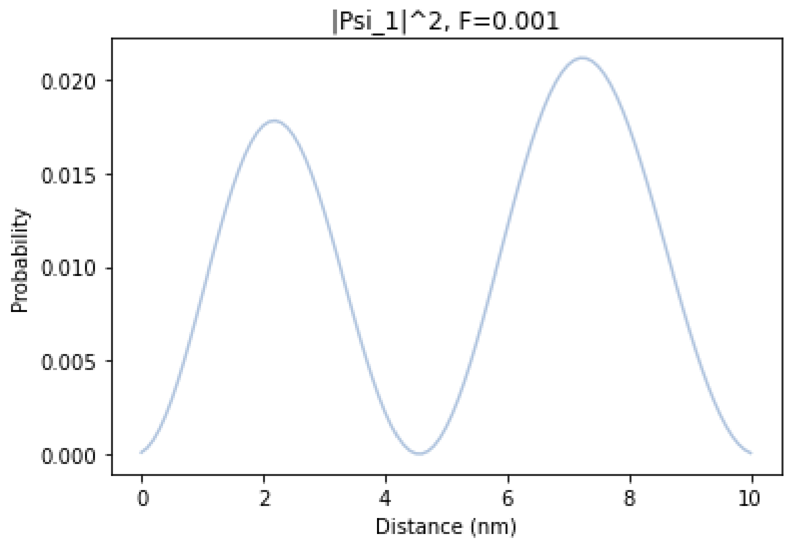
1. Plot the lowest two wavefunction probabilities as a function of x when there is small detuning bias.
2. What happens when the sign of F is changed? Plot for suitable choices of F in the order of meV from positive to negative values, the lowest two eigenvalues, and their differences.
3. Can this be related to the analytic solution from Part A?
4. Which parameter of the 2 × 2 Hamiltonian is controlled by this detuning bias?
5. From simulations, over what range of F can the qubit can be operated?

In [160…

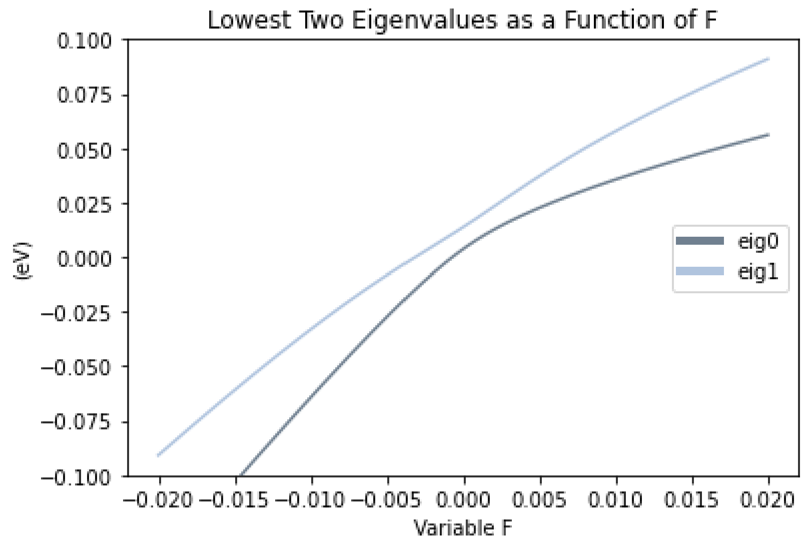
```
import numpy as np
import matplotlib.pyplot as plt
#constants
hbar=1.055e-34 #Js
q=1.602e-19    #C
m=9.1e-31      #kg
```

