1 Introduction

A potential well is a confined region of space where a particle's potential energy is lower than the surrounding area. Each discrete energy level of the particle in the well is in a bound state if it is smaller than the potential energy at the boundary. In contrast to classical mechanics, where particles can oscillate over a continuous range of energies, quantum mechanics restricts the energy levels of particles in a well to discrete values. The first energy level at which the confined particle is in equilibrium is called the ground state. Rarely, a particle may be able to tunnel through the borders and leave the well. Using numerical and iterative approaches to solve for their wavefunction and energy, this notebook investigates the behaviour of particles in bound states of a potential well, comparing the outcomes to analytical solutions when available.

Global variables section

Below are a list of global variables which involve both univsersal constats of physics aswell as paramters which relate to the properties of the sytem being investigated.

In [1]:

```
#importing modules
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
#!pip install pandas
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)

# USEFULL constants: mass off electron (kg), hbar, electron charge (C),
q_electron = 1.602176634e-19
m_electron = 9.109383702e-31
h_bar = 1.054571817e-34

#Parameters for potential I (Infinite square well):

energy_values_I = np.linspace(0,30000,1500) # Array of 30000 energy values in eV
energy_array_I = np.linspace(0,30000*q_electron,1500) # Array of energies in joules
frame_levels_I = np.linspace(1,28,28) # The number of energy levels that are calculated

#Parameters for potential II (Harmonic potential/ Finite square well / Double well):

energy_values_II = np.linspace(0,1500,350) # Array of 1000
energy_array_II = np.linspace(0,1500*q_electron,350)# Array of energies
frame_levels_II = np.linspace(0,3,4)#  number of energy levels


# parameters for numerical integration
N = 2000
half_width = 5e-11  #x10^11
x_start = -half_width
x_end = half_width
h_step = (abs(x_start)+abs(x_end))/N
x_points = np.arange(x_start, x_end, h_step) # Array of x points to iterate over
#ground state initialisation
wave_func_initial = 0 # Initial wave_function value
phi_initial = 2 # Initial phi value
r = np.array([wave_func_initial,phi_initial]) # r-array substituted into the Shroedinger

#Initial potential energy values:

potential_V0 = 700*q_electron # potential well (for HAROMNIC well)
potential_V0_finite = 600*q_electron # potential well (for FINTIE well)


ground_state_energy_level = 1
new_num_data_points = 2001
h_step2 = (abs(x_start)+abs(x_end))/new_num_data_points # Step size in units m
x_points2 = np.arange(x_start, x_end, h_step2) # Array of x points to iterate over
```

An computational approach to solving the Shrodinger equattion;

The Schrödinger equation, which is a cornerstone of quantum mechanics, defines how particles behave when exposed to potential energy. It is a one-dimensional equation that was developed by Erwin Schrödinger in 1926 that connects a particle's wavefunction, represented by the symbol, to both its energy and the system's potential energy.

The time-independent Schrödinger equation is given by:

where V(x) is the potential energy as a function of location, m is the particle's mass, and is the reduced Planck constant.

The link between the particle as a physical quantity and the likelihood of finding it at a particular point is provided by the wavefunction, which is a complex quantity. One can ascertain the energy levels and wavefunctions of particles in a specific potential energy well by resolving the Schrödinger equation.

$$i\hbar\partial\psi/\partial t = -\hbar^2/2m\ \partial^2\psi/\partial x^2 + V(x)\psi$$

where V(x) is the potential energy as a function of location, m is the particle's mass, and is the reduced Planck constant.

The energy and wavefunction of a particle in a potential well can be determined by numerically solving the Schrödinger equation. The equation can also be solved for a bound particle's permitted energy levels, or eigenstates, as well as the corresponding wavefunctions of these discrete energy states.

The Schrödinger equation is initially decomposed into one-dimensional differential equations for numerical solution. These equations can then be subjected to iterative approaches. The deconstructed one-dimensional relationships are described as:

$$\partial\psi/\partial x = \phi$$

and

$$\partial\phi/\partial x = 2m\hbar^2[V(x) - E]\psi(x)$$

where V(x) is the potential energy as a function of location, m is the particle's mass, and is the reduced Planck constant.

where E represents the particle's energy and is a brand-new function connected to the wavefunction.

These equations will be used throughout the notebook because they are applicable to all potentials. The answers to these equations offer a helpful framework for comprehending how particles behave in quantum mechanical systems like potential wells.

In [2]:

```
def solve_schrodinger_eq(r, x_points, E, V):
    """
    This function solves the one-dimensional time-independent Schrödinger equation by decomposing
    it into two one-dimensional differential equations. outputting these 2 elements.

    Parameters:
    r (np.array):Psi,dPsi/dx.
    x_points (np.array): Points for Runge kutta
    E (float): particle energy
    V (np.array): potential energy

    Returns:
    np.array: An array of values of dPsi/dx and d2Psi/dx2
    (output format= numpy float)
    """

    # Extracting Psi and dPsi/dx from the input array r
    Psi = r[0]
    dPsi_dx = r[1]

    # Computing the second derivative of Psi
    d2Psi_dx2 = (2*m_electron/h_bar**2)*(V-E)*Psi

    # Returning the output values as a numpy array
    return np.array([dPsi_dx, d2Psi_dx2], float)
```

In [ ]:

```

```

In [3]:

```
def is_even(x):
    '''
    This function take an integer number x as an input ans outputs a boolean value which indicates whether the
    '''
    if x%2==0:  # modulo operator
        return True
    else:
        return False
```

Wavefunction normalisation

This function takes an input array-like object psi and normalizes it using the trapezoidal rule. The np.abs function calculates the absolute value of psi, and then it is squared using **2 operator to get the psi_squared array.

Then, the psi_squared array is truncated to exclude the first and last element using slicing, and the integral is computed using the trapezoidal rule formula. The result of the integral is used to normalize the psi array.

Finally, the normalized psi array is returned as psi_norm.

In [4]:

```
#function good!
```

```
def normalize_wavefunction(psi):
    """
    Normalizes the wave function using the trapezoidal rule.
    Args:
    - psi (array-like): the wave function to normalize

    Returns:
    - psi_norm (array-like): the normalized wave function
    """
    psi_squared = np.abs(psi)**2
    psi_squared_truncated = psi_squared[1:-1]  # exclude boundary values
    integral = h_step/2 * (psi_squared[0] + psi_squared[-1] + 2*np.sum(psi_squared_truncated))
    psi_norm = psi / np.sqrt(integral)

    return psi_norm
```

Scientific computing with Shrodinger: 1.) Runge Kutta function.

In scientific computing, the Runge Kutta method is a potent numerical approach for approximating differential equation solutions. It can more accurately predict the behaviour of a quantum particle inside a potential well when applied to the Schrödinger equation. In order to approximate the weighted average of the actual slope, the approach uses four distinct slopes, k1, k2, k3, and k4. It is necessary to output an array of the derivatives of the ODEs created by the Schrödinger equation in order to use the Runge Kutta method. The Runge Kutta method then employs these derivatives directly to determine the slope's value. This method can be used to construct the wavefunction plot for a particular energy and potential while approximating the particle's wavefunction over a wide range of x-values. The Runge Kutta method merely offers an approximation of the answer, and there might be differences between the analytical and numerically derived solutions.

In [5]:

```
def RungeKutta3d(state, positions, step_size, ode_function, energy, potential_energy):
    """
    This function solves a system of two ordinary differential equations using
    the fourth-order Runge-Kutta method. The input function is split into ODEs
    for the dependent variable phi and its derivative dphi/dx. The inputs are:

    state: A 2-dimensional array with the first component being the dependent
           variable phi, and the second component being its derivative dphi/dx.
    positions: An array of N position points to iterate over.
    step_size: The step size for the Runge-Kutta method.
    ode_function: The function defining the differential equations to be solved.
    energy: The total energy of the particle.
    potential_energy: An array of potential energy values at each position point.

    Returns:
    [psi_points, phi_points]: Solutions to the inputted function at each position point.
    """
    # Initialise empty arrays to store calculated values:
    psi_points = [] # Array to store calculated psi values
    phi_points = [] # Array to store calculated phi values

    for i in range(len(positions)):
        x = positions[i] # Get the current position

        # Store the current psi and phi values:
        psi_points.append(state[0])
        phi_points.append(state[1])

        # Calculate the four K values using the current state and position:
        k1 = step_size * ode_function(state, x, energy, potential_energy[i])
        k2 = step_size * ode_function(state + 0.5*k1, x + 0.5*step_size, energy, potential_energy[i])
        k3 = step_size * ode_function(state + 0.5*k2, x + 0.5*step_size, energy, potential_energy[i])
        k4 = step_size * ode_function(state + k3, x + step_size, energy, potential_energy[i])

        # Update the state using the weighted average of the K values:
        state = state + (k1 + 2*k2 + 2*k3 + k4)/6

    # Append the final psi and phi values to their respective arrays:
    psi_points.append(state[0])
    phi_points.append(state[1])

    # Convert to a numpy array and return:
    return np.array([psi_points, phi_points])

# This is an adapted version of Runge Kutta method funnction from Session07(PHAS0029)
```

Scientific computing with Shrodinger: 2.) Secant function.

Another numerical method for calculating the energy of the particle inside the potential well is the Secant method. In this approach, two energy predictions are given, and the numerical algorithm repeatedly reduces the gap between them until they converge to be within a certain tolerance. Nonetheless, although being extremely accurate, applying such a tolerance could result in differences between the theoretical and numerical energy solutions, particularly at high energies.

In [6]:

```
def secant(initial_guesses,rk,num,V,change_of_sign_arr,deltaE):
    """
    Calculates initial energy guesses and uses the secant method to find a solution
    to the Schrodinger equation to within a certain tolerance for the energy.

    Args:
        initial_guesses (array): An array of initial psi and phi values.
        rk (function): A function to solve the Schrodinger equation using the
                       fourth-order Runge Kutta method.
        num (int): The energy level.
        V (array): An array of potentials corresponding to a total potential evaluated
                   across x.
        change_of_sign_arr (array): An array of points where a change of sign occurs.
        deltaE (array): An array of energy values corresponding to the array
                                of energies the change of sign was calculated from.

    Returns:
        float: The energy solution given the initial guesses in joules.
    """
    inital_entry = Energy_level_guesses(deltaE,change_of_sign_arr,num-1)
    arga = inital_entry[0]*q_electron #first value of array
    argb = inital_entry[1]*q_electron #second value of array
    # Runge kutta array calculation
    #implementing the runge kutta function
    solution1 = RungeKutta3d(initial_guesses,x_points,h_step, solve_schrodinger_eq, arga,V)[0,N]
    # This is the Runge kutta Solution for first energy array entry (arga)
    solution2 = RungeKutta3d(initial_guesses,x_points, h_step,solve_schrodinger_eq, argb,V)[0,N]
    # This is the Runge kutta Solution for second energy array entry (argb)
    tolerance = abs(arga)/1000
    # TOLERANCE of the inital guess for energy value




    while abs(argb-arga) > tolerance: # looping energy difference over tolerance
        energy_ = argb - solution2*(argb-arga)/(solution2-solution1)
        arga = argb # variable re-assignment
        argb = energy_ #

        solution1 = RungeKutta3d(r,x_points, h_step,solve_schrodinger_eq, arga,V)[0,N]
        solution2 = RungeKutta3d(r,x_points,h_step, solve_schrodinger_eq, argb,V)[0,N]
        #^^recalculations of energy solutions
    return(energy_)

#This is an adaption of the secant method from PHAS0029-Session08-SecantMethod
```

Calculating energy levels

The energy finding functions are designed to classify well types and then solve for energy solutions. It is impotrant to note that both types of solutions which are being solved for in this notebook are not always possible to solve for with certain types of potential i.e - finite well.