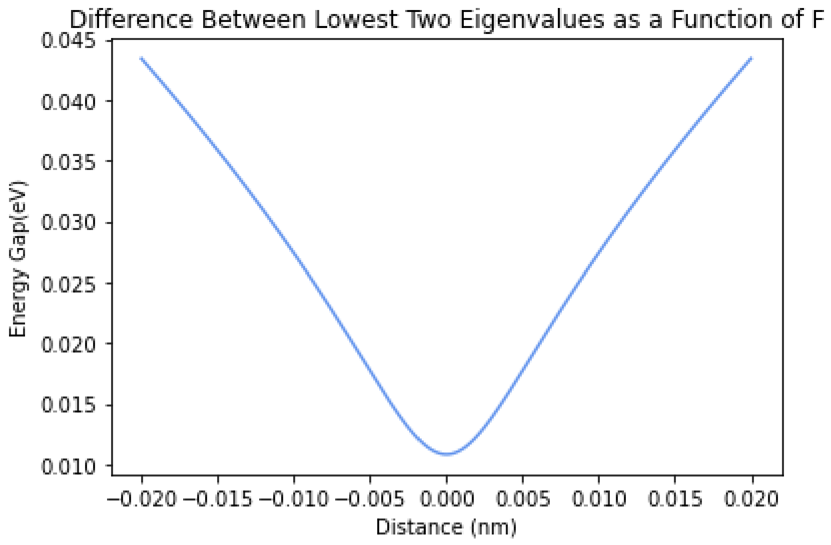
```
#grid
Np=100
a=1e-10    #m
X=a*np.linspace(0, Np, Np+1)/1e-9  #nm
#Define Hamiltonian as a tridiagonal matrix
t0=(hbar*hbar)/(2*m*a*a)/q #divide by q to convert to eV
on=2.0*t0*np.ones(Np+1)
off=-t0*np.ones(Np)
#Define double quantum dot potential
n1=25
n2=75
x1=n1*a/1e-9
x2=n2*a/1e-9
alpha=2.7e-5
F=0.001
V=[]
for i in X:
  v=min(alpha*(i-x1)**2, alpha*(i-x2)**2)+F*i; #eV
  V.append(v)
U=np.array(V*np.ones(Np+1))
#Define Hamiltonian
H=np.diag(on+U)+np.diag(off,1)+np.diag(off,-1)
#solve for eigenvalues and vectors
W,V=np.linalg.eig(H)
idx = W.argsort()[::1]
W = W[idx]
V = V[:,idx]
#calculate probablity
Psi0=np.multiply(V[:,0],V[:,0])
Psi1=np.multiply(V[:,1],V[:,1])

plt.figure(13)
plt.plot(X, Psi0, color='lightsteelblue')
plt.xlabel('Distance (nm)')
plt.ylabel('Probability')
plt.title('|Psi_0|^2, F=0.001')
plt.show()

plt.figure(14)
plt.plot(X, Psi1, color='lightsteelblue')
plt.xlabel('Distance (nm)')
plt.ylabel('Probability')
plt.title('|Psi_1|^2, F=0.001')
plt.show()

integral=np.sum(probPsi0)
integral2=np.sum(probPsi1)
print('Note that both probability distributions integrate to 1 and are nonnegative. Int
      ' Integral |Psi_1|^2:\n'+str(np.round(integral2)))

F=np.linspace(-0.02, 0.02,1000)
Np=100
X=a*np.linspace(0, Np, Np+1)/1e-9  #nm
t0=(hbar*hbar)/(2*m*a*a)/q #divide by q to convert to eV
on=2.0*t0*np.ones(Np+1)
off=-t0*np.ones(Np)
eig0=[]
eig1=[]
gap=[]
for i in F:
  V=[]
  for o in X:
    v=min(alpha*(o-x1)**2, alpha*(o-x2)**2)+i*o; #eV
    V.append(v)
  U=np.array(V*np.ones(Np+1))
  #Define Hamiltonian
  H=np.diag(on+U)+np.diag(off,1)+np.diag(off,-1)
  #solve for eigenvalues and vectors
  W,V=np.linalg.eig(H)
  idx = W.argsort()[::1]
  W = W[idx]
  eig0.append(W[0])
  eig1.append(W[1])
  gap.append(W[1]-W[0])
fig=plt.figure(15)
ax=fig.add_subplot(111, label="21")
ax2=fig.add_subplot(111, label="22", frame_on=False)
ax.plot(F, eig0, color='slategrey')
ax2.plot(F, eig1, color='lightsteelblue')
ax.set_xlabel('Variable F (eV)')
ax.set_ylabel('Energy Gap(eV)')
ax.set_ylim([-0.1,0.1])
plt.title('Lowest Two Eigenvalues as a Function of F')
custom_lines = [Line2D([0], [0], color='slategrey', lw=4),
                Line2D([0], [0], color='lightsteelblue', lw=4)]
ax.legend(custom_lines, ['eig0', 'eig1', 'gap'], loc='center right')
ax2.set_xticks([])
ax2.set_yticks([])
plt.show()

plt.figure(14)
plt.plot(F, gap, color='cornflowerblue')
plt.xlabel('Variable F (eV))')
plt.ylabel('Energy Gap(eV)')
plt.title('Difference Between Lowest Two Eigenvalues as a Function of F')
plt.show()
```





```
Note that both probability distributions integrate to 1 and are nonnegative. Integral |P
si_0|^2:1.0 Integral |Psi_1|^2:
1.0
```





When the sign of F is changed, the value of the energy gap is uneffected because the energy gap as a function of F is symmetric. Again, the detuning bias is a modification made to the potential and the potential corresponds to the parameter E of the Hamiltonian in part A. It is related to the analytic solution of part A in the sense that the detuning bias serves to modify the seperation between the lowest energy levels. For example, the energy eigenvalues plotted in part A intersect at two points, but the detuning bias applied here causes them to never intersect. From the simulation, the qubits should be operated with a detuning bias between -0.0075 and 0.0075 eV. The plots utilize a range of F from -20 to 20 meV to display the interesting feature that the eigenvalues never intersect and to ensure that any numerical artifacts from a small simulation domain do not effect the range of F values desired to be studied, that is, -10 meV to 10 meV. For values of F greater in magnitude than 0.0075 eV, the difference between the energy eigenvalues begins to no longer be parabolic, indicating possible quantum decoherence.

Part D

(i) Objectives:

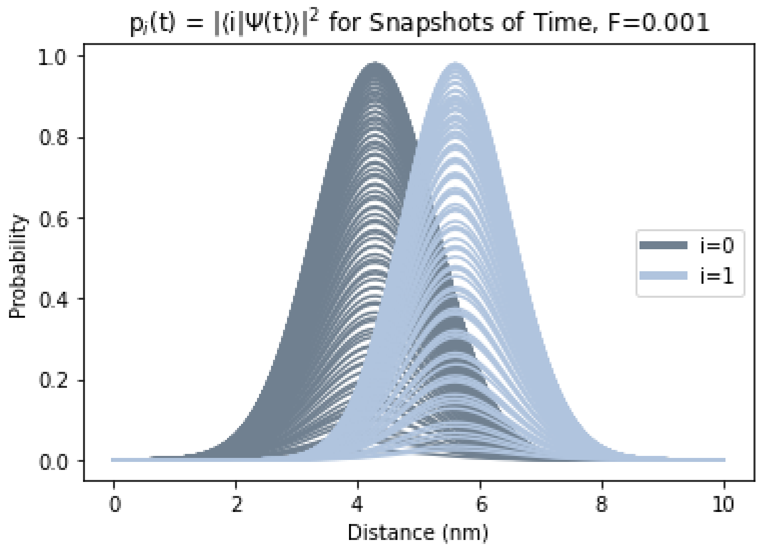
In practice, charge qubits are controlled by a time varying detuning potential that periodically moves the electron between the left and right dots. This creates an oscillation between the two qubit levels called a “Rabi oscillation”. Model this with a numerical solution of the time dependent Schrodinger equation on a double quantum dot subjected to a time varying potential V(x,t) = F x cos(wt), where F is a chosen electric field value. Assume the system is in the ground state at the start and resonantly driven with w = w_r = (E_1-E_0)/ℏ, the energy difference between the two lowest levels without any detuning bias. Let the two lowest energy states at t = 0 be the ‘0’ and the ‘1’ state of the qubit. Simulate time evolution at the qubit resonant frequency, for time between t = 0 and $t_{\max}$, chosen to cover at least a full period of oscillation, with suitable timestep Δt. During simulation calculate and store projections of the wavefunction on |0⟩ and |1⟩ states as changing in time i.e. for each time step calculate p_i(t) = $|\langle i|\Psi(t)\rangle|^2$ for i $\in \{0, 1\}$ and store them in some array. Plot p_0 and p_1 vs. time together in one figure for a fixed value of F. Discuss findings from exploring the impact of various parameters. Discuss what steps were taken to ensure convergence.