In [7]:

```
def energy_calculator(energy_values_joules, num_levels, energy_guesses_ev, potential_energy_v):
    """
    Calculates the energy levels of a quantum system using the provided energy values and potential.
    First, it finds the point of change in sign in the wave function, indicating the existence of an energy leve
    Next, it generates energy level guesses using the energy_guesses_ev array and the previously obtained sign
    Finally, it returns a pandas data frame containing the calculated energy levels and two corresponding energ
    in electron volts (eV).

    Arguments:
    energy_values_joules: An array of energy values in joules.
    num_levels: The number of energy levels to generate guesses for.
    energy_guesses_ev: An array of energy guesses in electron volts (eV).
    potential_energy_v: The potential energy of the system.

    Outputs:
    energy_levels_df: A pandas data frame containing the calculated energy levels and two corresponding energy
    intersections: A list of values indicating the intersection of the wave function with the potential energy
    sign_changes: An array of indices indicating where a change of sign occurs in the wave function.
    """

    intersections = [] # create empty array to store intersection data
    for i in range(len(energy_values_joules)):
        energy_value = energy_values_joules[i]
        # calculate the intersection point with the potential energy boundary using Runge-Kutta 3d method
        solution_for_guesses = RungeKutta3d(r, x_points, h_step, solve_schrodinger_eq, energy_value, potential_
        intersections.append(solution_for_guesses) # appending intersection solutions to empty array

    sign_changes = np.asarray(np.where(np.diff(np.sign(intersections)))) + 1
    # ^^the sign_changes array finds the indices where a change of sign occurs in the wave function
    column_names = ['Energy Level', 'Energy Guess 1', 'Energy Guess 2'] # Column headers
     # ^^comlumn headers are for the pandas module dataframe(table)
    energy_levels_df = pd.DataFrame(columns=column_names)
    for level in range(len(num_levels)):
        energy_guesses = Energy_level_guesses(energy_guesses_ev, sign_changes, level) # Finding the energy gues
        #^^# generate energy level guesses for each energy level using the energy_guesses_ev array and sign_cha
        guess_1 = energy_guesses[0] #guess value(s)
        guess_2 = energy_guesses[1] #guess value(s)
        energy_levels_df = energy_levels_df.append(
            pd.Series([level+1, guess_1, guess_2], index=column_names),# append the calculated values to the pa
            ignore_index=True
        )
    energy_levels_df = energy_levels_df.style\
        .hide_index()\
        .set_caption('Pandas dataframe showing energy gues values output from energy_calculator() function')

    return energy_levels_df, intersections, sign_changes
```

In [8]:

```
def Energy_level_guesses(energy_guesses_ev, delta, n):
    """
    Calculates the initial energy guesses in joules for a given energy level based on the array
    of energy guesses in electron volts (eV) and the indices indicating where a change of sign
    occurs in the wave function.

    Arguments:
    energy_guesses_ev: An array of energy guesses in electron volts (eV).
    delta: An array of indices indicating where a change of sign occurs in the wave function.
    n: The energy level for which the guesses are generated.

    Outputs:
    output1_1: The lower initial energy guess in joules.
    output_2: The upper initial energy guess in joules.
    """

    # Find the index of the change of sign array with the correct energy level
    index_of_change = delta[0][n]

    # Find the energy guess array with the right value for energy (cross point)
    Energy_L = energy_guesses_ev[index_of_change]

    # Calculate the energy guesses
    output1_1 = Energy_L - Energy_L / 50
    output_2 = Energy_L + Energy_L / 50

    return output1_1, output_2
```

In [ ]:

In [9]:

```
#frday: GOOD ENOUGH FOR NOW
def energy_finder1(energylevels,pot,delta,prev_sol, potential_classification):
    """
This function uses the secant method to calculate energy solutions for given energy levels and well types.
The solutions are outputted in a pandas data frame along with the energy solutions
for each level. The function takes in five inputs including the energy levels array,
the potential energy, and the well type. The theoretical energy for each eigenstate
is calculated and compared to the calculated energy using the secant method.
The solution array and data frame are outputted.
    """
    emptyarr = [] # An empty array is created to store the calculated energy solutions.
    data_entry = ['N(level)','Predicted / eV','Calculated (eV)', 'Difference (eV)']
    #^^This line creates a list of column names to be used in the final pandas data frame.
    out = pd.DataFrame(columns = data_entry)
    for N_ in range(len(energylevels)):
        # This for-loop iterates through each energy level in the array of energy levels to solve for.
        if potential_classification == "Infinite": # checks if the type of well is an infinite potential well,
            eig = (((np.pi**2)*(h_bar**2)*((N_+1)**2))/(2*m_electron*((2*half_width)**2)))/q_electron
            pn = eig*q_electron # Conversion of the theoretical value to eV from joules
            VALS1 = secant(r,RungeKutta3d,N_+1,pot,delta,prev_sol)
            # calculates the energy solution for the current energy level using the secant method.
            VALS2 = VALS1/q_electron # This line converts the energy solution from joules to eV.
            DELTAE = abs(eig-VALS2) # This line calculates the absolute difference between the theoretical and
            emptyarr.append(VALS1) # This line appends the energy solution to the array of energy solutions.
        elif potential_classification == "Harmonic":
            # This if-statement checks if the type of well is a Harmonic potential well, and if so, calculates
            eig = ((N_ + 1/2)*h_bar*np.sqrt((2*potential_V0/half_width**2)/m_electron))/q_electron # Analytical
            pn = eig*q_electron
            VALS1 = secant(r,RungeKutta3d,N_+1,pot,delta,prev_sol)
            VALS2 = VALS1/q_electron
            DELTAE = abs(eig-VALS2)
            emptyarr.append(VALS1)
        else: # Solutions to the harmonic potnetial
            pn = "n/a"
            eig = pn
            # This line sets the difference between the theoretical and analytical energy values as "n/a" since
            DELTAE = pn   #
            VALS1 = secant(r,RungeKutta3d,N_+1,pot,delta,prev_sol)
            VALS2 = VALS1/q_electron
            emptyarr.append(VALS1)

        row1 = eig
        row2 = VALS2
        row3 = DELTAE
        out = out.append(
                            pd.Series([N_+1,
                                     row1,
                                    row2,
                                    row3],
                                    index = data_entry),
                            ignore_index = True)

    fd = out.style\
            .hide_index()\
            .set_caption('Pandas dataframe showing energy values output from energy_finder1() function')
    return fd, emptyarr # solution output
```

In [ ]:

In [ ]:

In [10]:

```
#function good!
```

The plotting wavefunction:

The function offers both the analytical and numerical solutions to the wavefunction, which are normalised and plotted wherever it is practical on the same plot. It functions according to the same categorization scheme as the energy finding function. The analytical wavefunction is displayed as a green dotted line next to the numerical solution determined using the Runge Kutta method. The calcplot function uses a loop structure to ensure that the required classification is obeyed. The differences between the analytical and numerical solutions are easier to perceive and help one comprehend the particle's physical behaviour inside the potential well by plotting both solutions on the same plot. The energy finding function can be used in conjunction with the plotter function to gain a more comprehensive understanding of the system. The plotter function offers a helpful tool for visualising the wavefunction of the particle, which can help in understanding the physical behaviour of the particle.

In [11]:

```
def calcplot(energies, pos, n_, V, potential_classification):
    """
    Calculates and plots the wave function for each energy level inputted against the theoretical solution.

    Args:
        energies: Energy solutions for the well.
        pos: An array of x points that is one longer than the other x point array as the dimensions of the wave
        n_: ENERGY LEVEL
        potential_energy: Potential energy at a point x in the well.
        potential_classification: Keyword that assigns the correct analytical wavefunction if one exists, to th

    Returns:
        None.
    """
    init_pos = pos[0]
    # Calculate wave function using Runge-Kutta method
    wav_func_solution = RungeKutta3d(r,x_points,h_step,solve_schrodinger_eq,(energies),V)
    # Normalize the wave function
    Normalised_wf = normalize_wavefunction(wav_func_solution[0])


     # Plot the wave function based on the given wave type
        #conditionals for potential_classification
    if potential_classification == "Infinite":
        # Even energy level check usisng earlier defined is_even() function
        if is_even(n_)==True: # This assigns the correct analytical solution depending on energy level.
            Analytical_Nwf = (1/np.sqrt(half_width))*np.sin((n_*np.pi*x_points)/(2*half_width))
            plt.plot(x_points,Analytical_Nwf,'-.',label="analytical",color = 'g')
            plt.plot(x_points2,Normalised_wf,'-',label="Phi NORM",color = 'b')




        else:  # Odd energy level
            Analytical_Nwf = (1/np.sqrt(half_width))*np.cos((n_*np.pi*x_points)/(2*half_width))
            plt.plot(x_points,Analytical_Nwf,'-.',label="analytical",color = 'g')
            plt.plot(x_points2,Normalised_wf,'-',label="Phi NORM",color = 'b')




    elif potential_classification == "Harmonic":  # Harmonic potential well
         # Find the analytical wave function
        Analytical_Nwf = harmonic_wavefunction(n_, x_points)
        plt.plot(x_points,Analytical_Nwf,'-.',label="analytical",color = 'g')
        plt.plot(x_points2,Normalised_wf,'-',label="Phi NORM",color = 'b')
    else: # Finite potential well
        # Plot only the normalized wave function since there are no analytical solutions
        plt.plot(x_points2,Normalised_wf,'-',label="Phi NORM",color = 'b')

    plt.xlabel("X value (m)")
    plt.ylabel(" Wavefunction : $\psi(x)$")
    plt.ticklabel_format(axis= 'y',style = 'sci', scilimits=(0,0)) # Setting the y-axis into scientific notatio
    plt.axvline(x=-half_width, c = 'b')
    plt.axhline(y=0, c = 'b')
    plt.axvline(x=half_width, c = 'b')
    plt.legend(loc="lower left", fontsize = 'small');
    #^^formatting
    return
```

The Harmonic potential function

It is crucial to define the potential energy of the well in order to calculate a particle's behaviour inside an infinite square well. Potential energy is equal to zero for an infinite square well. Nevertheless, an array is produced instead of just printing a constant value of zero, with a length equal to the number of x points used to determine the well width. This is due to the fact that, despite the fact that the potential energy within the well is constant and equal to zero, it is frequently essential to evaluate the potential energy for certain x positions. The potential energy for each individual x point can be accurately assessed by creating an array without the need for extra calculations. This strategy guarantees that the potential energy is accurately represented for each place inside the well, enabling precise numerical calculations of the particle's behaviour. Understanding the physical behaviour of the particle inside the well and gaining valuable insights into the system depend on accurate computation of the potential energy. In our specific case, we have chosen an infinite square well with a well depth of 700e.

In [12]:

```
#all good!
def potential_energy(pos):
    """
    This function calculates and returns the potential energy at each position
    in the well.
    Inputs:
    pos: A 1D numpy array of the x positions being considered.
    Outputs:
    A 1D numpy array of zeros, the same length as pos array, representing the
    potential energy at each position in the well.
    """
    # Since potential_classificcation == INFINITE : (from prev function), the potential energy is zero --> unit
    return np.zeros_like(pos)
```

In [13]:

```
def harmonic_wavefunction(level, x_values):
    """
    Calculates the wavefunction of a particle in a harmonic potential using
    Hermite polynomials. The input is the energy level of the particle and an
    array of x values the well is distributed across. The output is an array
    containing the wave function for the given energy level.

    Parameters:
    level (int): Energy level of the particle.
    x_values (array): Array of x values representing the potential well.

    Returns:
    array: An array of the wave function for a particle in a harmonic potential.
    """
    m = m_electron # Mass of the particle
    V0 = potential_V0 # Depth of the potential well
    a = half_width # Width of the potential well
    alpha = np.sqrt((2*m*V0)/(a**2))/h_bar # Constant alpha for the potential well
    const = (alpha/np.pi)**0.25 # Constant used throughout the wavefunction calculation
    y_values = np.sqrt(alpha)*x_values # Array of y values corresponding to x values
    wavefunction = [] # Array to hold the wavefunction values

    # Calculation of wavefunction based on energy level
    if level == 0: # Ground state
        for y in y_values:
            wf = const*np.exp(-(y**2)/2)
            wavefunction.append(wf)
    elif level == 1: # First excited state
        for y in y_values:
            wf = const*np.sqrt(2)*y*np.exp(-(y**2)/2)
            wavefunction.append(wf)
    else: # Higher energy levels
        for y in y_values:
            wf = const*(1/np.sqrt(2))*(2*(y**2)-1)*np.exp(-(y**2)/2)
            wavefunction.append(wf)

    return wavefunction
```

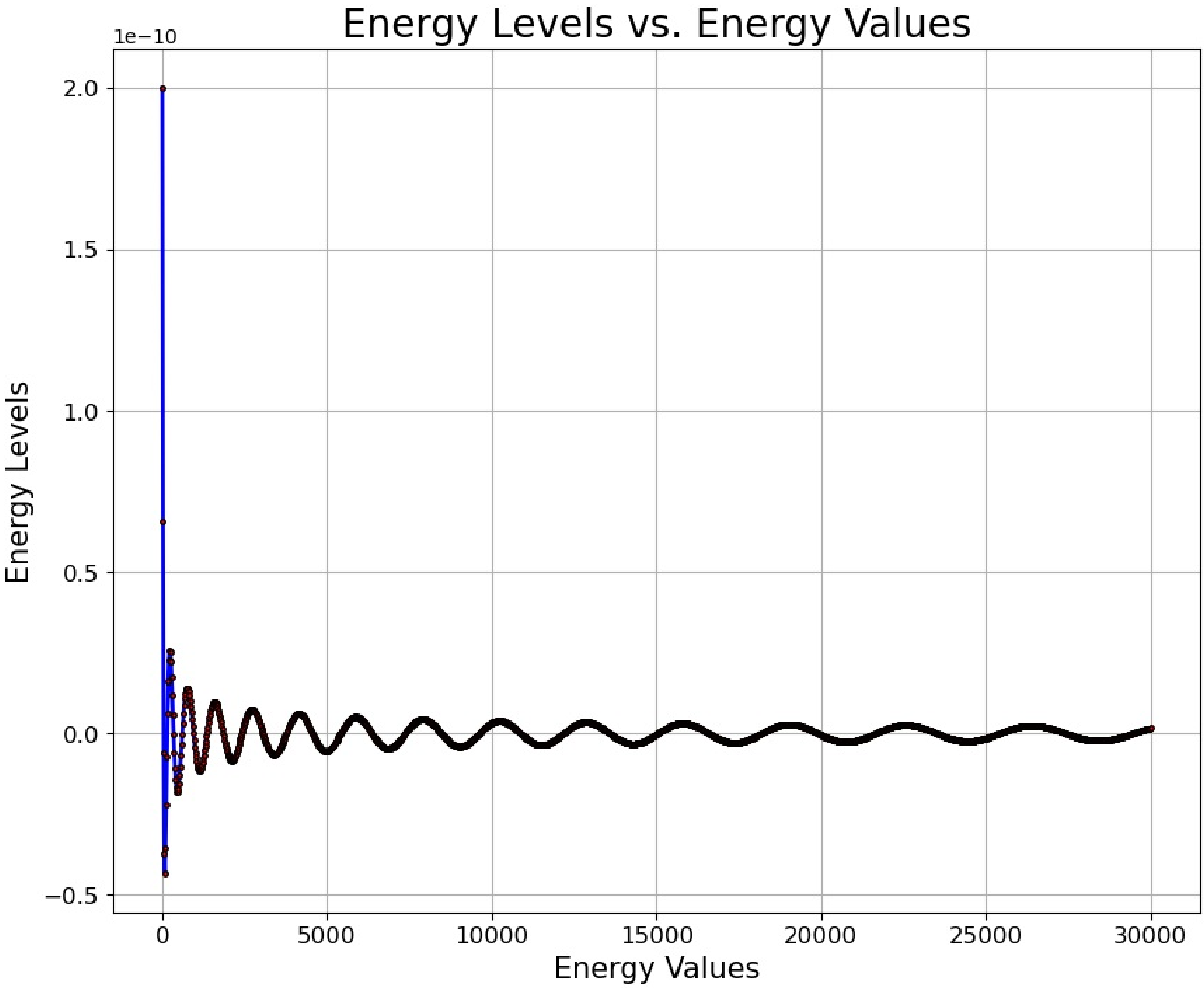
In [ ]:

In [ ]:

```
```

In [14]:

```
# calculate the potential energy of x points
V_pot0 = potential_energy(x_points)

# calculate the energy levels and associated energies
Energy_guess = energy_calculator(energy_array_I, frame_levels_I, energy_values_I, V_pot0)

# plot energy levels as a function of energy values
plt.figure(figsize=(10,8))
plt.plot(energy_values_I, Energy_guess[1], color='blue', linewidth=2, marker='.', markersize=5, markerfacecolor
plt.xlabel('Energy Values', fontsize=15)
plt.ylabel('Energy Levels', fontsize=15)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.title('Energy Levels vs. Energy Values', fontsize=20)
plt.grid(True)
plt.show()

# print the lowest energy level
print(f"Lowest Energy Level: {Energy_guess[0]}")
```



```
Lowest Energy Level: <pandas.io.formats.style.Styler object at 0x7fa61972c370>
```

3.2 Finding the ground state wavefunction and ascociated energies

In [15]:

```
#old block:0.2

# Theoretical ground state energy
E = ((np.pi**2)*(h_bar**2))/(2*m_electron*(2*half_width)**2)
# Finding the energy solution for the ground state
En2 = secant(r,RungeKutta3d,1,V_pot0,Energy_guess[2],energy_values_I)
# # Conversion to eV
Energy_sol1 = En2/q_electron
#eV from joules
E_Eigenvalue = E/q_electron
# deltaE
delE = abs(E_Eigenvalue-Energy_sol1)

# Printing the results:
print('Preliminary energy readings')
print("Ground state energy:",Energy_sol1,"eV")
print("Energy at n=1   \n defined by: \n E-ground = {0:n} eV \
\n    \n   \n \
Delta E = {1:n} eV".format(E_Eigenvalue,delE))
```

```
Preliminary energy readings
Ground state energy: 37.603032633841515 eV
Energy at n=1
 defined by:
 E-ground = 37.603 eV


 Delta E = 1.64203e-05 eV
```

In [ ]:

In [16]:

```
#calculation of the ground state wavefunction with rungekitta

3
#non normalised solution
solution_wavefunction_NN = RungeKutta3d(r,x_points,h_step,solve_schrodinger_eq,En2,V_pot0)
```

In [17]:

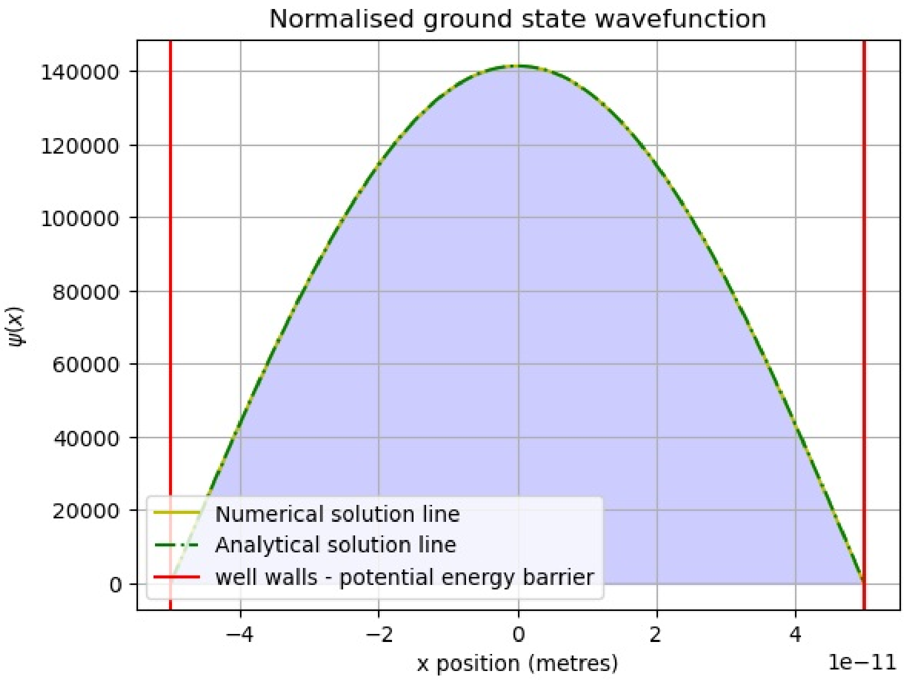
```
solution_wavefunction_N = normalize_wavefunction(solution_wavefunction_NN[0]) # Normalising the wavefunction fo

sol2 = (1/np.sqrt(half_width))*np.cos((1*np.pi*x_points)/(2*half_width)) #analytical solution
```

In [18]:

```
#producing plot --> NORMALISED


plt.figure()
plt.plot(x_points2, solution_wavefunction_N, '-', label="Numerical solution line", color='y')  # Plotting norma
plt.plot(x_points, sol2, '-.', label="Analytical solution line", color='g')  # Plotting the analytical solution
# marking the walls of the system (-> 'You may wish to represent the walls of the infinite square well potentia
plt.axvline(x=x_start, c='r', label="well walls - potential energy barrier")
plt.axvline(x=x_end, c='r')
# Shade under the curve
plt.fill_between(x_points2, solution_wavefunction_N, 0, alpha=0.2, color='b')
# Add grid lines
plt.grid(True)
# Plot formatting
plt.legend(loc="lower left")
plt.title("Normalised ground state wavefunction")
plt.xlabel("x position (metres)")
plt.ylabel("$\psi(x)$")
```

Out[18]:

```
Text(0, 0.5, '$\\psi(x)$')
```



Matching numerical and analytical solutions

As expected we see that we have an excellend matech between out numerical and analytical solutions. This is a good indication that we are on the right track.

In quantum mechanics, wave-functions describe probability amplitudes of finding a particle at some particular position. Wave functions are mathematical functions that satisfy the Schrödinger equation. They can be calculated analytically or numerically for certain systems.

The Schrödinger equation's analytical wave function can be stated in a closed form, usually using well-known mathematical functions. For instance, depending on the boundary conditions, the analytical wave function for a particle in a one-dimensional infinite potential well is either a sine or a cosine function.

The numerical wave function, on the other hand; is computed numerically using computing techniques.The Runge Kutta method (can be found in functions) is used in the code you gave to solve the time-independent Schrödinger equation for a given potential to produce the numerical wave function.

The normalisation of the wave function is very important as it ensures that the summed total probability of finding the particle in all possible positions is equal to 1(in out case this means all possible x values). This means that the wave function has a well-defined probability density, and it can be used to calculate the probability of finding the particle in a certain region of space. In the code above, the normalisation of the numerical wave function is done using the normalisation() function, calculating the L2 norm of the wave function and scales it such that its integral over all space is equal to 1.

In [19]:

```
well_class1 = "Infinite"  #setting well type to infinite for loop in energy_finder1() function
energy_values = energy_finder1(frame_levels_I, V_pot0, Energy_guess[2],energy_values_I, well_class1)
```

3.3 Finding the higher energy states

Table showing eenergy guess values, output using pandas module with pandas dataframe

In [20]:

```
energy_values[0]
```

Out[20]:

Pandas dataframe showing energy values output from energy_finder1() function

N(level)	Predicted / eV	Calculated (eV)	Difference (eV)
1.000000	37.603016	37.603033	0.000016
2.000000	150.412065	150.412174	0.000109
3.000000	338.427146	338.424993	0.002153
4.000000	601.648259	601.670089	0.021830
5.000000	940.075405	940.069721	0.005685
6.000000	1353.708584	1353.695293	0.013290
7.000000	1842.547794	1842.520480	0.027314
8.000000	2406.593038	2406.556378	0.036660
9.000000	3045.844313	3045.792129	0.052185
10.000000	3760.301621	3760.261845	0.039777
11.000000	4549.964962	4549.882967	0.081995
12.000000	5414.834335	5414.743437	0.090897
13.000000	6354.909740	6354.790926	0.118814
14.000000	7370.191178	7370.011661	0.179516
15.000000	8460.678648	8460.517228	0.161420
16.000000	9626.372151	9626.205720	0.166430
17.000000	10867.271686	10866.880009	0.391677
18.000000	12183.377253	12183.077468	0.299785
19.000000	13574.688853	13574.198675	0.490178
20.000000	15041.206485	15040.708346	0.498139
21.000000	16582.930150	16582.349400	0.580750
22.000000	18199.859847	18199.081798	0.778050
23.000000	19891.995577	19891.352705	0.642872
24.000000	21659.337339	21658.181123	1.156216
25.000000	23501.885133	23500.587416	1.297717
26.000000	25419.638960	25417.990691	1.648270
27.000000	27412.598820	27411.237038	1.361782
28.000000	29480.764711	29478.497535	2.267177

In [21]:

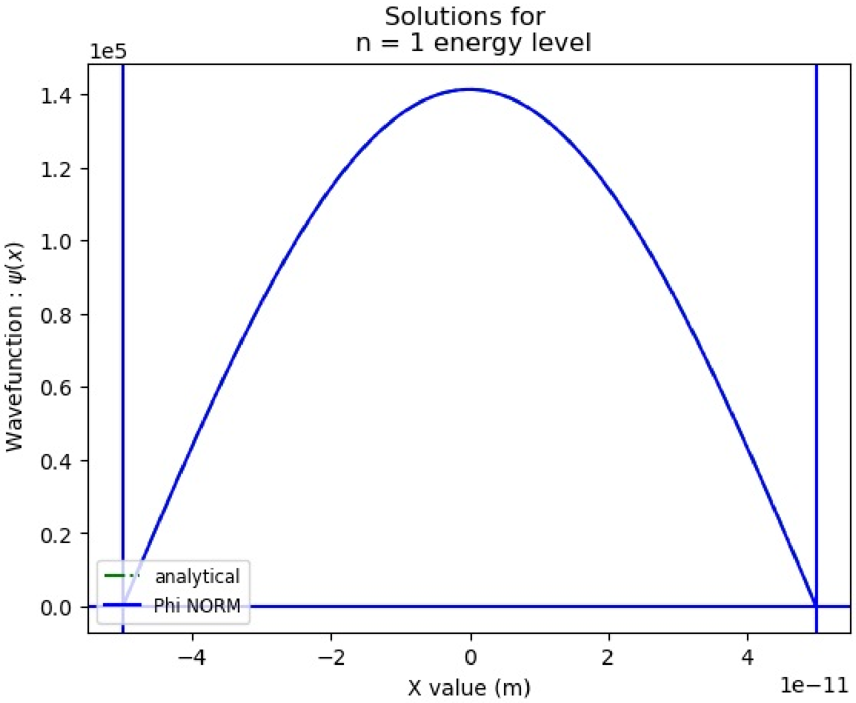
```
#PLOTTING FURTHER NORMALISED WAVEFUNCTIONS
```