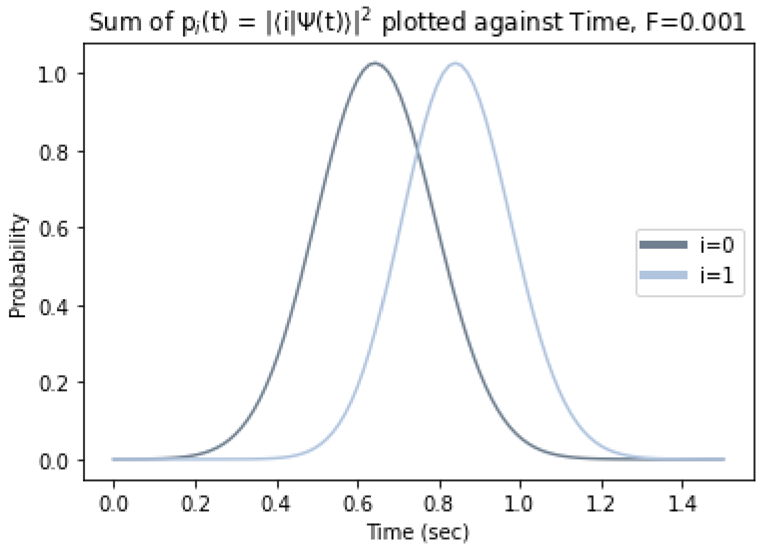
In [266…

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
#constants
hbar=1.055e-34 #Js
q=1.602e-19    #C
m=9.1e-31      #kg
#grid
Np=100
a=1e-10   #m
X=a*np.linspace(0, Np, Np+1)/1e-9  #nm
#Define Hamiltonian as a tridiagonal matrix
t0=(hbar*hbar)/(2*m*a*a)/q #divide by q to convert to eV
on=2.0*t0*np.ones(Np+1)
off=-t0*np.ones(Np)
#Define double quantum dot potential
n1=25
n2=75
x1=n1*a/1e-9
x2=n2*a/1e-9
alpha=2.7e-5
V=[]
```

```
F=0.001
for i in X:
  v=min(alpha*(i-x1)**2, alpha*(i-x2)**2)+F*i; #eV
  V.append(v)
U=np.array(V*np.ones(Np+1))
#Define Hamiltonian
H=np.diag(on+U)+np.diag(off,1)+np.diag(off,-1)
#solve for eigenvalues and vectors
W,e=np.linalg.eig(H)
idx = W.argsort()[::1]
W = W[idx]
e = e[:,idx]
#calculate probablity
psi=(e[:,0])
w=(W[0]-W[1])/hbar
p0plot1=[]
p1plot1=[]
p0plot2=[]
p1plot2=[]
t=np.linspace(0,1.5,101)
for o in t:
    V_new=[]
    for i in X:
        v=min(alpha*(i-x1)**2, alpha*(i-x2)**2)+F*i*np.cos(w*o); #eV
        V_new.append(v)
    A=np.diag(1+1j*o*(on+V_new))+np.diag(off,-1)+np.diag(off,1)
    B=np.diag(1-1j*(on+U)*psi)+np.diag(off,-1)+np.diag(off,1)
    A=A.astype(complex)
    B=B.astype(complex)
    psi_new=np.linalg.solve(A,B)
    Psi0=np.multiply(e[:,0],psi_new[:,0])
    Psi0=np.abs((Psi0))
    #Psi0=np.abs(Psi0)
    Psi1=np.multiply(e[:,1],psi_new[:,1])
    Psi1=np.abs((Psi1))
    #Divide by 900 to renormalize
    p0plot1.append(Psi0/900)
    p1plot1.append(Psi1/900)
    p0plot2.append(np.sum(Psi0/900))
    p1plot2.append(np.sum(Psi1/900))
    U=V_new
    psi=psi_new

fig=plt.figure(16)
ax=fig.add_subplot(111, label="21")
ax2=fig.add_subplot(111, label="22", frame_on=False)
ax.plot(X, p0plot1, color='slategrey')
ax2.plot(X, p1plot1, color='lightsteelblue')
ax.set_xlabel('Distance (nm)')
ax.set_ylabel('Probability')
plt.title('p$_i$(t) = |⟨i|Ψ(t)⟩|$^2$ for Snapshots of Time, F=0.001')
custom_lines = [Line2D([0], [0], color='slategrey', lw=4),
                Line2D([0], [0], color='lightsteelblue', lw=4)]
ax.legend(custom_lines, ['i=0', 'i=1'], loc='center right')
ax2.set_yticks([])
ax2.set_xticks([])
plt.show()