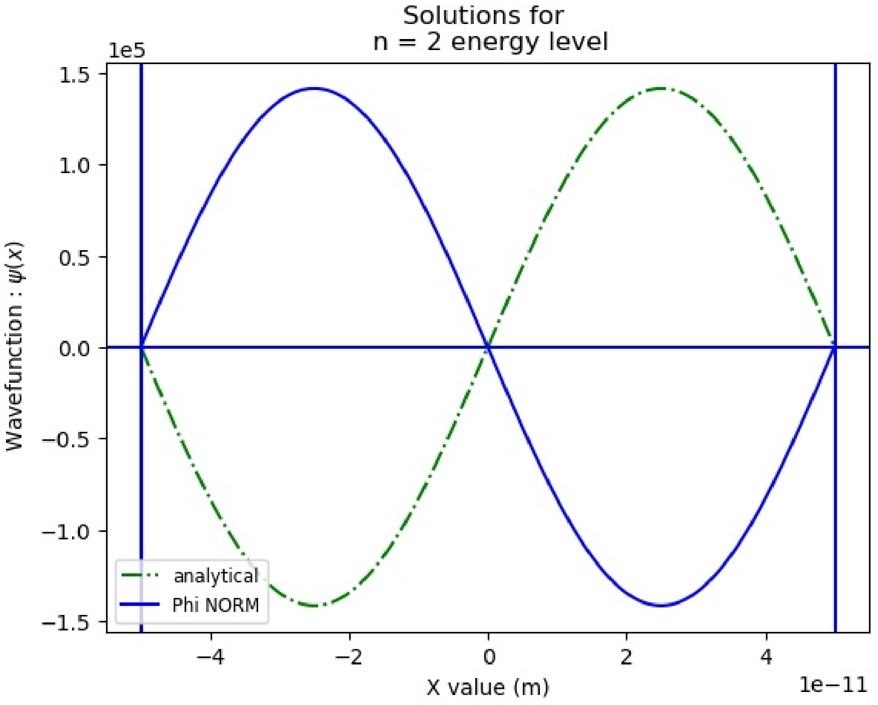
FIRST FEW EXCITED STATES

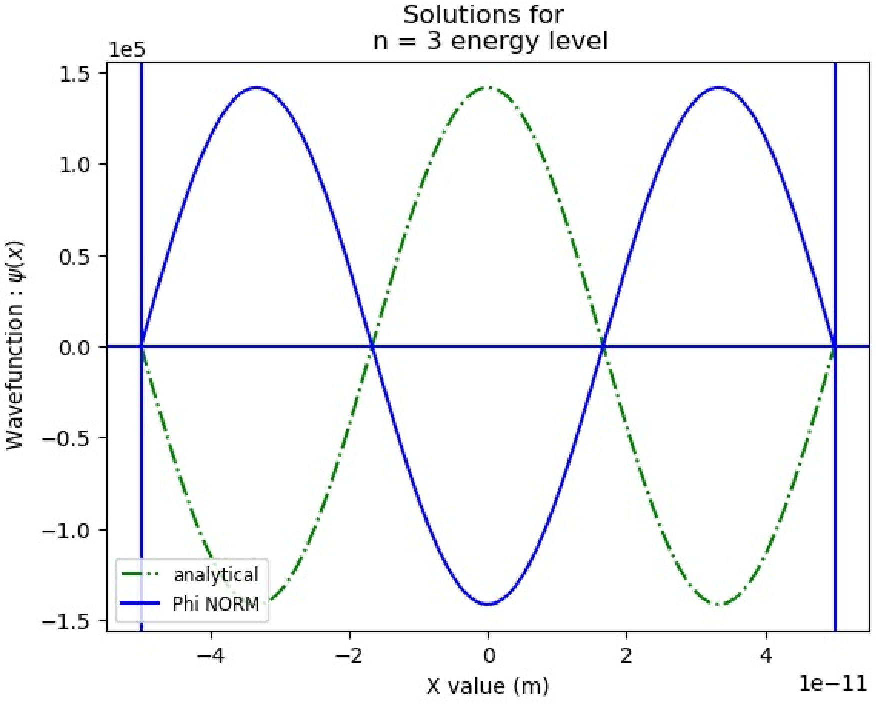
Below we can see the plots for the wavefunction for the first 5 energy states. Note that these wavefunctions have already been normalised using the normalisation() function. As shown in the legend 2 solutions for the wavefunciton: numerical and analytical have been plotted.

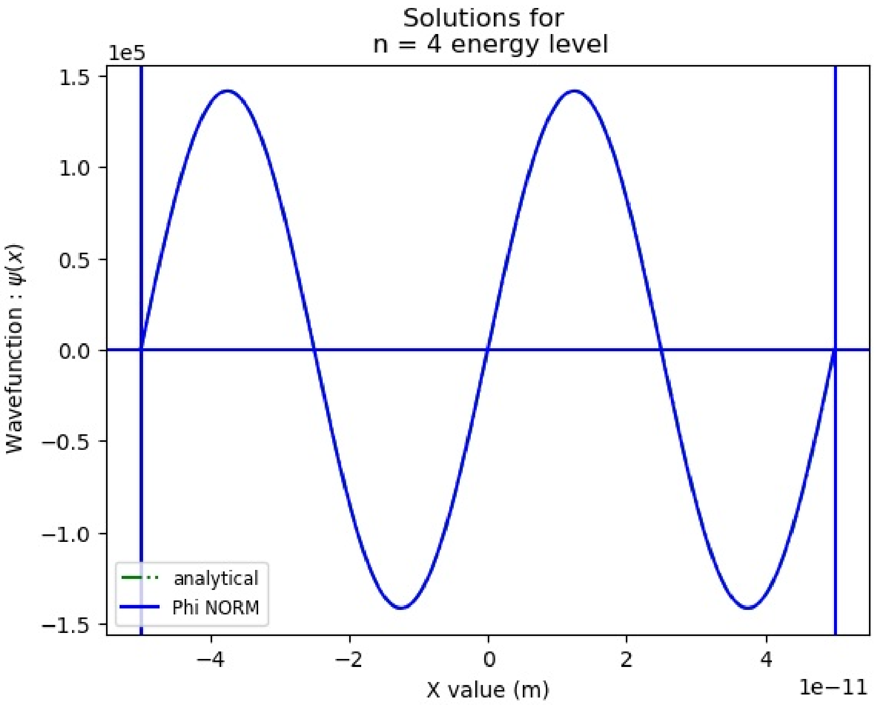
In [22]:
```
Energy_sols = energy_values[1] # Setting a variable equal to the energy solutions
```

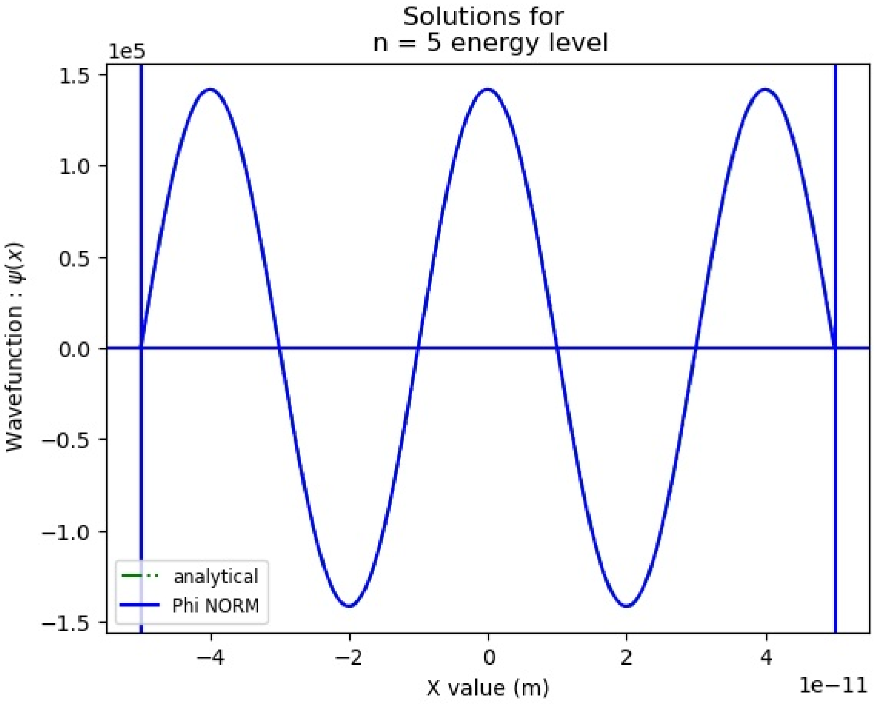
In [23]:
```
# Create the first plot
plt.figure()
calcplot(Energy_sols[0], x_points2, 1, V_pot0, well_class1)
plt.title("Solutions for \n n = 1 energy level")
plt.show()


# Create the second plot
plt.figure()
calcplot(Energy_sols[1], x_points2, 2, V_pot0, well_class1)
plt.title("Solutions for \n n = 2 energy level")
plt.show()

# Create the third plot
plt.figure()
calcplot(Energy_sols[2], x_points2, 3, V_pot0, well_class1)
plt.title("Solutions for \n n = 3 energy level")
plt.show()

# Create the fourth plot
plt.figure()
calcplot(Energy_sols[3], x_points2, 4, V_pot0, well_class1)
plt.title("Solutions for \n n = 4 energy level")
plt.show()

# Create the fifth plot
plt.figure()
calcplot(Energy_sols[4], x_points2, 5, V_pot0, well_class1)
plt.title("Solutions for \n n = 5 energy level")
plt.show()
```




Solutions for
n = 2 energy level
1e5
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
Wavefunction : $\psi(x)$
analytical
Phi NORM
−4
−2
0
2
4
1e−11
X value (m)


Solutions for
n = 3 energy level
1e5
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
Wavefunction : $\psi(x)$
analytical
Phi NORM
−4
−2
0
2
4
1e−11
X value (m)





Symmetry of the wavefunctions

Note that the analytical and numeric solutions to the wavefunctions are either: a.) Perfectly alligened or Symmetric or b.) oppite, reflected or Antisymmetric

This is not a mistake in our method but rather an intrinsic property of the wavefunction. Because of the characteristics of the potential well that they describe, wave functions can have either odd or even symmetry. Particularly, depending on whether the particle it describes is a boson or a fermion, the wave function must also be symmetric or antisymmetric for symmetric potential wells (i.e., potential wells that are identical on each side of the centre).

The wave function must also be symmetric or antisymmetric around the centre, as is the case, for instance, in the infinite square well, where the potential is symmetric about the well's centre. This indicates that the wave function is either oddly symmetric (i.e., it changes when reflected around the well's centre) or evenly symmetric. (i.e., it changes sign when reflected about the center of the well).

The requirement for odd or even symmetry arises from the boundary conditions imposed on the wave function at the edges of the potential well. These boundary conditions dictate that the wave function must be continuous and differentiable at the edges of the well, and this in turn leads to the requirement for odd or even symmetry.

HIGHER ENERGY STATES

Does your method also work to find eigenstates for large n? Check this explicitly for at least two states in the range 18 ≤ n ≤ 28.

plotting higher energy states

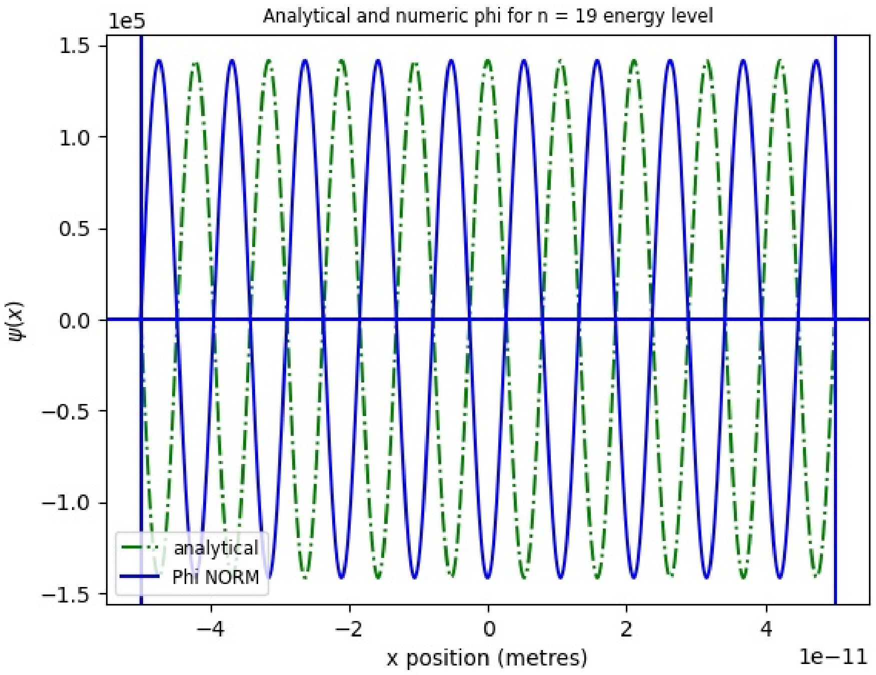
Now we calculate and plot, on an appropriately labelled plot, the ground state wavefunction.

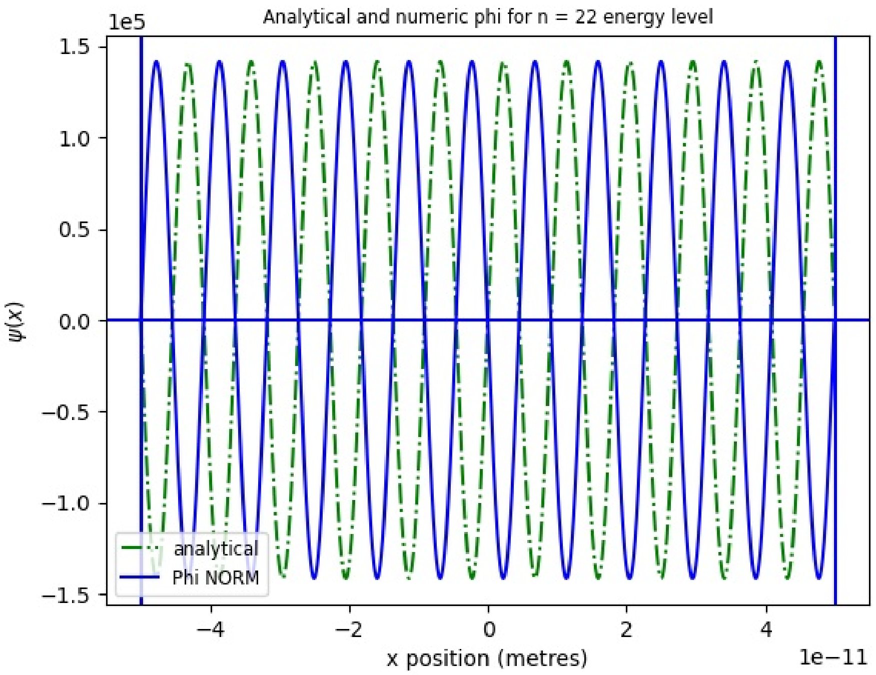
In [24]:

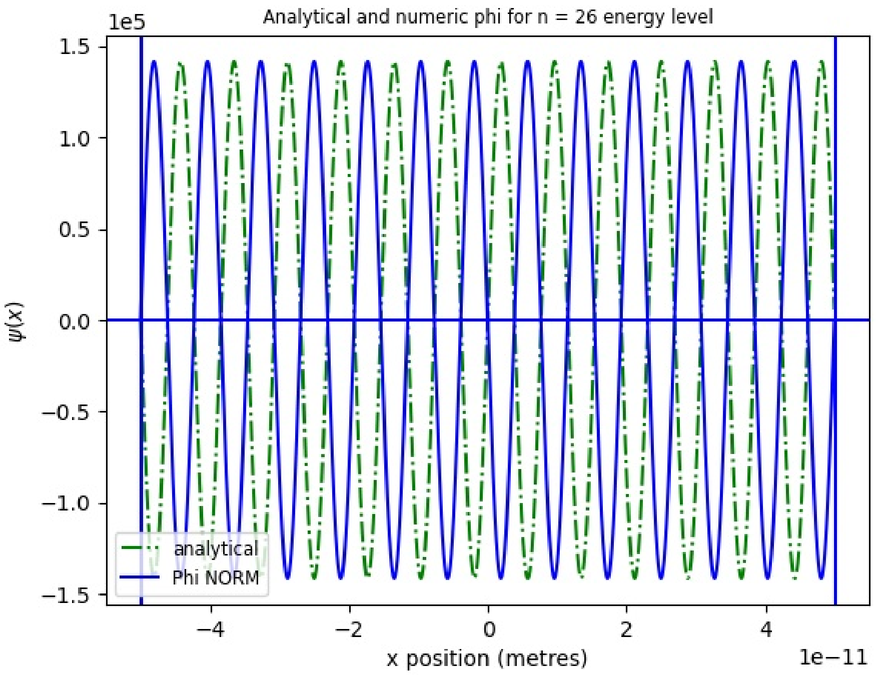
```
#Checking that the code works for states in the range specified
#Checking for n=19, n=22, n=26

# Plot 1
fig, ax = plt.subplots()
calcplot(Energy_sols[18],x_points2,19,V_pot0,well_class1)
ax.set_title("Analytical and numeric phi for n = 19 energy level", fontsize='small')
ax.set_xlabel("x position (metres)")
ax.set_ylabel("$\psi(x)$")

# Plot 2
fig, ax = plt.subplots()
calcplot(Energy_sols[21],x_points2,22,V_pot0,well_class1)
ax.set_title("Analytical and numeric phi for n = 22 energy level", fontsize='small')
ax.set_xlabel("x position (metres)")
ax.set_ylabel("$\psi(x)$")

# Plot 3
fig, ax = plt.subplots()
calcplot(Energy_sols[25],x_points2,26,V_pot0,well_class1)
ax.set_title("Analytical and numeric phi for n = 26 energy level", fontsize='small')
ax.set_xlabel("x position (metres)")
ax.set_ylabel("$\psi(x)$")
'''

plt.subplot(grid[1,1])
calcplotplotter(Energy_sols[27],x_points2,28,V_pot0,well_class1)
plt.title("Numerical Wavefunction \n against theoretical wavefunction for \n n = 28 energy level", fontsize = '

'''
```

Out[24]:

```
'\n\nplt.subplot(grid[1,1])\ncalcplotplotter(Energy_sols[27],x_points2,28,V_pot0,well_class1)\nplt.title("Numer
ical Wavefunction \n against theoretical wavefunction for \n n = 28 energy level", fontsize = \'small\');\n\n'
```







Eigenstates for large values of n:

As we can see, our method is also successfull at finding eigenstates for larger values of n in the range : $18 \leq n \leq 28$.

3.4 EXTENSION TO 3D

lab script text

3.4 Extension to 3D

A true quantum dot needs to be considered in all three dimensions, which means the Schroedinger's equation becomes a partial differential equation. Thankfully, we can separate the dimensions and solve for each individually. The analytical solution for the final energy states of a cubic dot with side length d is then expressed as: 2 2 2 π2ħ2 Enx,ny,nz = (nx + ny + nz)2md2 (9) where nx,ny and nz are three discrete quantum numbers - one corresponding to each dimension. Since we can vary each quantum number separately, the

3D case gives us many more energy levels than 1D approximation. Calculate the first 10 energy levels in a 3D quantum dot, present them in a suitable format and comment on their values. One of the most desired properties of quantum dots is the tunable wavelength of the emitted light. Considering the first transition only (from E112 to E111), discuss how the light emitted by a quantum dot changes with its physical properties. (You may wish to supplement your comments with an appropriate graph).

STILL TO DO :

Calculate the first 10 energy levels in a 3D quantum dot, present them in a suitable format and COMMENT ON THEIR VALUES.

In [25]:

```python
# Dimensions of the cubic dot
d = 10e-9   # Side length in meters

# Calculate constant
constant = (np.pi**2 * h_bar**2) / (2 * m_electron * d**2)

# Calculate first 10 energy levels
energy_levels = []
for n_x in range(1, 4):
    for n_y in range(1, 4):
        for n_z in range(1, 4):
            n = np.sqrt(n_x**2 + n_y**2 + n_z**2)
            energy = n**2 * constant
            energy_levels.append((n_x, n_y, n_z, energy))

# Print energy levels
print("Energy Levels for a 3D Quantum Dot:")
print("----------------------------------")
print("n_x  n_y  n_z  Energy (eV)")
for i, level in enumerate(energy_levels[:10]):
    print("{:3d}  {:3d}  {:3d}  {:10.6f}".format(level[0], level[1], level[2], level[3] / 1.6e-19))
```

```
Energy Levels for a 3D Quantum Dot:
----------------------------------
n_x  n_y  n_z  Energy (eV)
  1    1    1    0.011296
  1    1    2    0.022593
  1    1    3    0.041420
  1    2    1    0.022593
  1    2    2    0.033889
  1    2    3    0.052716
  1    3    1    0.041420
  1    3    2    0.052716
  1    3    3    0.071543
  2    1    1    0.022593
```

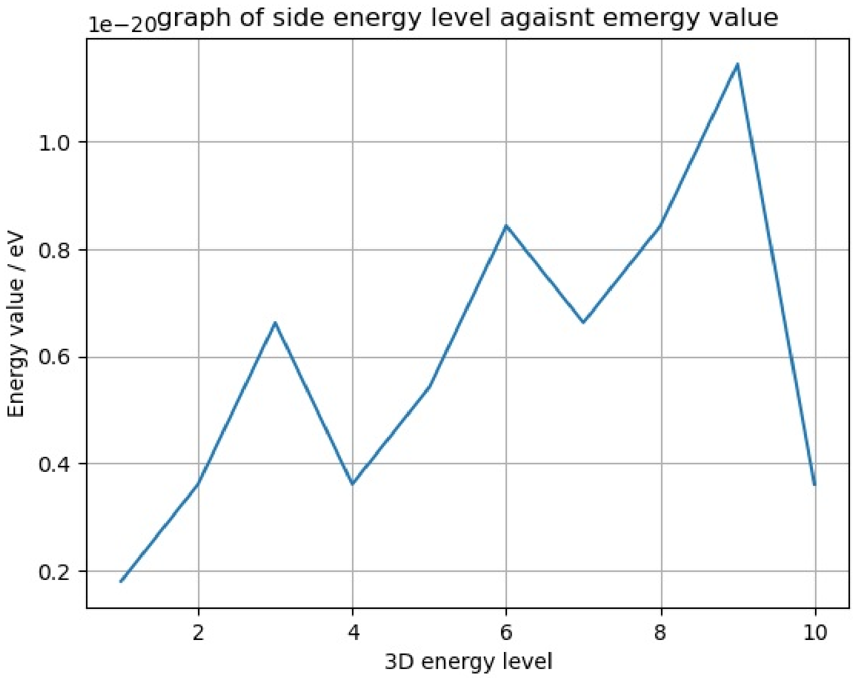
In [ ]:

In [26]:

```python
#graph of 3d energy level agaisnt emergy value
#note assumption of energy level rising as such (in order of energy)
#n=1 : (1,1,1)
#n=2 : (1,1,2)
#n=3 : (1,1,3)
```

In [27]:

```python
#PLOT OF N against energy value

lvs = [1,2,3,4,5,6,7,8,9,10]
vals = []
plt.plot
xv = energy_levels
for i in range(10):
    #print(xv[i][3])
    vals.append(xv[i][3])

plt.plot(lvs,vals)
plt.xlabel('3D energy level')
plt.ylabel('Energy value / eV')
plt.title('graph of side energy level agaisnt emergy value')
plt.grid()
```



Discussion

The energy levels are quantized and discrete, which is a normal for quantum systems. The electron's quantum state in each of the three dimensions is represented by the quantum numbers nx, ny, and nz, which determine the energy levels. According to the formula E nx,ny,nz = (nx + ny + nz)2 *h bar2 / (2* m electron * d2), where h bar is the reduced Planck constant and m electron is the mass of the electron, the energy levels rise as the quantum numbers rise.

In [ ]:

```

```

How the light emitted by a quantum dot changes with its physical properties:

One of the most desired properties of quantum dots is the tunable wavelength of the emitted light. Considering the first transition only (from E(1,1,2) to E(1,1,1), discuss how the light emitted by a quantum dot changes with its physical properties. (You may wish to supplement your comments with an appropriate graph).

Answer: According to the equation E=hc/, where E is the energy difference, h is Planck's constant, c is the speed of light, and is the wavelength of light, the energy difference between the E(1,1,2) and E(1,1,1) states determines the wavelength of the emitted light. Hence, by modifying a quantum dot's physical attributes like size, shape, and composition, the wavelength of light it emits may be changed.

The emission wavelength shifts towards the blue when the size of a quantum dot shrinks due to closer-spaced energy levels. Quantum confinement is the term for this phenomenon. The quantum dot's form can also influence the emission wavelength.

For instance, a quantum dot of the same size with a cylindrical shape will emit light at a different wavelength than one with a spherical shape.

Additionally, the emission wavelength might be impacted by the quantum dot's composition. For instance, the energy levels of the quantum dot can be altered, affecting the emission wavelength, by altering the type and quantity of dopants or the makeup of the host material.

The energy difference between E(1,1,2) and E(1,1,1) is:

energy of E(2,1,1) = 0.022593 eV

energy of E(1,1,1) = 0.011296 eV

energy difference = 0.022593 eV

since:

E=hc/λ

corresponding wavelength shift: 548.7 nm

We can illustrate the relation between energy level and wavelength by plotting the energy level agaisnt the

To summarize: Quantum numbers and the physical characteristics of the quantum dot, such as the length and the electron mass (m_electron), both affect the energy of the transition. For instance, if we lengthen the quantum dot's side, the energy levels and transition energy will both drop, resulting in a longer wavelength of light being emitted. Conversely, if we decrease the mass of the electron, the energy levels will increase, and the energy of the transition will increase as well, leading to a shorter wavelength of the emitted light.

4 infinite unsquare well

4 part 1 : HARMONIC POTENTIAL

To create a harmonic potential within an infinite square well, an appropriate value of V0 was be selected, with around 700e being a good starting point. The eigenvalue and wavefunction results will be compared to the analytical solutions for a pure harmonic potential, and any similarities or differences between the results shall be discussed and interpreted.

The equation for the harmonic potential is given by:

V(x) = V0x^2/a^2

In [28]:

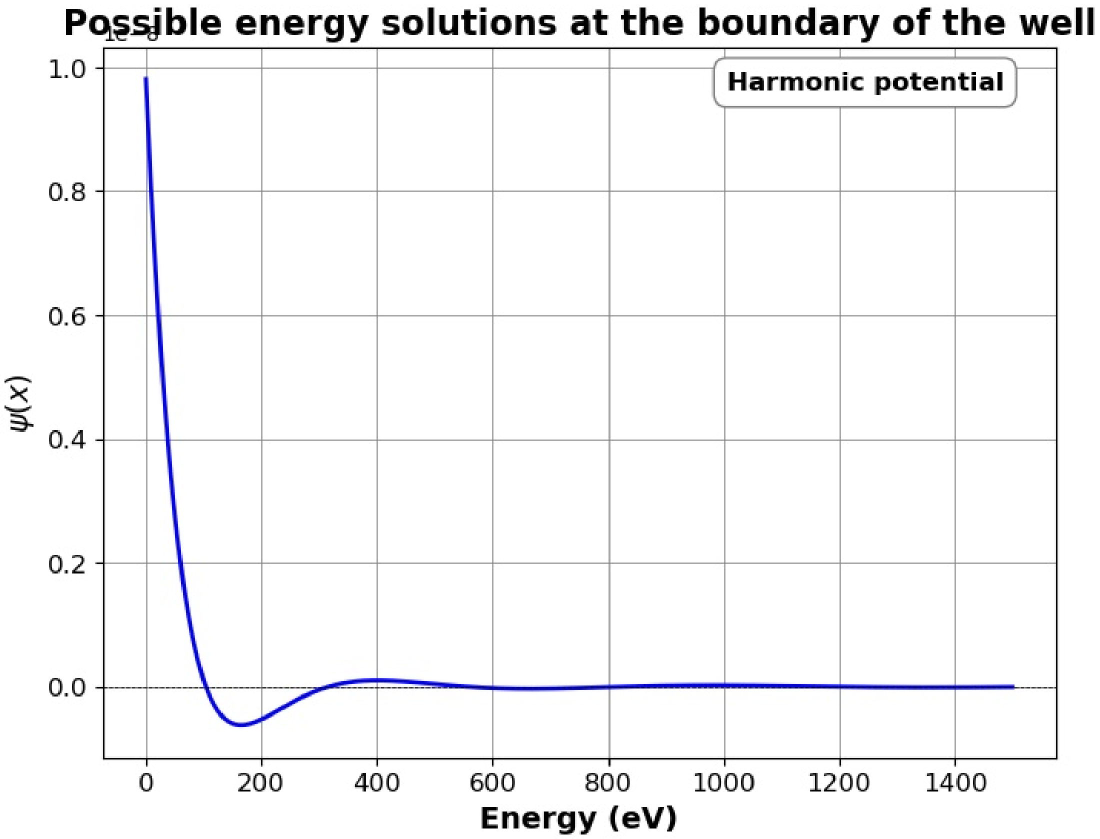
```
def new_V(p):
    """Return an array of potential energy values for a harmonic potential well.

    Args:
        x_points (array-like): An array of x values to find the potential energy for.

    Returns:
        array: An array of points representing the potential energy for a range of x values.
    """
    pot_arr = []  #creating empty aray to fill with potentials
    for x in p:
        i = potential_V0*(x**2/half_width**2)
        pot_arr.append(i)
    return pot_arr #returning array containing values for potential energy
```