fig=plt.figure(15)
ax=fig.add_subplot(111, label="21")
```

```
ax2=fig.add_subplot(111, label="22", frame_on=False)
#Divide by 60 to renormalize
ax.plot(t, np.array(p0plot2)/60, color='slategrey')
ax2.plot(t, np.array(p1plot2)/60, color='lightsteelblue')
ax.set_xlabel('Time (sec)')
ax.set_ylabel('Probability')
#ax.yaxis.set_ticks([0,0.25,0.5,0.75,1])
plt.title('Sum of p$_i$(t) = |⟨i|Ψ(t)⟩|$^2$ plotted against Time, F=0.001')
custom_lines = [Line2D([0], [0], color='slategrey', lw=4),
                Line2D([0], [0], color='lightsteelblue', lw=4)]
ax.legend(custom_lines, ['i=0', 'i=1'], loc='center right')
ax2.set_yticks([])
ax2.set_xticks([])
plt.show()
```





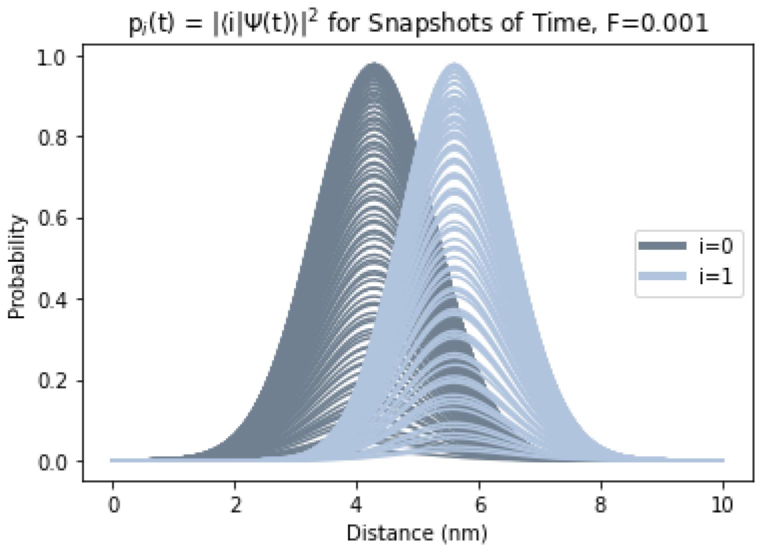
Adjusting F makes the transitions more are less continuous (i.e. the layering of the $p_i(t) = |\langle i|\Psi(t)\rangle|^2$ for Snapshots of Time plot is more or less fine. Changing α however seemed to have little or no effect. For longer time scales, the wavefunctions both seem to die out after 1.5 seconds, suggesting quantum decoherence. Print statements were used to ensure convergence and that all values of arrays were updating as they should then removed for readability.

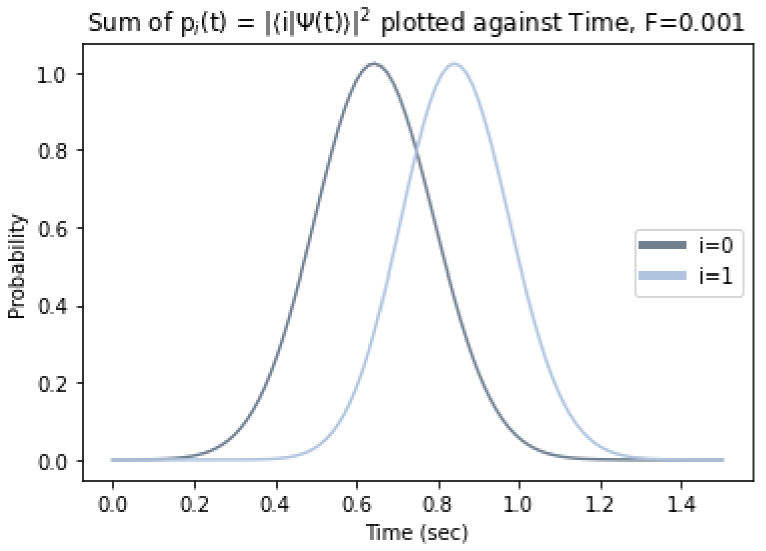
(ii) Objective:

The resonant transition between the states occurs only at or close to the resonant frequency. When one moves away from w$_r$ the transition becomes less and less probable. Repeat the simulation of (i) from part D for a chosen value of F, but now for the frequency w = 0.95w$_r$ and see how the oscillations differ in comparison to w = w$_r$ case. Discuss findings.

In [264…

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
#constants
hbar=1.055e-34 #Js
q=1.602e-19    #C
m=9.1e-31      #kg
#grid
Np=100
a=1e-10   #m
X=a*np.linspace(0, Np, Np+1)/1e-9  #nm
#Define Hamiltonian as a tridiagonal matrix
t0=(hbar*hbar)/(2*m*a*a)/q #divide by q to convert to eV
on=2.0*t0*np.ones(Np+1)
off=-t0*np.ones(Np)
#Define double quantum dot potential
n1=25
n2=75
x1=n1*a/1e-9
x2=n2*a/1e-9
alpha=2.7e-5
V=[]
F=0.001
for i in X:
  v=min(alpha*(i-x1)**2, alpha*(i-x2)**2)+F*i; #eV
  V.append(v)
U=np.array(V*np.ones(Np+1))
#Define Hamiltonian
H=np.diag(on+U)+np.diag(off,1)+np.diag(off,-1)
#solve for eigenvalues and vectors
W,e=np.linalg.eig(H)
idx = W.argsort()[::1]
W = W[idx]
e = e[:,idx]
#calculate probablity
psi=(e[:,0]+e[:,1])
w=0.95*(W[0]-W[1])/hbar
p0plot1=[]
p1plot1=[]
p0plot2=[]
p1plot2=[]
t=np.linspace(0,1.5,101)
for o in t:
    V_new=[]
    for i in X:
        v=min(alpha*(i-x1)**2, alpha*(i-x2)**2)+F*i*np.cos(w*o); #eV
        V_new.append(v)
    A=np.diag(1+1j*o*(on+V_new))+np.diag(off,-1)+np.diag(off,1)
    B=np.diag(1-1j*(on+U)*psi)+np.diag(off,-1)+np.diag(off,1)
    A=A.astype(complex)
    B=B.astype(complex)
    psi_new=np.linalg.solve(A,B)
    Psi0=np.multiply(e[:,0],psi_new[:,0])
    Psi0=np.abs((Psi0))
    #Psi0=np.abs(Psi0)
    Psi1=np.multiply(e[:,1],psi_new[:,1])
    Psi1=np.abs((Psi1))
    #Divide by 900 to renormalize
    p0plot1.append(Psi0/900)
    p1plot1.append(Psi1/900)
    p0plot2.append(np.sum(Psi0/900))
    p1plot2.append(np.sum(Psi1/900))
    U=V_new
    psi=psi_new

fig=plt.figure(16)
ax=fig.add_subplot(111, label="21")
ax2=fig.add_subplot(111, label="22", frame_on=False)
ax.plot(X, p0plot1, color='slategrey')
ax2.plot(X, p1plot1, color='lightsteelblue')
ax.set_xlabel('Distance (nm)')
ax.set_ylabel('Probability')
plt.title('p$_i$(t) = |⟨i|Ψ(t)⟩|$^2$ for Snapshots of Time, F=0.001')
custom_lines = [Line2D([0], [0], color='slategrey', lw=4),
                Line2D([0], [0], color='lightsteelblue', lw=4)]
ax.legend(custom_lines, ['i=0', 'i=1'], loc='center right')
ax2.set_yticks([])
ax2.set_xticks([])
plt.show()

fig=plt.figure(15)
ax=fig.add_subplot(111, label="21")
ax2=fig.add_subplot(111, label="22", frame_on=False)
#Divide by 60 to renormalize
ax.plot(t, np.array(p0plot2)/60, color='slategrey')
ax2.plot(t, np.array(p1plot2)/60, color='lightsteelblue')
ax.set_xlabel('Time (sec)')
ax.set_ylabel('Probability')
#ax.yaxis.set_ticks([0,0.25,0.5,0.75,1])
plt.title('Sum of p$_i$(t) = |⟨i|Ψ(t)⟩|$^2$ plotted against Time, F=0.001')
custom_lines = [Line2D([0], [0], color='slategrey', lw=4),
                Line2D([0], [0], color='lightsteelblue', lw=4)]
ax.legend(custom_lines, ['i=0', 'i=1'], loc='center right')
ax2.set_yticks([])
ax2.set_xticks([])
plt.show()
```

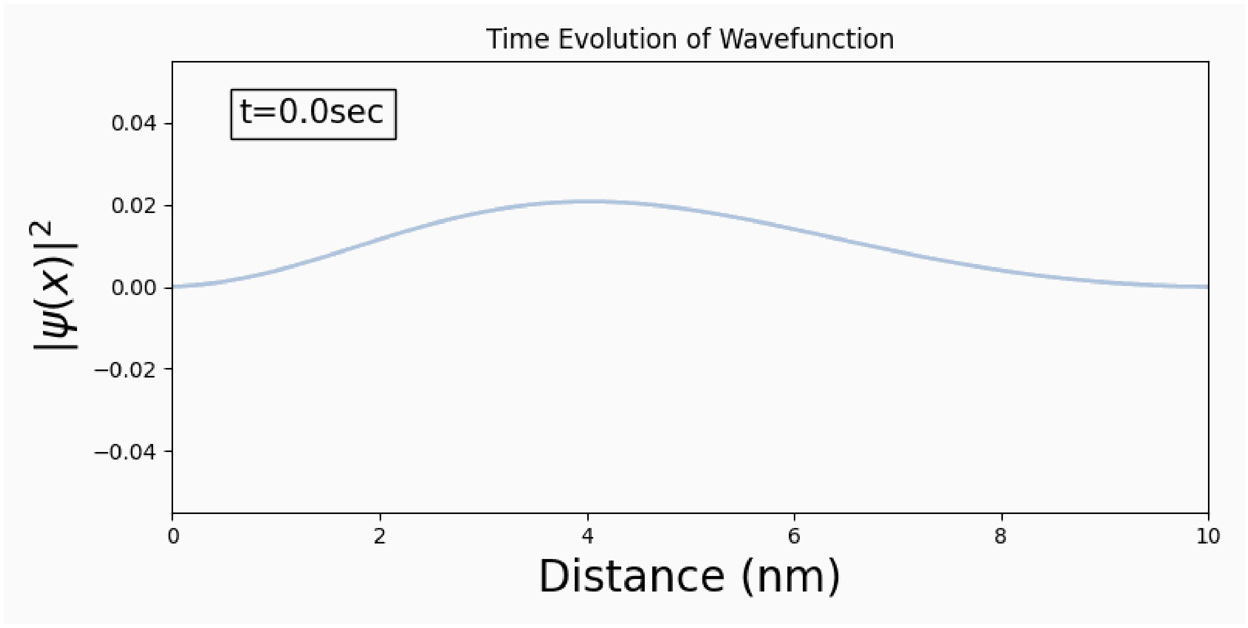


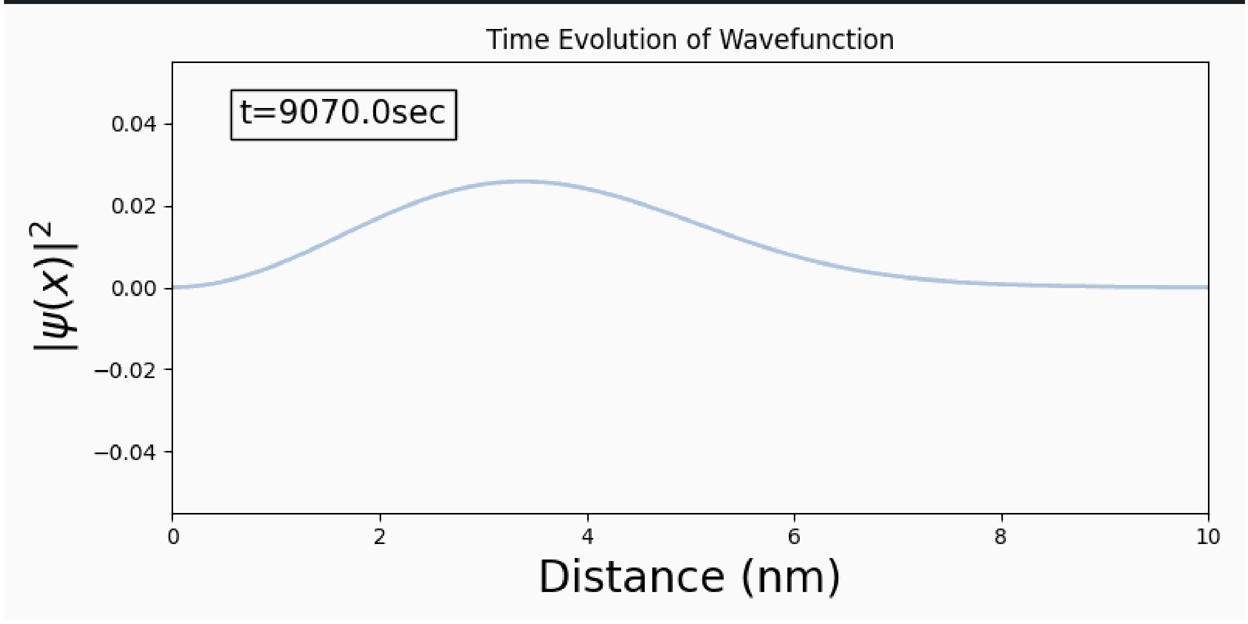


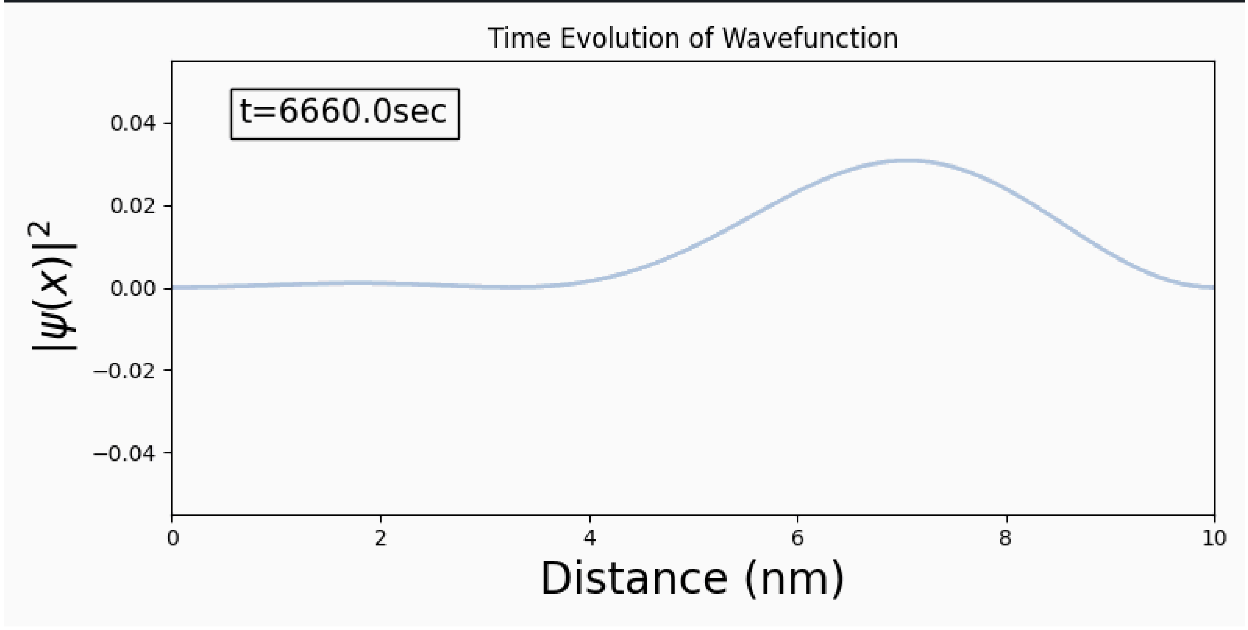
The oscilation appears essentially the same for w=0.95w$_r$ as it did for w=w$_r$.

Extension- Animation

Part D would be easier to understand with an animation, so what follows is the code for viewing the probability function plotted against distance, varying in time. My laptop is old and slow, so I had to run the code in Google Colab. This is a link to the Google Colab: https://colab.research.google.com/drive/1x7bBFJYJwzCFHPjRAIVnDNYVxVBUreNA?usp=sharing and this is a link to the generated gif: https://drive.google.com/file/d/1UkWxEv41nL2T14SrfRKKhYNQpIfNLmnc/view?usp=sharing. I took inspiration from the Git Hub Page in the references. The code is placed below but not executed in this Jupyter Notebook. What follows is the gif produced. I think the beginning of the animation is chaotic due to numerical artifacts from intializing the animation by solving the finite difference equation using matrix methods before switching to iterative methods. The main thing to note from this animation of the simulation is that the wavefunction alternates between the centers of the two wells at 2.5 and 7.5 nm for long times. This alternation can be seen in the screenshots taken and inserted below the gif.


Time Evolution of Wavefunction
t=0.0sec
$|\psi(x)|^2$
Distance (nm)
0.04
0.02
0.00
−0.02
−0.04
0
2
4
6
8
10


Time Evolution of Wavefunction
t=9070.0sec
$|\psi(x)|^2$
Distance (nm)
0.04
0.02
0.00
−0.02
−0.04
0
2
4
6
8
10


Time Evolution of Wavefunction
t=6660.0sec
$|\psi(x)|^2$
Distance (nm)
0.04
0.02
0.00
−0.02
−0.04
0
2
4
6
8
10