In [ ]:

```

```

In [29]:

```
# Calculate new energy guesses with a harmonic potential
V_NEW = new_V(x_points)  # Create a harmonic potential
new_potential_guess = energy_calculator(energy_array_II, frame_levels_II, energy_values_II, V_NEW)  # Calculate

# Plot the points at which there is indication of an energy level
fig, ax = plt.subplots(figsize=(8, 6))  # Create a figure with a custom size
ax.plot(energy_values_II, new_potential_guess[1], color='blue', linewidth=2)  # Plot energy values against corr
ax.set_xlabel("Energy (eV)", fontsize=14, fontweight='bold')  # Add X-axis label with custom font size and weig
ax.set_ylabel("$\psi(x)$", fontsize=14, fontweight='bold')  # Add Y-axis label with custom font size and weight
ax.set_title("Possible energy solutions at the boundary of the well", fontsize=16, fontweight='bold')  # Add pl
ax.grid(True, color='gray', linestyle='-', linewidth=0.5)  # Add grid lines to the plot with custom color, line
ax.tick_params(axis='both', which='major', labelsize=12)  # Set tick label font size
ax.axhline(y=0, color='black', linestyle='--', linewidth=0.5)  # Add a horizontal line at y=0 with custom color
ax.text(0.8, 0.95, "Harmonic potential", transform=ax.transAxes, fontsize=12, fontweight='bold', ha='center', v
plt.show()  # Show the plot

# Output the energy guesses in a data frame
print("Energy guesses ; possible energy solutions:")
print(new_potential_guess[0].data.to_string(index=False))  # Output energy guesses without row index
```



```
Energy guesses ; possible energy solutions:
 Energy Level  Energy Guess 1  Energy Guess 2
          1.0      105.300860      109.598854
          2.0      311.690544      324.412607
          3.0      543.352436      565.530086
          4.0      817.134670      850.487106
```

In [ ]:

In [30]:

```
#old block:2

well_class2 = "Harmonic" # Setting well class to harmonic for energy_finder1() loop
Frame_levels3 = np.linspace(0,2,3) # Taking the first 3 energy levels
new_potential_energy = energy_finder1(Frame_levels3, V_NEW, new_potential_guess[2],energy_values_II,well_class2
new_potential_energy[0] # Outputting as a data frame
```

Out[30]:

Pandas dataframe showing energy values output from energy_finder1() function

N(level)	Predicted / eV	Calculated (eV)	Difference (eV)
1.000000	103.285768	103.918605	0.632837
2.000000	309.857305	317.008801	7.151496
3.000000	516.428841	551.873355	35.444515

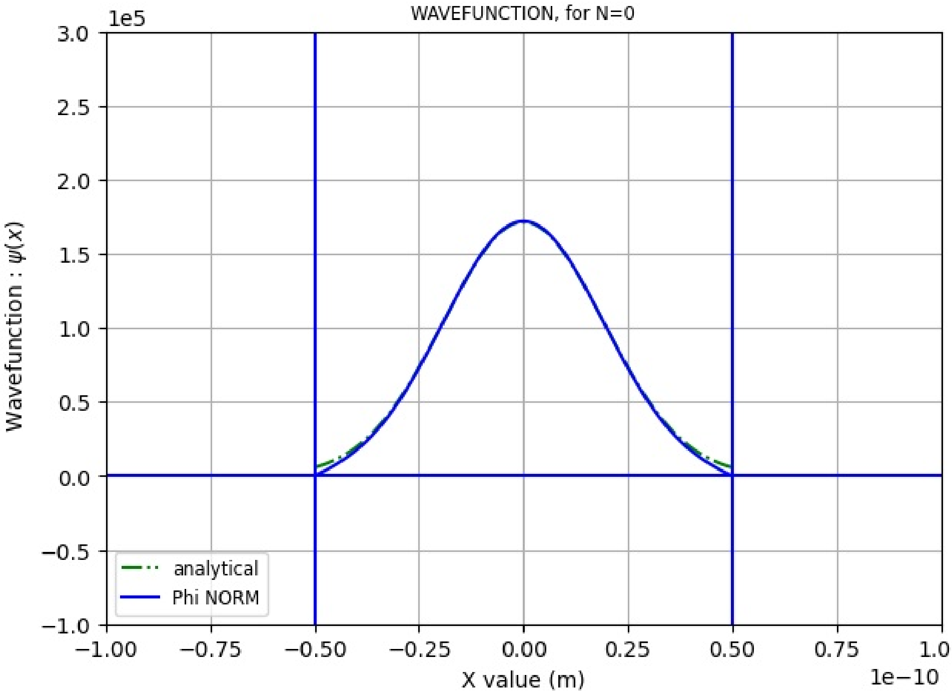
In [ ]:

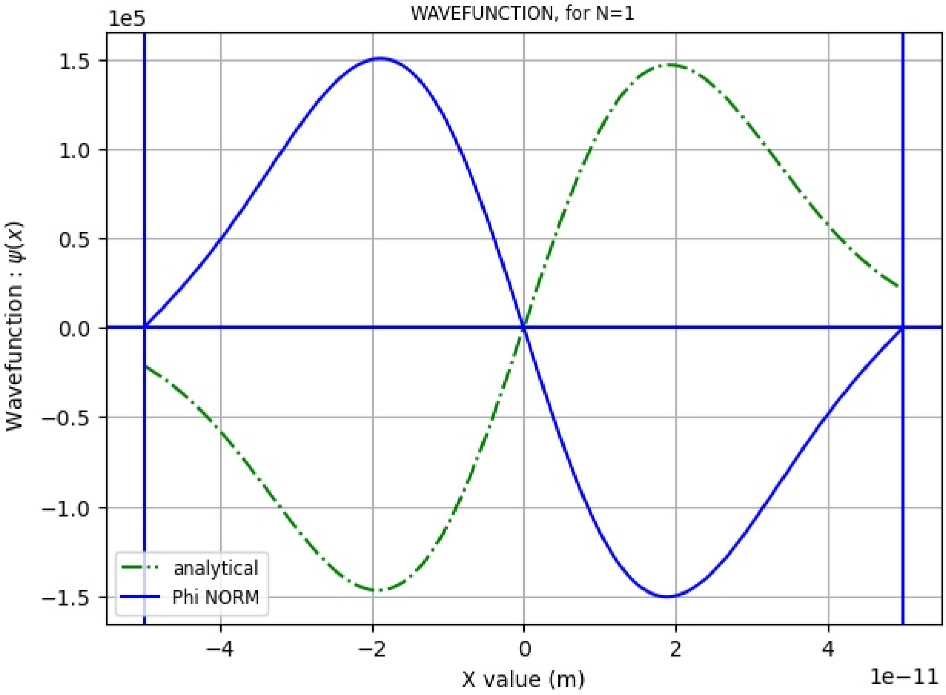
In [ ]:

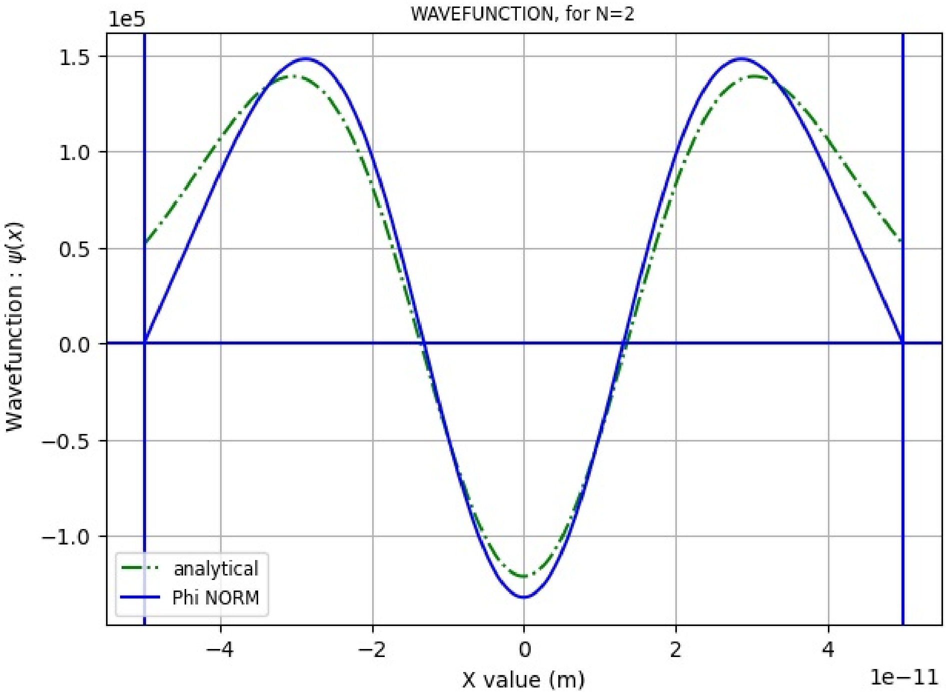
In [31]:

```
#nunu block:0.5


#NEW BLOCK--> 0.5
well_class2 = "Harmonic" # Setting well class to harmonic for energy_finder1() loop
Frame_levels3 = np.linspace(0, 2, 3) # Taking the first 3 energy levels
new_potential_energy = energy_finder1(Frame_levels3, V_NEW, new_potential_guess[2], energy_values_II, well_clas
new_potential_energy[0] # Outputting as a data frame

Energy_sols2 = new_potential_energy[1]

# Creating the first plot
plt.figure(figsize=(7, 5))
# Plotting the numerical and theoretical wavefunctions for the first energy level
calcplot(Energy_sols2[0], x_points2, 0, V_NEW, well_class2)
plt.title("WAVEFUNCTION, for N=0", fontsize='small')
plt.grid(True)
plt.xlim(-10e-11, 10e-11)
plt.ylim(-1e5, 3e5)

# Creating the second plot
plt.figure(figsize=(7, 5))
```

```
# Plotting the numerical and theoretical wavefunctions for the second energy level
calcplot(Energy_sols2[1], x_points2, 1, V_NEW, well_class2)
plt.title("WAVEFUNCTION, for N=1", fontsize='small')
plt.grid(True)

# Creating the third plot
plt.figure(figsize=(7, 5))
# Plotting the numerical and theoretical wavefunctions for the third energy level
calcplot(Energy_sols2[2], x_points2, 2, V_NEW, well_class2)
plt.title("WAVEFUNCTION, for N=2", fontsize='small')
plt.grid(True)


#the 3 lowest eigenstatees
```



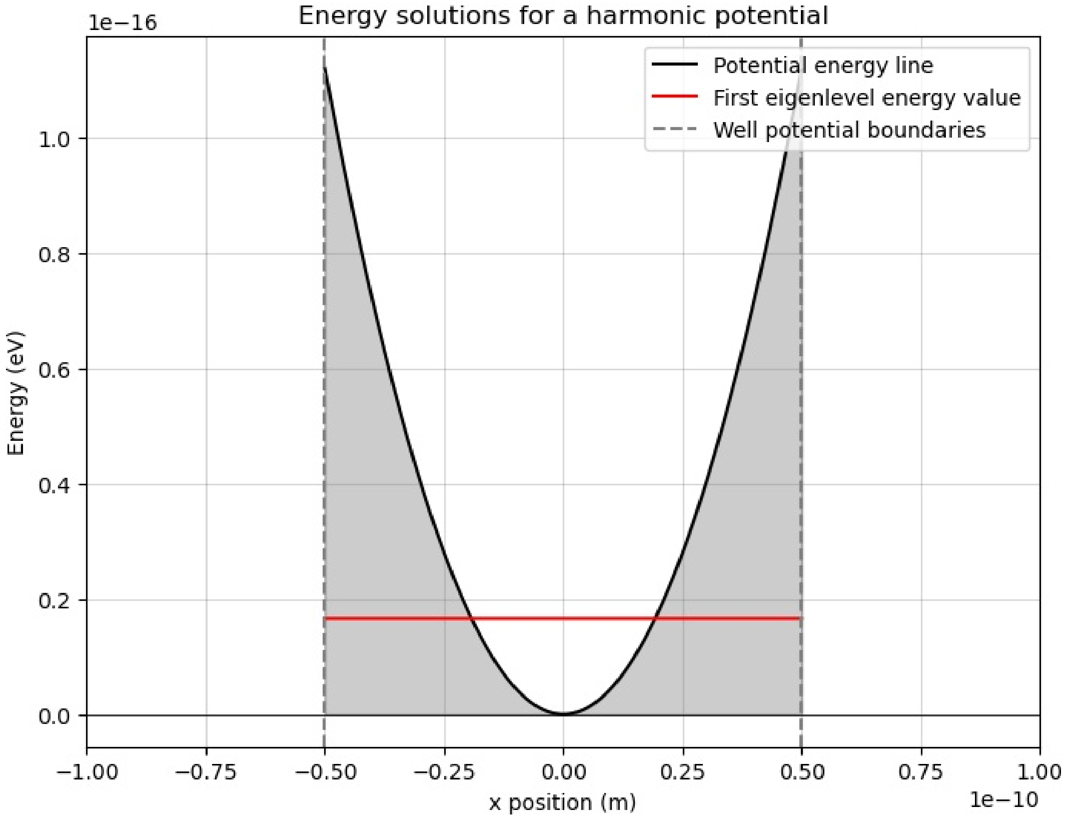


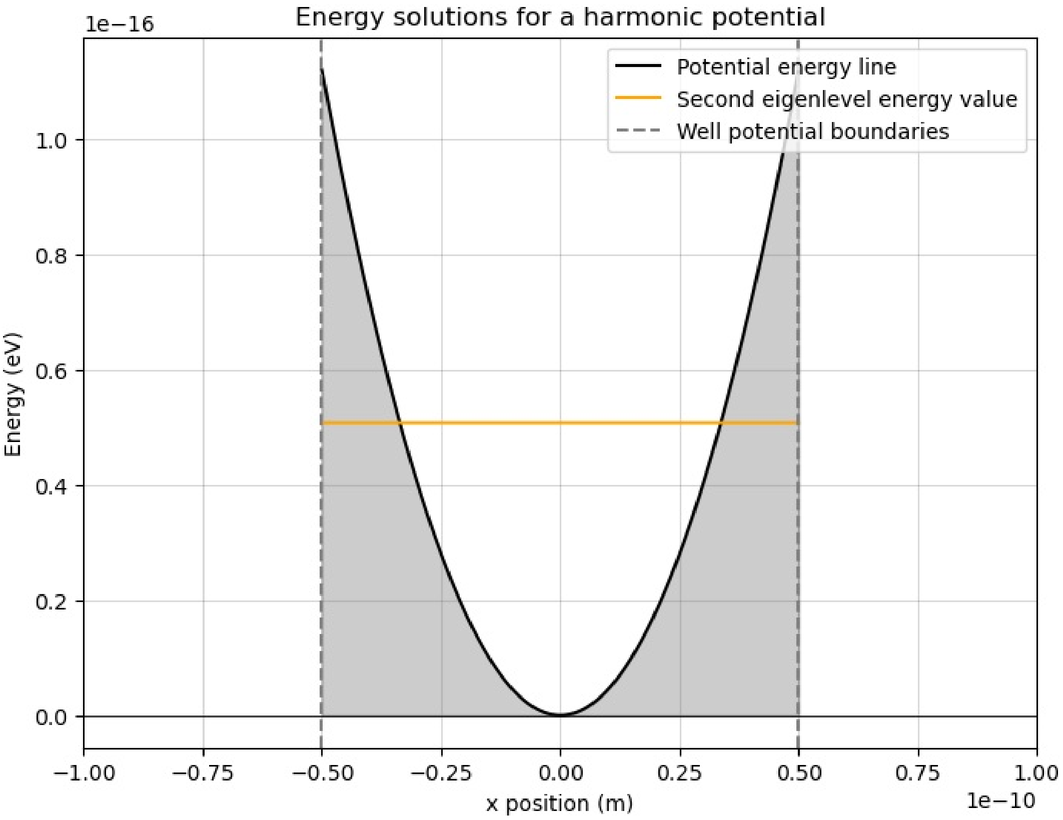


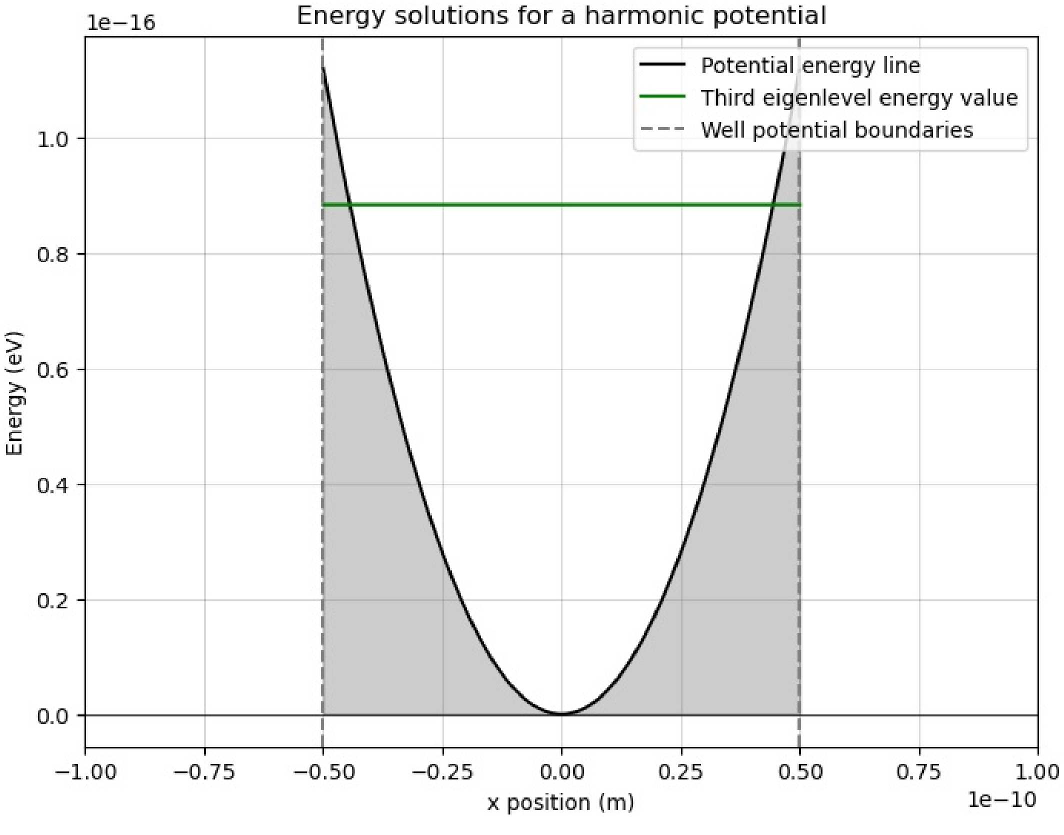
In [ ]:

In [32]:

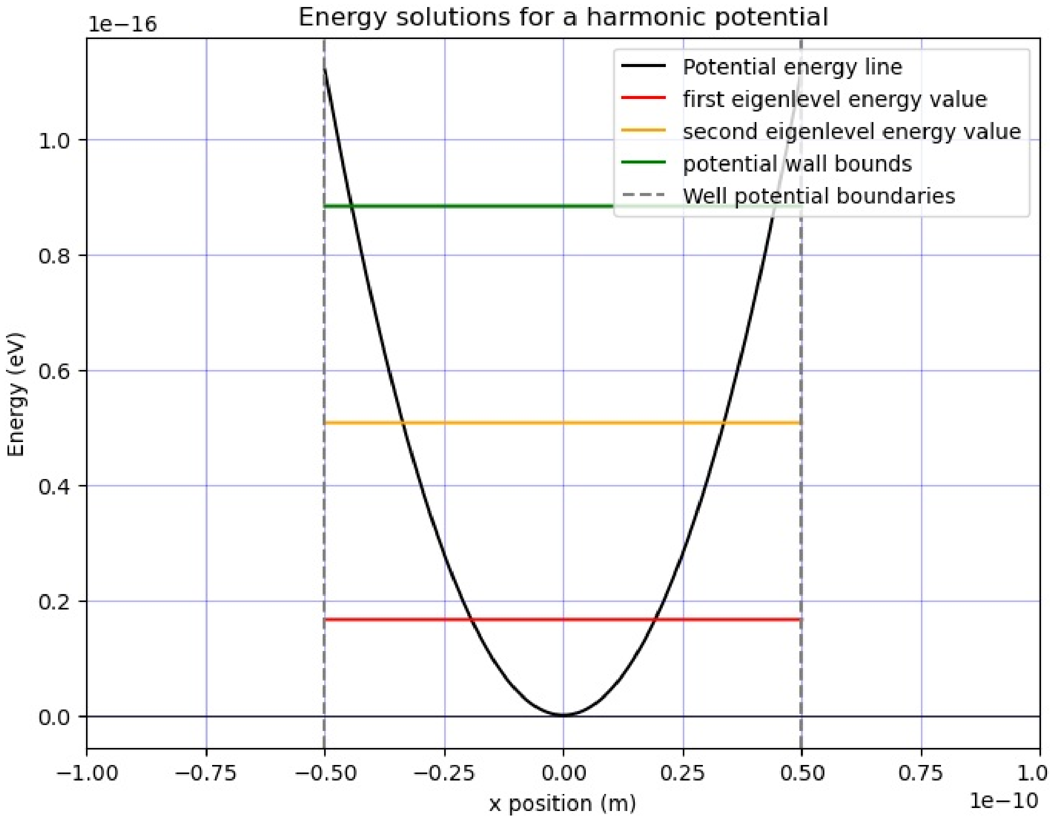
```
# Plotting the harmonic potential and the first 4 energy levels:


# plot the harmonic potential energy and the first eigenlevel energy value
fig, ax = plt.subplots(figsize=(8, 6))
ax.plot(x_points, V_NEW, color='black', label='Potential energy line')
ax.plot(x_points2, [Energy_sols2[0]]*len(x_points2), color='red', label='First eigenlevel energy value')
ax.set_xlim([-10e-11, 10e-11])
ax.axvline(x=x_start, c='grey', linestyle='--', label='Well potential boundaries')
ax.axvline(x=x_end, c='grey', linestyle='--')
ax.axhline(y=0, c='black', linewidth=0.5)
ax.fill_between(x_points, V_NEW, alpha=0.2, color='black')
ax.grid(color='grey', alpha=0.3)
ax.set_xlabel('x position (m)')
ax.set_ylabel('Energy (eV)')
ax.set_title('Energy solutions for a harmonic potential')
ax.legend(loc='upper right')
plt.show()

# plot the harmonic potential energy and the second eigenlevel energy value
fig, ax = plt.subplots(figsize=(8, 6))
ax.plot(x_points, V_NEW, color='black', label='Potential energy line')
ax.plot(x_points2, [Energy_sols2[1]]*len(x_points2), color='orange', label='Second eigenlevel energy value')
ax.set_xlim([-10e-11, 10e-11])
ax.axvline(x=x_start, c='grey', linestyle='--', label='Well potential boundaries')
ax.axvline(x=x_end, c='grey', linestyle='--')
ax.axhline(y=0, c='black', linewidth=0.5)
ax.fill_between(x_points, V_NEW, alpha=0.2, color='black')
ax.grid(color='grey', alpha=0.3)
ax.set_xlabel('x position (m)')
ax.set_ylabel('Energy (eV)')
ax.set_title('Energy solutions for a harmonic potential')
ax.legend(loc='upper right')
plt.show()

# plot the harmonic potential energy and the third eigenlevel energy value
fig, ax = plt.subplots(figsize=(8, 6))
ax.plot(x_points, V_NEW, color='black', label='Potential energy line')
ax.plot(x_points2, [Energy_sols2[2]]*len(x_points2), color='green', label='Third eigenlevel energy value')
ax.set_xlim([-10e-11, 10e-11])
ax.axvline(x=x_start, c='grey', linestyle='--', label='Well potential boundaries')
ax.axvline(x=x_end, c='grey', linestyle='--')
ax.axhline(y=0, c='black', linewidth=0.5)
ax.fill_between(x_points, V_NEW, alpha=0.2, color='black')
ax.grid(color='grey', alpha=0.3)
ax.set_xlabel('x position (m)')
ax.set_ylabel('Energy (eV)')
ax.set_title('Energy solutions for a harmonic potential')
ax.legend(loc='upper right')
plt.show()


# plot the harmonic potential energy and the first 4 energy levels
fig, ax = plt.subplots(figsize=(8, 6))
ax.plot(x_points, V_NEW, color='black', label='Potential energy line')
ax.plot(x_points2, [Energy_sols2[0]]*len(x_points2), color='red', label='first eigenlevel energy value')
ax.plot(x_points2, [Energy_sols2[1]]*len(x_points2), color='orange', label='second eigenlevel energy value')
ax.plot(x_points2, [Energy_sols2[2]]*len(x_points2), color='green', label='potential wall bounds')

#DISPLAYING ALL levels on one plot

ax.set_xlim([-10e-11, 10e-11])
ax.axvline(x=x_start, c='grey', linestyle='--', label='Well potential boundaries')
ax.axvline(x=x_end, c='grey', linestyle='--')
ax.axhline(y=0, c='black', linewidth=0.5)
ax.grid(color='blue', alpha=0.3)
ax.set_xlabel('x position (m)')
ax.set_ylabel('Energy (eV)')
ax.set_title('Energy solutions for a harmonic potential')
ax.legend(loc='upper right')
plt.show()
```




Energy solutions for a harmonic potential
1e−16
Energy (eV)
0.0
0.2
0.4
0.6
0.8
1.0
x position (m)
1e−10
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
Potential energy line
Second eigenlevel energy value
Well potential boundaries



Energy solutions for a harmonic potential
1e−16
1.0
0.8
0.6
0.4
0.2
0.0
Energy (eV)
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
1e−10
x position (m)
Potential energy line
Third eigenlevel energy value
Well potential boundaries



In [33]:
```
#Finite square Well
```

In [34]:
```
def V_coc(init_pot, xvals, sizewall):  #single well potential (ref:double well potential)

    """
    Calculates the potential energy for a finite square well.

    Inputs:
    V02: float, initial value of the potential
    x_points: array-like, range of x values to evaluate the potential across
    a: float, half the width of the well

    Outputs:
    V_2: array-like, potential energy values evaluated at each x value
    """
    pot_arr = []  # initialize the potential energy array with empty array pot_arr
    for p in xvals:
        if abs(p) <= sizewall/2:  # limit for the potential
            x = 0  # potential is 0 within the well
        else:
            x = init_pot
        pot_arr.append(x)  # append the solution to the empty array pot_arr
    return pot_arr  # potential array
```

In [35]:
```
pots = V_coc(potential_V0_finite,x_points,half_width) # The finite well potential
```

In [36]:
```
#calculating energies for this potential
```

In [36]:

```
#calculating energies for this potential
```

4 The infinite “unsquare” well

Part 2 - Finite Square well

Now we will investigate the energy and wavefunction solutions for the finite square well. The potential in a finite sqaure well can be dfescribed by the following equation:

V_f = \begin{cases} 0 & \text{if } -halfwidth/2 \leq x \leq +halfwidth/2, \\ V_0 & \text{if } |x| > halfwidth/2, \end{cases}[7]

In [37]:

```
new_potential_guess2 =  energy_calculator(energy_array_II,Frame_levels3,energy_values_II, pots)
```

In [38]:

```
# Calculating the enrgies for this potential
well_class3 = "Finite"
finpot = energy_finder1(frame_levels_II, pots, new_potential_guess2[2],energy_values_II, well_class3)
finpot[0] # Outputting the energies
```

Out[38]:

Pandas dataframe showing energy values output from energy_finder1() function

N(level)	Predicted / eV	Calculated (eV)	Difference (eV)
1	n/a	85.660784	n/a
2	n/a	328.854834	n/a
3	n/a	670.179230	n/a
4	n/a	981.956546	n/a

No analytical solutions explanation

Note that for the finite square well there are no analytical solutions and only numerical solutions.