In [267…

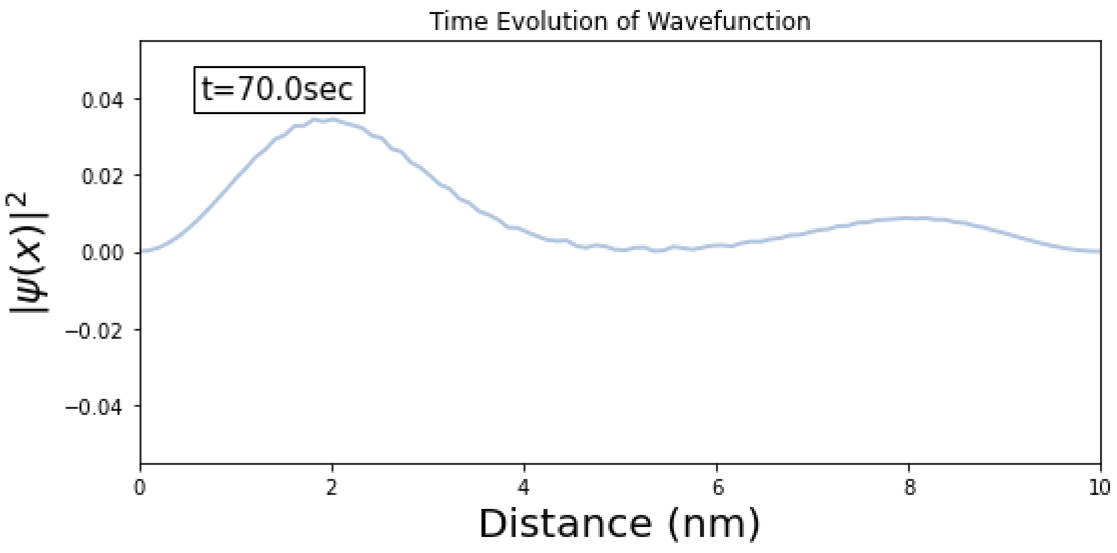
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
#constants
hbar=1.055e-34 #Js
q=1.602e-19    #C
m=9.1e-31      #kg
#grid
Np=100
a=1e-10   #m
X=a*np.linspace(0, Np, Np)/1e-9  #nm
#Define Hamiltonian as a tridiagonal matrix
t0=(hbar*hbar)/(2*m*a*a)/q #divide by q to convert to eV
on=2.0*t0*np.ones(Np)
off=-t0*np.ones(Np-1)
#Define double quantum dot potential
n1=25
n2=75
x1=n1*a/1e-9
x2=n2*a/1e-9
alpha=2.7e-5
V=[]
F=0.001
for i in X:
  v=min(alpha*(i-x1)**2, alpha*(i-x2)**2)+F*i; #eV
  V.append(v)
U=np.array(V*np.ones(Np))
#Define Hamiltonian
H=np.diag(on+U)+np.diag(off,1)+np.diag(off,-1)
#solve for eigenvalues and vectors
W,e=np.linalg.eig(H)
idx = W.argsort()[::1]
W = W[idx]
e = e[:,idx]
#calculate probablity
psi0=e[:,0]
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation
from matplotlib.animation import PillowWriter
import numba
from numba import jit
Np = 100
Nt = 100000
dx = 1/(Np-1)
dt=1e-5
x = np.linspace(0, 10, Np)
hbar=1.055e-34
n1=2.5
n2=7.5
alpha=2.7e-5
F=0.001
#used lowest two energy states of hydrogen atom in eV to find w
w=(10.2+13.6)/hbar
psi=np.zeros([Nt, Np])
psi[0]=psi0
#V=np.zeros([Nt, Np])
#V[0]=U
@numba.jit("c16[:,:](c16[:,:])", nopython=True, nogil=True)
```

```python
def compute_psi(psi):
    for t in range(0, Nt-1):
        for i in range(1, Np-1):
            psi[t+1][i] = psi[t][i] + 1j/2 * dt/dx**2 * (psi[t][i+1] - 2*psi[t][i] + ps
            #V[t][i]=(min(alpha*(i-x1)**2, alpha*(i-x2)**2)+F*i*np.cos(w*t))
        normal = np.sum(np.absolute(psi[t+1])**2)*dx
        for i in range(1, Np-1):
            psi[t+1][i] = psi[t+1][i]/normal
    return psi
psi_m1 = compute_psi(psi.astype(complex))
def animate(i):
    ln1.set_data(x, np.absolute(psi_m1[100*i])**2)
    #ln2.set_data(x, V[100*i])
    time_text.set_text('t='+'{:.1f}'.format(100*i*dt*1e4)+'sec')

fig, ax = plt.subplots(1,1, figsize=(8,4))
#ax.grid()
ln1, = plt.plot([], [], 'lightsteelblue', lw=2, markersize=8, label='Method 1')
#ln2, = plt.plot([], [], 'lightsteelblue', lw=3, markersize=8, label='Method 2')
time_text = ax.text(0.65, 0.04, '', fontsize=15,
               bbox=dict(facecolor='white', edgecolor='black'))
ax.set_xlim(0,10)
#ax.set_ylim(-0.01,0.04)
ax.set_ylabel('$|\psi(x)|^2$', fontsize=20)
ax.set_xlabel('Distance (nm)', fontsize=20)
#ax.legend(loc='upper left')
ax.set_title('Time Evolution of Wavefunction')
plt.tight_layout()
ani = animation.FuncAnimation(fig, animate, frames=1000, interval=50)
ani.save('pen.gif',writer='pillow',fps=50,dpi=100)
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-267-11b63f2f1396> in <module>
     87 plt.tight_layout()
     88 ani = animation.FuncAnimation(fig, animate, frames=1000, interval=50)
---> 89 ani.save('pen.gif',writer='pillow',fps=50,dpi=100)

~\anaconda3\lib\site-packages\matplotlib\animation.py in save(self, filename, writer, fp
s, dpi, codec, bitrate, extra_args, metadata, extra_anim, savefig_kwargs, progress_callb
ack)
   1159                         progress_callback(frame_number, total_frames)
   1160                         frame_number += 1
-> 1161                 writer.grab_frame(**savefig_kwargs)
   1162 
   1163     def _step(self, *args):

~\anaconda3\lib\site-packages\matplotlib\animation.py in grab_frame(self, **savefig_kwar
gs)
    545         from PIL import Image
    546         buf = BytesIO()
--> 547         self.fig.savefig(
    548             buf, **{**savefig_kwargs, "format": "rgba", "dpi": self.dpi})
    549         renderer = self.fig.canvas.get_renderer()

~\anaconda3\lib\site-packages\matplotlib\figure.py in savefig(self, fname, transparent,
**kwargs)
   2309                 patch.set_edgecolor('none')
   2310 
-> 2311         self.canvas.print_figure(fname, **kwargs)
   2312 
   2313         if transparent:
```

```
~\anaconda3\lib\site-packages\matplotlib\backend_bases.py in print_figure(self, filenam
e, dpi, facecolor, edgecolor, orientation, format, bbox_inches, pad_inches, bbox_extra_a
rtists, backend, **kwargs)
   2208
   2209             try:
-> 2210                 result = print_method(
   2211                     filename,
   2212                     dpi=dpi,

~\anaconda3\lib\site-packages\matplotlib\backend_bases.py in wrapper(*args, **kwargs)
   1637             kwargs.pop(arg)
   1638
-> 1639         return func(*args, **kwargs)
   1640
   1641     return wrapper

~\anaconda3\lib\site-packages\matplotlib\backends\backend_agg.py in print_raw(self, file
name_or_obj, *args)
    451     @_check_savefig_extra_args
    452     def print_raw(self, filename_or_obj, *args):
--> 453         FigureCanvasAgg.draw(self)
    454         renderer = self.get_renderer()
    455         with cbook.open_file_cm(filename_or_obj, "wb") as fh:

~\anaconda3\lib\site-packages\matplotlib\backends\backend_agg.py in draw(self)
    405              (self.toolbar._wait_cursor_for_draw_cm() if self.toolbar
    406               else nullcontext()):
--> 407             self.figure.draw(self.renderer)
    408             # A GUI class may be need to update a window using this draw, so
    409             # don't forget to call the superclass.

~\anaconda3\lib\site-packages\matplotlib\artist.py in draw_wrapper(artist, renderer, *ar
gs, **kwargs)
     39                 renderer.start_filter()
     40
---> 41             return draw(artist, renderer, *args, **kwargs)
     42         finally:
     43             if artist.get_agg_filter() is not None:

~\anaconda3\lib\site-packages\matplotlib\figure.py in draw(self, renderer)
   1861
   1862             self.patch.draw(renderer)
-> 1863             mimage._draw_list_compositing_images(
   1864                 renderer, self, artists, self.suppressComposite)
   1865

~\anaconda3\lib\site-packages\matplotlib\image.py in _draw_list_compositing_images(rende
rer, parent, artists, suppress_composite)
    129     if not_composite or not has_images:
    130         for a in artists:
--> 131             a.draw(renderer)
    132     else:
    133         # Composite any adjacent images together

~\anaconda3\lib\site-packages\matplotlib\artist.py in draw_wrapper(artist, renderer, *ar
gs, **kwargs)
     39                 renderer.start_filter()
     40
---> 41             return draw(artist, renderer, *args, **kwargs)
     42         finally:
     43             if artist.get_agg_filter() is not None:

~\anaconda3\lib\site-packages\matplotlib\cbook\deprecation.py in wrapper(*inner_args, **
inner_kwargs)
```

```
    409                                 else deprecation_addendum,
    410                        **kwargs)
--> 411         return func(*inner_args, **inner_kwargs)
    412 
    413     return wrapper

~\anaconda3\lib\site-packages\matplotlib\axes\_base.py in draw(self, renderer, inframe)
   2745             renderer.stop_rasterizing()
   2746 
-> 2747         mimage._draw_list_compositing_images(renderer, self, artists)
   2748 
   2749         renderer.close_group('axes')

~\anaconda3\lib\site-packages\matplotlib\image.py in _draw_list_compositing_images(rende
rer, parent, artists, suppress_composite)
    129     if not_composite or not has_images:
    130         for a in artists:
--> 131             a.draw(renderer)
    132     else:
    133         # Composite any adjacent images together

~\anaconda3\lib\site-packages\matplotlib\artist.py in draw_wrapper(artist, renderer, *ar
gs, **kwargs)
     39                 renderer.start_filter()
     40 
---> 41             return draw(artist, renderer, *args, **kwargs)
     42         finally:
     43             if artist.get_agg_filter() is not None:

~\anaconda3\lib\site-packages\matplotlib\axis.py in draw(self, renderer, *args, **kwarg
s)
   1176         self._update_label_position(renderer)
   1177 
-> 1178         self.label.draw(renderer)
   1179 
   1180         self._update_offset_text_position(ticklabelBoxes, ticklabelBoxes2)

~\anaconda3\lib\site-packages\matplotlib\artist.py in draw_wrapper(artist, renderer, *ar
gs, **kwargs)
     39                 renderer.start_filter()
     40 
---> 41             return draw(artist, renderer, *args, **kwargs)
     42         finally:
     43             if artist.get_agg_filter() is not None:

~\anaconda3\lib\site-packages\matplotlib\text.py in draw(self, renderer)
    679 
    680         with _wrap_text(self) as textobj:
--> 681             bbox, info, descent = textobj._get_layout(renderer)
    682             trans = textobj.get_transform()
    683 

~\anaconda3\lib\site-packages\matplotlib\text.py in _get_layout(self, renderer)
    285 
    286         # Full vertical extent of font, including ascenders and descenders:
--> 287         _, lp_h, lp_d = renderer.get_text_width_height_descent(
    288             "lp", self._fontproperties,
    289             ismath="TeX" if self.get_usetex() else False)

~\anaconda3\lib\site-packages\matplotlib\backends\backend_agg.py in get_text_width_heigh
t_descent(self, s, prop, ismath)
    235 
    236         flags = get_hinting_flag()
--> 237         font = self._get_agg_font(prop)
    238         font.set_text(s, 0.0, flags=flags)
    239         w, h = font.get_width_height()  # width and height of unrotated string

~\anaconda3\lib\site-packages\matplotlib\backends\backend_agg.py in _get_agg_font(self, prop)
    270         Get the font for text instance t, caching for efficiency
    271         """
--> 272         fname = findfont(prop)
    273         font = get_font(fname)
    274 

~\anaconda3\lib\site-packages\matplotlib\font_manager.py in findfont(self, prop, fontext, directory, fallback_to_default, rebuild_if_missing)
   1312             prop, fontext, directory, fallback_to_default, rebuild_if_missing,
   1313             rc_params)
-> 1314         return os.path.realpath(filename)
   1315 
   1316     @lru_cache()

~\anaconda3\lib\ntpath.py in realpath(path)
    645             path = join(cwd, path)
    646         try:
--> 647             path = _getfinalpathname(path)
    648             initial_winerror = 0
    649         except OSError as ex:

KeyboardInterrupt: 
```



Conclusion

Therefore, it has been determined that a value of alpha=2.7e-5 produced an energy level splitting of 4 meV and that the theoretical qubits simulated in this experiment would function between the temperature range of 2.44 to 3.79 K. It was also determined that the magnitude of any detuning bias applied must be less than 7.5 meV for the qubits to remain operations.

References

I. Griffiths D., "Introduction to Quantum Mechanics" (2018) - 3rd edition
II. Rahman, R. (2024). "Time Dependent PDEs"

https://moodle.telt.unsw.edu.au/pluginfile.php/11703002/mod_resource/content/6/Time_dependence_le
III. Rahman, R. (2024). "Eigenvalue
Problems."https://moodle.telt.unsw.edu.au/pluginfile.php/11579722/mod_resource/content/5/Schroding
IV. Polson, L. (2021). "Eigenvalue
Problems."https://github.com/lukepolson/youtube_channel/blob/main/Python%20Metaphysics%20Serie