The reason why there are no analytical solutions and only numeric solutions when using the secant and Runge Kutta methods to solve the Schrödinger equation for a finite square well is that the potential function (V_f) for a finite square well is not symmetrical. Unlike the infinite square well, the potential function for the finite square well varies within the well, this then makes it impossible to solve the Schrödinger equation analytically.

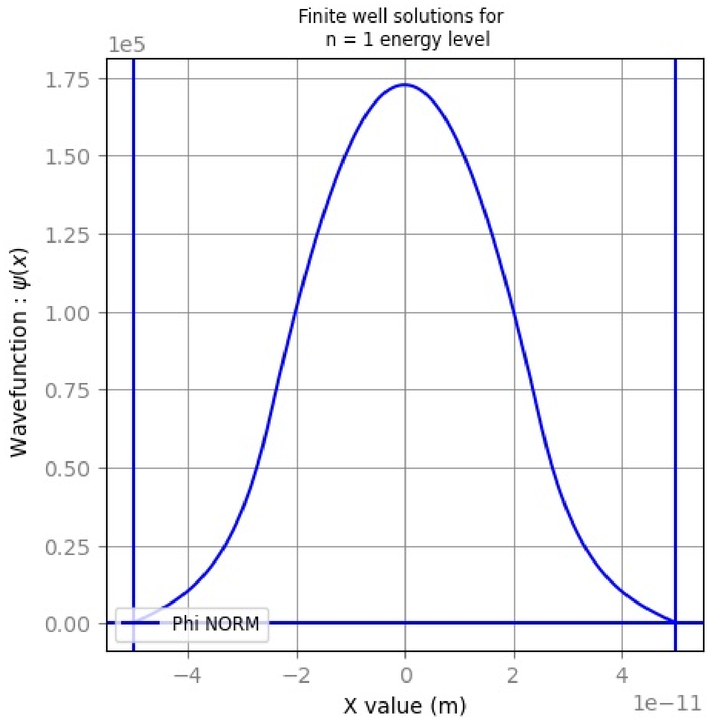
The secant and Runge Kutta methods are numerical methods used to solve the Schrödinger equation for non-analytical potential functions, and they work by iteratively approximating the solution until a certain level of accuracy is achieved. These methods are used to solve the Schrödinger equation numerically for the finite square well, as there are no analytical solutions that can be obtained for this case.

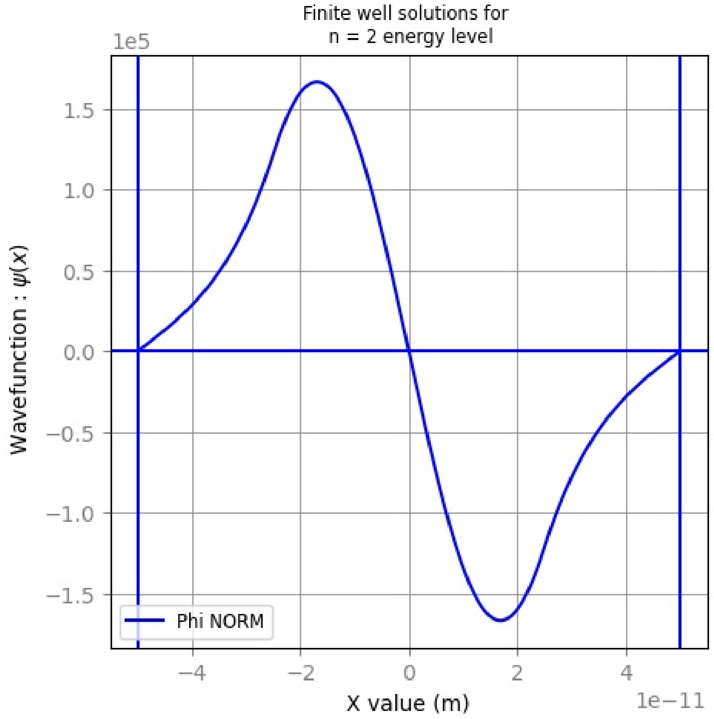
Therefore, while analytical solutions exist for the infinite square well, they cannot be obtained for the finite square well due to the varying nature of the potential function. Instead, numerical methods such as the secant and shooting methods must be used to solve the Schrödinger equation for the finite square well.

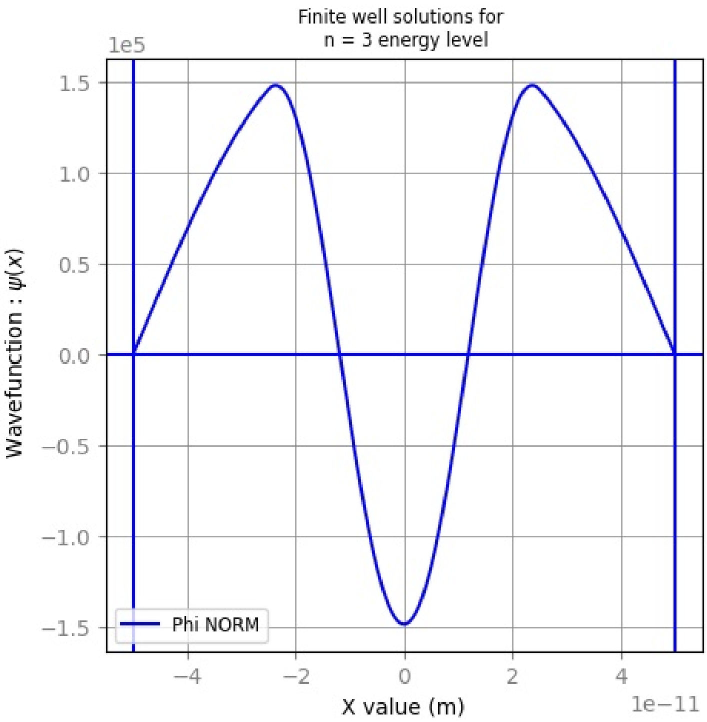
In [39]:

```
# Plotting the wavefunctions of the energy solutions:
Energy_sols3 = finpot[1]

# set color for the axis
color = 'gray'

# plot for n = 1
plt.figure(figsize=(5,5))
calcplot(Energy_sols3[0], x_points2, 1, pots, well_class3)

# add gridlines and change axis color
plt.grid(color=color, linestyle='-', linewidth=0.5)
plt.tick_params(axis='x', colors=color)
plt.tick_params(axis='y', colors=color)

plt.title("Finite well solutions for \n n = 1 energy level", fontsize = 'small')
plt.show()

# plot for n = 2
plt.figure(figsize=(5,5))
calcplot(Energy_sols3[1], x_points2, 2, pots, well_class3)

# add gridlines and change axis color
plt.grid(color=color, linestyle='-', linewidth=0.5)
plt.tick_params(axis='x', colors=color)
plt.tick_params(axis='y', colors=color)

plt.title("Finite well solutions for \n n = 2 energy level", fontsize = 'small')
plt.show()
```

```
# plot for n = 3
plt.figure(figsize=(5,5))
calcplot(Energy_sols3[2], x_points2, 3, pots, well_class3)

# add gridlines and change axis color
plt.grid(color=color, linestyle='-', linewidth=0.5)
plt.tick_params(axis='x', colors=color)
plt.tick_params(axis='y', colors=color)

plt.title("Finite well solutions for \n n = 3 energy level", fontsize = 'small')
plt.show()
```







In [ ]:

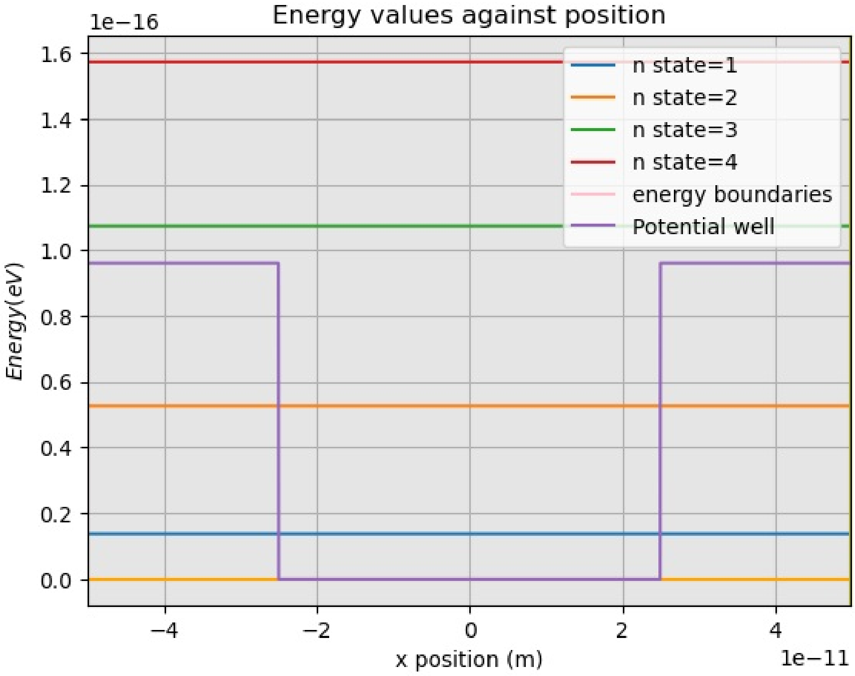
In [ ]:

In [40]:

```python
# Plotting the harmonic potential and the first 4 energy levels:
# Plotting the harmonic potential and energy eigenfunctions
plt.figure()

# Shade the allowed energy states within the potential energy well
plt.axvspan(x_start, x_end, color='gray', alpha=0.2)

# Plot the energy eigenfunctions and energy levels
plt.plot(x_points2, [Energy_sols3[0]]*len(x_points2), label='n state=1 ')
plt.plot(x_points2, [Energy_sols3[1]]*len(x_points2), label='n state=2')
plt.plot(x_points2, [Energy_sols3[2]]*len(x_points2), label='n state=3')
plt.plot(x_points2, [Energy_sols3[3]]*len(x_points2), label='n state=4')

# Add vertical lines at the positions of the energy levels

# Set x and y-axis limits, add vertical and horizontal lines at well boundaries and y=0 respectively
plt.xlim([-10e-11, 10e-11])
plt.axvline(x=x_start, c='pink', label="energy boundaries")
plt.axhline(y=0, c='orange')
plt.axvline(x=x_end, c='y')
plt.xlim(-.5e-10,.5e-10)
# Add grid lines, x and y-axis labels, and title
plt.grid()
plt.xlabel("x position (m)")
plt.ylabel("$Energy (eV)$")
plt.title("Energy values against position")
plt.plot(x_points, pots, label='Potential well')
# Add an arrow going up at x=0
# Add a legend and display the plot
plt.legend(loc="upper right")
```

Out[40]:

```
<matplotlib.legend.Legend at 0x7fa61e220490>
```



Difference between n=3 and n=2,1

As seen from 'Energy values against position' graph the n=3 eigenstate is the only state which is above the potenial barier. This explains the difference in shape seen in the graph displaying the wavefuncion. We would expect teh n=4 state to follow the same trend if we where to compute its wavefunction.

Similarities and differences between infinte and finate potential well plots

The lower energy levels for the finite well resemble the levels for the infite square well despite having a more crooked shape.

The reasoning behind this difference is likely quantum tunnelling, which gives the wavefunctions an exponential component. The wavefunction shape strays from the shape of the infinite square well solution at higher energies, but at lower energies (as een with n=1 and n=2) the chance of tunnelling is greatly reduced. In other words, the wavefunction solution becomes more and more like the infinite square well as the energy level falls.

CONCLUSION

Investigating variations in energy levels and wavefunctions under various contraints within a quantum system has been very insightfull. A lot can be learned by considering the physics underlying the shapes of graphs representing both analytical and numerical solutions discussed in this notebook.

In [ ]:

In [ ]:

In [ ]: