

Final Project Report PHYS 5319 MM3

Solving 1-D Schrödinger equation by the shooting method

Author:

Shashank Kumbhare

University of Texas at Arlington, TX

1001433909

shashank.kumbhare@mavs.uta.edu

Project Advisor:

Dr. Qiming Zhang

University of Texas at Arlington, TX

zhang@uta.edu

August 2021

Abstract

The goal of this project is to solve the 1-D Schrödinger equation for a 1-D double-well potential by a numerical shooting method using 4th order Runge-Kutta method. This report contains the analysis of the problem, the methodology used, the python code used, the results, and discussion.

The Schrödinger equation is the fundamental quantum mechanical equation. However, only for a handful of cases it can be solved analytically, thus requiring a numerical method to solve for systems where no analytical solution exists.

The shooting method is a numerical method to solve differential equations such as the Schrödinger equation where the boundary conditions are known and certain parameters to solve the equations must be found. In this project, we study the parameter energy (E) as the eigenvalue of the system. The shooting method used in this project is the double-shooting method. In double-shooting method, we take the left & right boundary conditions as initial conditions of the equation as the starting points and shoot from both left & right side with defined initial values. Then we observe whether the solutions from left & right shooting comes close enough at some matching point. If this is the case, we then refine it further to a specified accuracy.

In this project, the Schrödinger equation is solved for a 1-D double-well potential and later for a simple harmonic oscillator around x_0 , where x_0 is one of the two bottoms of the double-well. To demonstrate the method's accuracy, we use a simple harmonic oscillator around $x=0$ as a test potential to compare the numerical solutions to their analytical counterparts. Overall, the results match the analytical solutions proving the shooting method to be a useful tool for obtaining numerical solutions for the Schrödinger equation.

Acknowledgement

Thanks to the course instructor Dr. Qiming Zhang for his wonderful classes and his simplistic explanations. I really appreciate his guidance and help for the completion of this project.

Contents

1	Introduction	5
1.1	Project aims	5
1.2	Approach	5
2	Methodology	7
2.1	Information from Schrödinger Equation	7
2.2	The Fourth Order Runge Kutta Method	7
2.3	The Double-Shooting Method	8
3	Flowchart of shooting method by 4 th -RK method	10
4	Results.....	11
4.1	Testing code with Simple Harmonic Oscillator (SHO) at $x = 0$ $Vx = 12kx^2$	11
4.2	Double well Potential: En and ψnx $Vx = 2x^2 - 8x^4$	13
4.3	Simple Harmonic Oscillator at $x = 2$ $Vx = 12(x - 2)^2$	15
5	Discussion	17
6	Code.....	17

1 Introduction

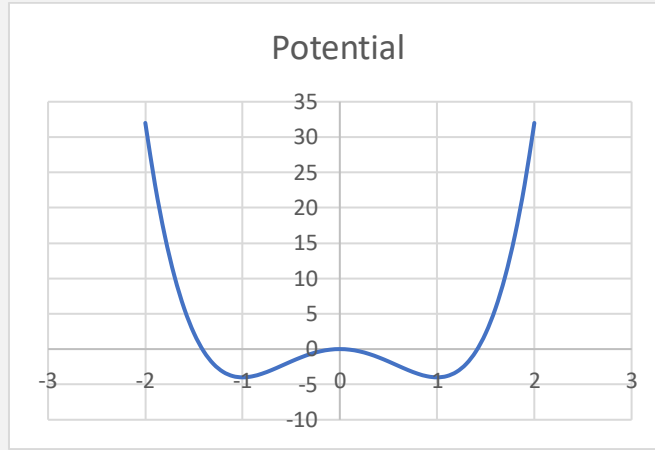
The goal of this project is to solve the 1-D Schrödinger equation for a 1-D double-well potential by a numerical shooting method using 4th order Runge-Kutta method.

$$\left[-\frac{1}{2} \frac{d^2}{dx^2} + V(x)\right]\psi(x) = E\psi(x) \quad (1)$$

where the atomic unit (a.u.) is used and the potential $V(x)$ is given by,

$$V(x) = ax^4 - bx^2 \quad (2)$$

with $a = 2$ and $b = 8$.



1.1 Project aims

The motivation for this project is to test out the double shooting method implemented with Python and to see how it can be used for solving the Schrödinger equation. The aims of this project are as following:

- Find all the energy eigenvalues (E) below zero ($E < 0$). (at least 4 significant figures)
- Plot the lowest 2 states (orthonormal wave functions) together with $V(x)$ in $(-2.5 < x < 2.5)$.
- Discuss that if $V(x)$ is approximated as a simple harmonic oscillator around x_0 , where x_0 is one of the two bottoms, what kind of structures for the energy eigenvalues and wavefunctions will be. Compare (qualitatively) with your results from a) and b).

1.2 Approach

- Python is a well-suited language for scientific programming with clear, easily readable syntax and add-on packages for many computing needs. This project uses the *NumPy* & *SciPy* libraries and the *matplotlib* plotting environment. It is free and operating system independent, making it easily transferable.
- The Schrödinger equation takes the form,

$$\frac{d^2}{dx^2} \psi(x) = 2[V(x) - E]\psi(x) \quad (3)$$

Now this is the standard form of a 2nd order differential equation. Knowing the Schrödinger equation and both boundary conditions, the solutions for arbitrary energies can be computed with a numerical integration method. The method used here is 4th order Runge-Kutta method. But the issue here is that there is a parameter E . Now, here we can use the shooting method which allows us to find both the parameter and the solution of the differential equation.

- c) In this project, double shooting method is used. The idea is to guess the value of E and start solving the differential equation using a numerical integration method from both the boundaries (left & right) assuming boundary conditions as the initial values of the function (hence the name double shooting). And the integration will progress till a point (matching point) where we check if the values of the function match. If they match, we keep the values of E , else we keep trying different values of E . In this way, we will get those values of E which will satisfy our differential equation.
- d) The matching energy values can then be refined with either an interpolation method, or with shooting as often as needed.
- e) To test the accuracy of the double shooting method, we use a simple harmonic oscillator around $x = 0$ as a test potential to compare the numerical solutions to their analytical counterparts.

2 Methodology

The Schrödinger equation is the fundamental equation in quantum mechanics. However, it is not possible to solve it analytically for most quantum mechanical systems. Therefore, the methodology involves the double shooting approaches of the shooting method aiming to solve the case of the double-well potential. This section introduces the Schrödinger equation and the numerical methods used.

2.1 Information from Schrödinger Equation

The potential $V(x)$ given by eq. (2), where $a = 2$ and $b = 8$ which takes the form as following,

$$V(x) = 2x^4 - 8x^2 \quad (4)$$

The minima of eq. (4) can be calculated analytically or by using *python's SciPy* library.

From the Schrödinger Equation eq. (3) and $V(x)$ we can have the following information,

- The boundary conditions of wave-function $\psi(x)$,
 $\psi(-\infty) = 0$ and $\psi(\infty) = 0$
- The energy of the system cannot be lower than the minima of the $V(x)$ which can be calculated analytically or by using *python's SciPy* library.

```
import scipy.optimize
def V(x): return 2*x**4-8*x**2
max_x = scipy.optimize.fmin(lambda x: V(x), 0)

Optimization terminated successfully.
Current function value: -8.000000
```

2.2 The Fourth Order Runge Kutta Method

Say a 2nd order differential equation is given as following,

$$\frac{d^2y}{dx^2} = f_2(x, y, \frac{dy}{dx}) \quad (5)$$

which can be written as two coupled first order equations,

$$\frac{dz}{dx} = f_2(x, y, z) \quad (6)$$

$$\frac{dy}{dx} = f_1(x, y, z) = z \quad (7)$$

Then the solution is given by,

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad (8)$$

$$z = z_i + \frac{1}{6}(l_1 + 2l_2 + 2l_3 + l_4) \quad (9)$$

where,

$$k_1 = h \times f_1(x_i, y_i, z_i)$$
$$l_1 = h \times f_2(x_i, y_i, z_i)$$

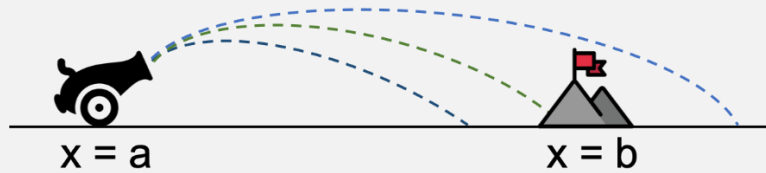
$$\begin{aligned}
k_2 &= h \times f_1 \left(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}, z_i + \frac{l_1}{2} \right) \\
l_2 &= h \times f_2 \left(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}, z_i + \frac{l_1}{2} \right) \\
k_3 &= h \times f_1 \left(x_i + \frac{h}{2}, y_i + \frac{k_2}{2}, z_i + \frac{l_2}{2} \right) \\
l_3 &= h \times f_2 \left(x_i + \frac{h}{2}, y_i + \frac{k_2}{2}, z_i + \frac{l_2}{2} \right) \\
l_4 &= h \times f_1(x_i + h, y_i + k_3, z_i + l_3) \\
l_4 &= h \times f_2(x_i + h, y_i + k_3, z_i + l_3)
\end{aligned} \tag{10}$$

2.3 The Double-Shooting Method

In the initial value problems, we can start at the initial value and march forward to get the solution. But this method is not working for the boundary value problems, because there are not enough initial value conditions to solve the ODE to get a unique solution. Therefore, the shooting methods was developed to overcome this difficulty.

Shooting Method

The shooting methods are developed with the goal of transforming the ODE boundary value problems to an equivalent initial value problem, then we can solve it using one of the integration methods such as 4th order Runge Kutta method. It involves finding solutions to the initial value problem for different initial conditions until one finds the solution that also satisfies the boundary conditions of the boundary value problem. In layman's terms, one "shoots" out trajectories in different directions from one boundary until one finds the trajectory that "hits" the other boundary condition.



The name of the shooting method is derived from analogy with the target shooting as shown in the above figure, we shoot the target and observe where it hits the target, based on the errors, we can adjust our aim and shoot again in the hope that it will hit close to the target. We can see from the analogy that the shooting method is an iterative method.

Shooting Method to solve Eigenvalue problem

The shooting method can also be used to solve eigenvalue problems. Consider the time-independent Schrödinger equation eq. (3),

In quantum mechanics, one seeks normalizable wavefunctions function $\psi_n(x)$, and their corresponding energies subject to the boundary conditions,

$$\psi(-\infty) = 0 \text{ and } \psi(\infty) = 0$$

To apply it, first note some general properties of the Schrödinger equation:

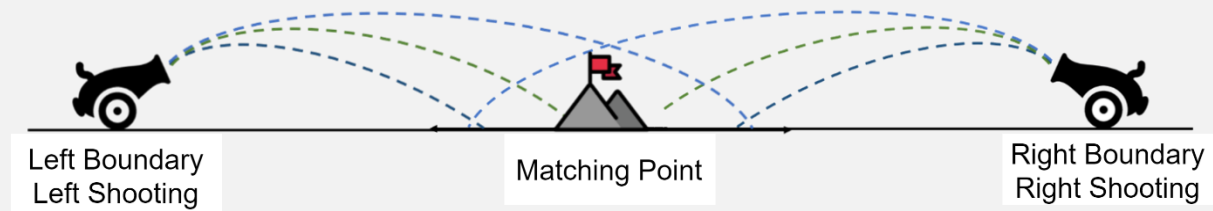
- If $\psi_n(x)$ is an eigenfunction, so is $C\psi_n(x)$ for any nonzero constant C .
- The n^{th} excited state $C\psi_n(x)$ has roots where $\psi_n(x) = 0$.
- For even n , the n^{th} excited state is symmetric and nonzero at the origin.
- For odd n , the n^{th} excited state is antisymmetric and thus zero at the origin.

To find the n^{th} excited state $\psi_n(x)$ and its energy E_n , the shooting method is then to:

1. Guess some energy E_n .
2. Integrate the Schrödinger equation (For example, using 4th order Runge Kutta method in this project.) with $\psi(-\infty) = 0$ as initial condition.
3. Check if $\psi(\infty) = 0$.
 - a. If yes, save $\psi_n(x)$ and corresponding the value of E_n .
 - b. If no, guess another value of E_n and repeat from step 1.

Double Shooting Method

This method divides the interval over which a solution is sought into 2 intervals, solves an initial value problem in each of the smaller intervals, and imposes additional matching condition at the junction of those 2 intervals to ensure continuity and to form a solution on the whole interval. The method constitutes a significant improvement in distribution of nonlinearity and numerical stability over single shooting methods.



The shooting method used in this project is the double-shooting method. Here, we take the left & right boundary conditions as initial conditions of the equation as the starting points and shoot from both left & right side with defined initial values ($\psi(-\infty) = 0$ and $\psi(\infty) = 0$). Then we observe whether the matching condition satisfies at some matching point (either left or right of centre but not at the centre though).

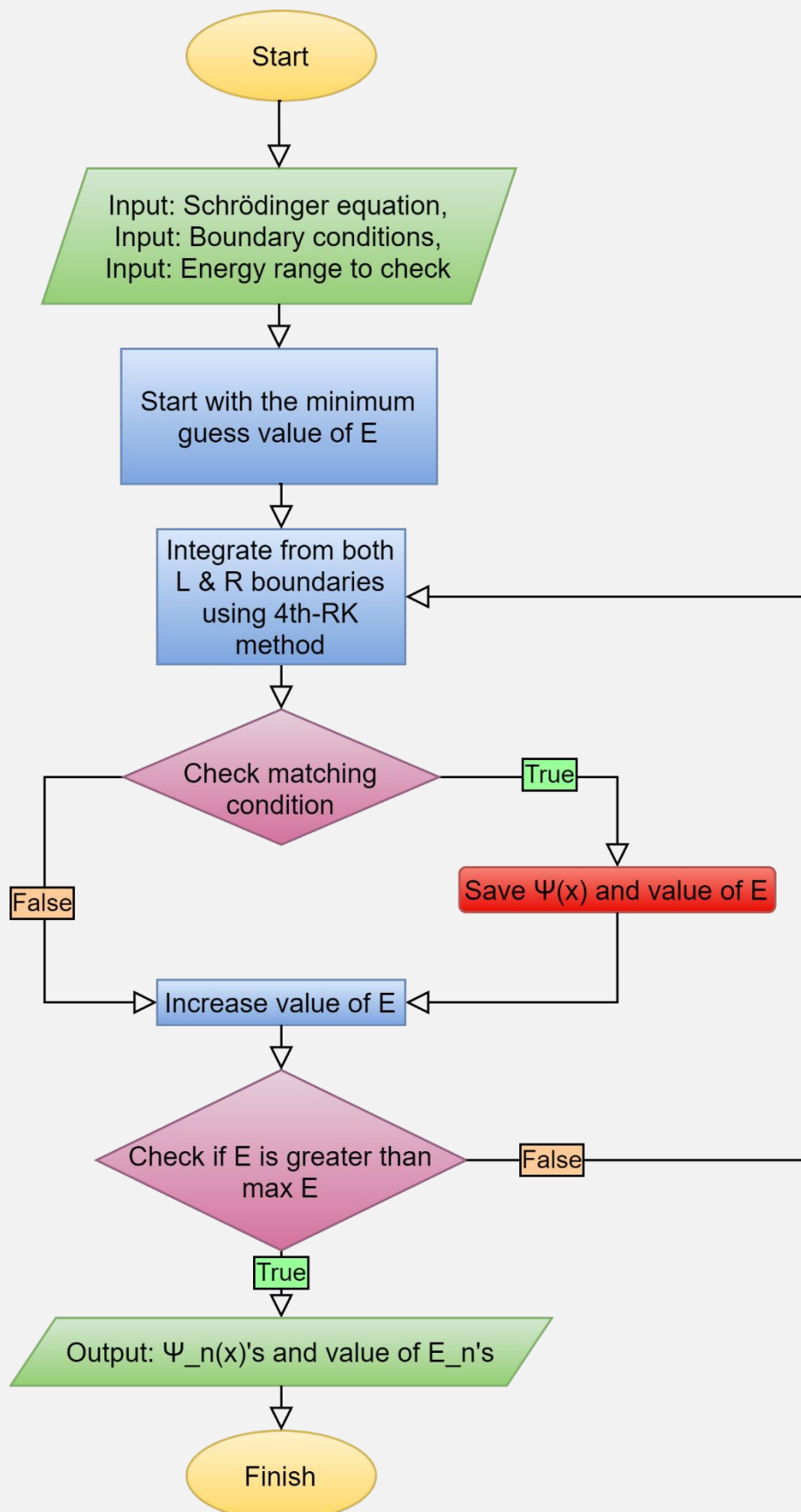
The matching condition used in this project is,

$$\frac{\psi'_L}{\psi_L} = \frac{\psi'_R}{\psi_R} \quad (11)$$

For double shooting method, to find the n^{th} excited state $\psi_n(x)$ and its energy E_n ,

1. Guess some energy E_n .
2. Integrate the Schrödinger equation from left side with $\psi(-\infty) = 0$ as initial condition till the matching point.
3. Integrate the Schrödinger equation from right side with $\psi(\infty) = 0$ as initial condition till the matching point.
4. Check if matching condition satisfies.
 - a. If yes, save $\psi_n(x)$ and corresponding the value of E_n .
 - b. If no, guess another value of E_n and repeat from step 1.

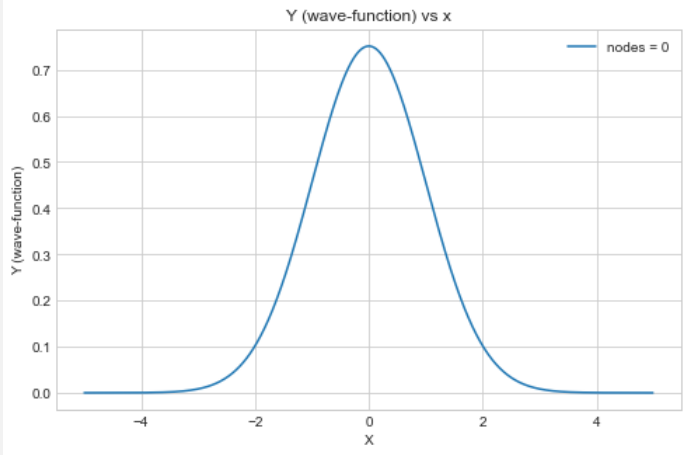
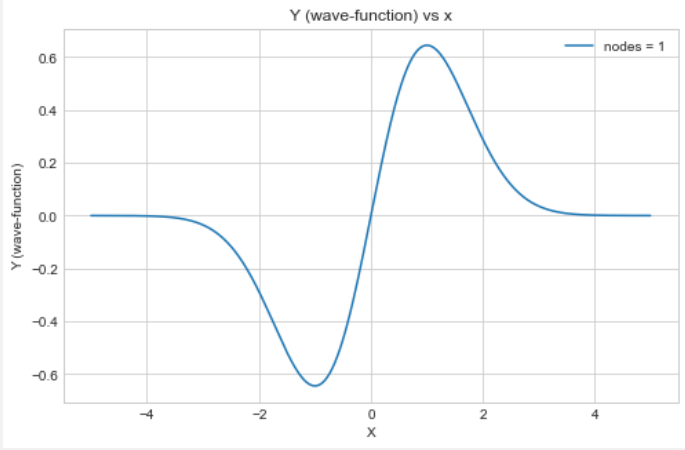
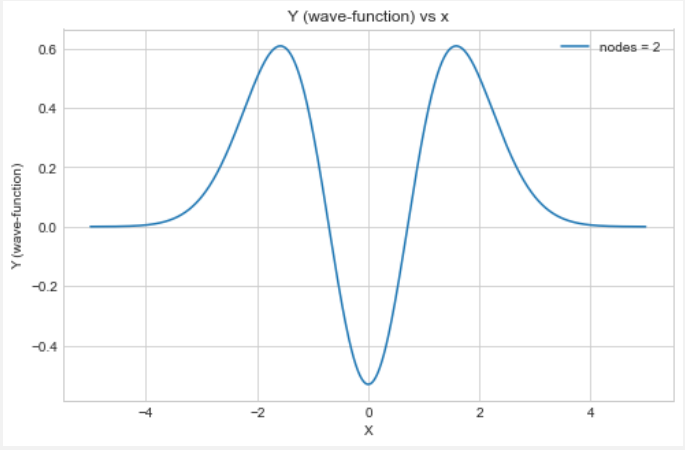
3 Flowchart of shooting method by 4th-RK method

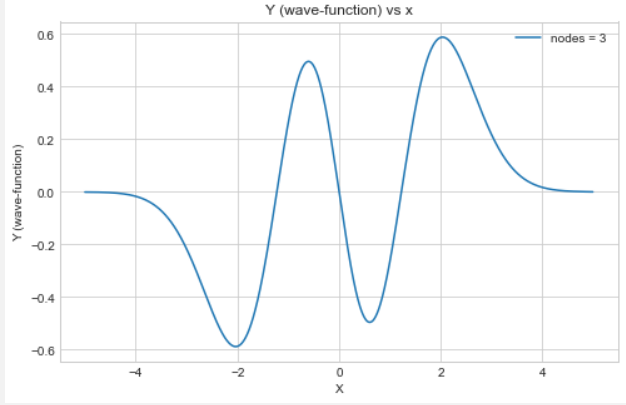
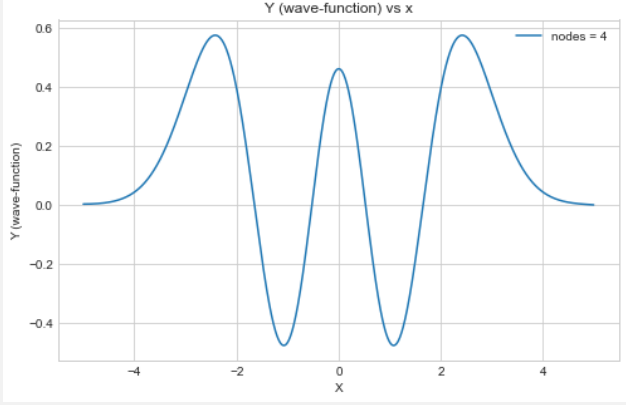


4 Results

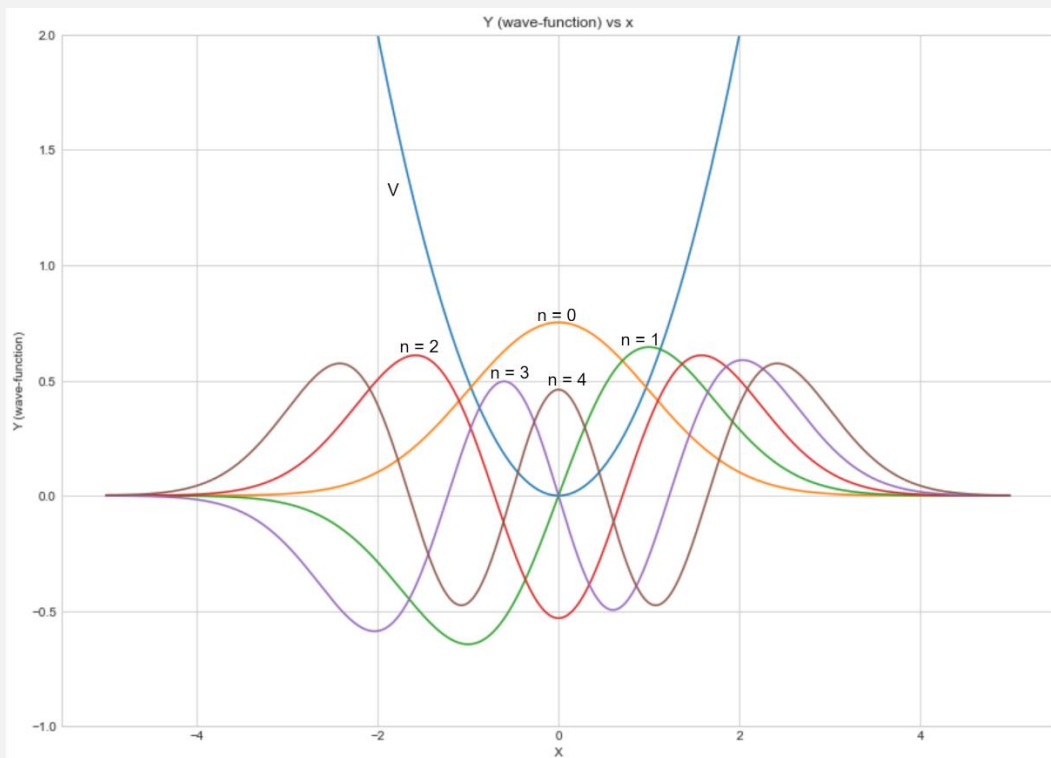
4.1 Testing code with Simple Harmonic Oscillator (SHO) at $x = 0$ ($V(x) = \frac{1}{2}kx^2$)

1. E_n and $\psi_n(x)$

nodes	Energy	Eigen wave function
0	$E_0 = -0.5 \text{ Hartree}$	
1	$E_1 = -1.5 \text{ Hartree}$	
2	$E_2 = -2.5 \text{ Hartree}$	

3	$E_3 = -3.5 \text{ Hartree}$	
4	$E_4 = -4.5 \text{ Hartree}$	

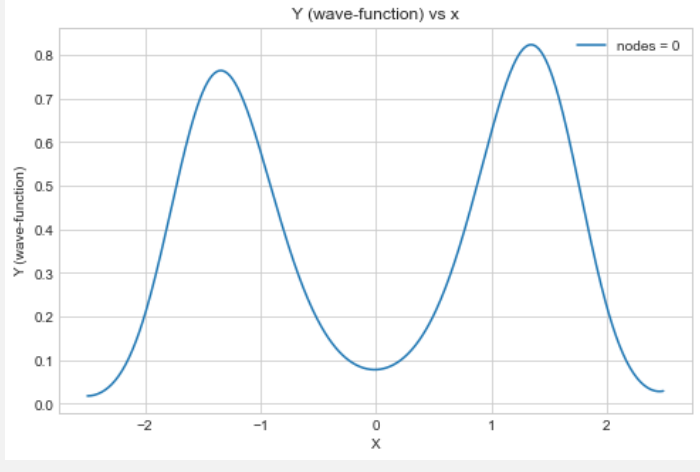
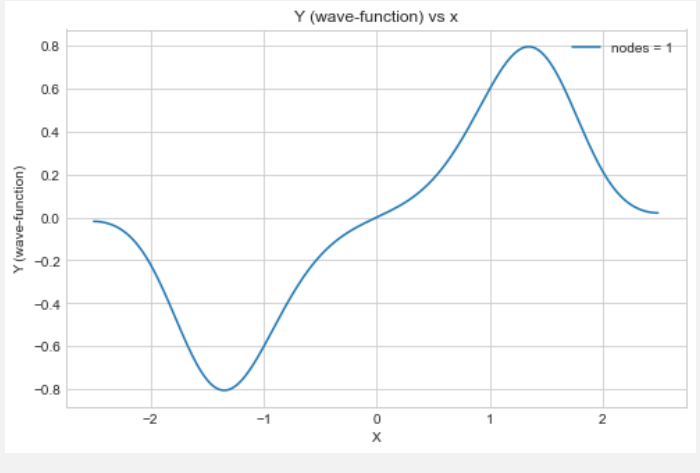
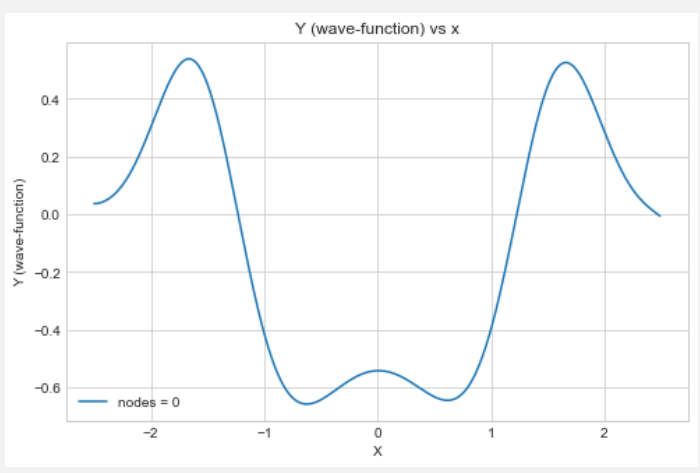
2. $\psi_n(x)$ s with SHO Potential ($V(x) = \frac{1}{2}kx^2$)

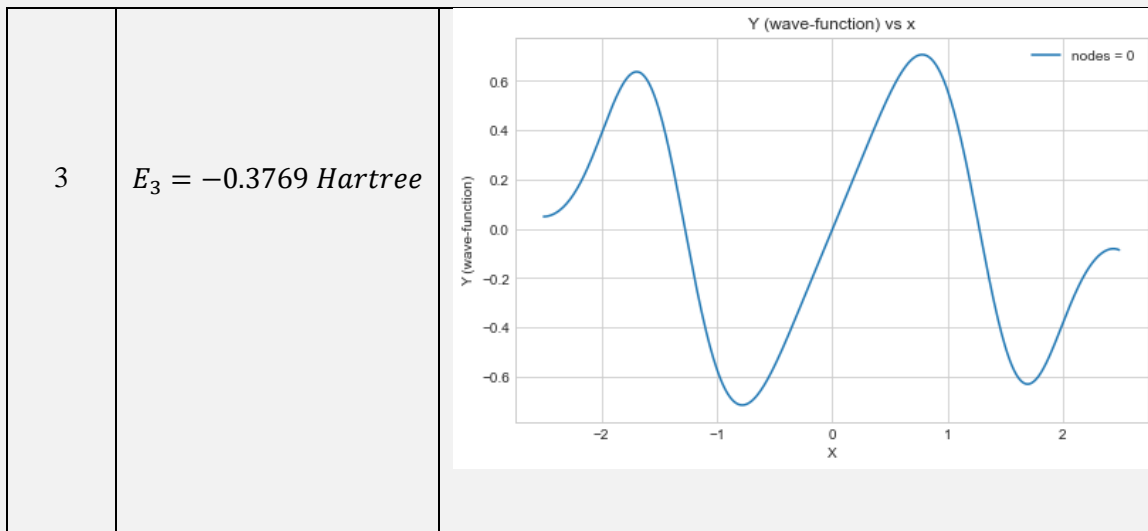


Simple Harmonic Oscillator at $x = 0$

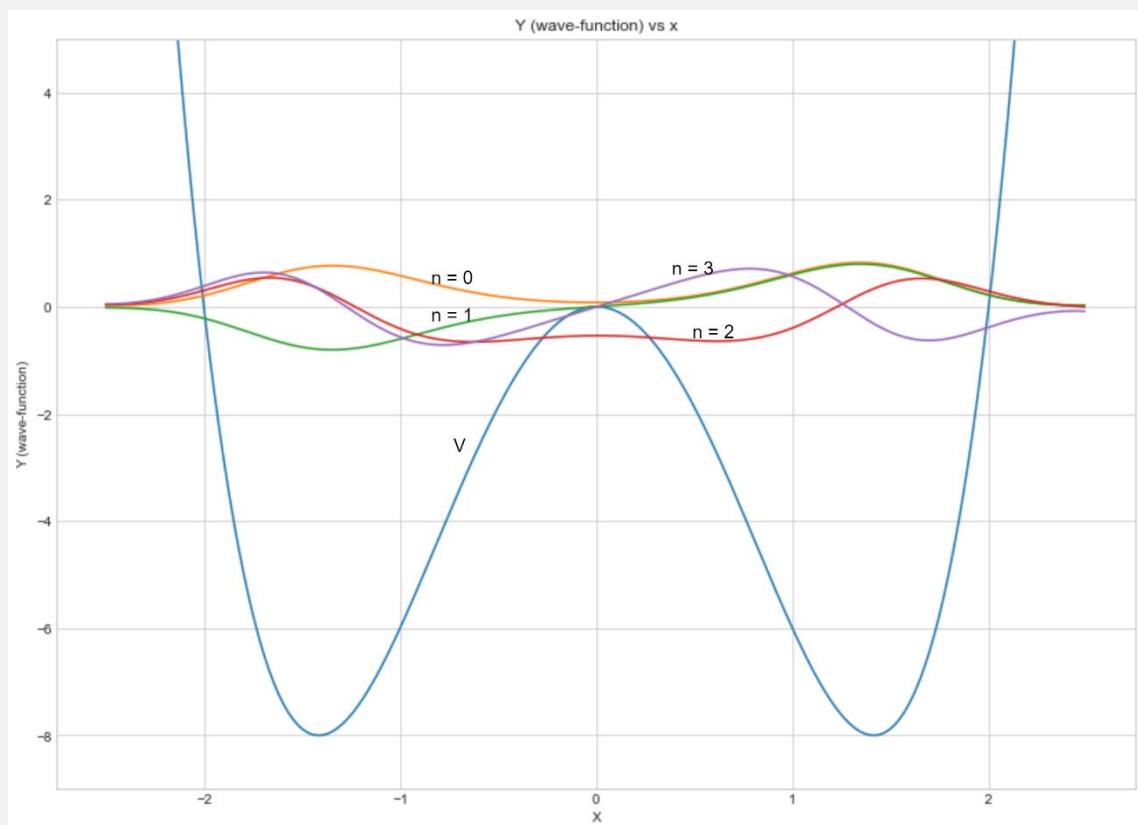
4.2 Double well Potential: E_n and $\psi_n(x)$ ($V(x) = 2x^2 - 8x^4$)

1. E_n and $\psi_n(x)$

nodes	Energy	Eigen wave function
0	$E_0 = -5.3250 \text{ Hartree}$	
1	$E_1 = -5.3050 \text{ Hartree}$	
2	$E_2 = -1.0199 \text{ Hartree}$	



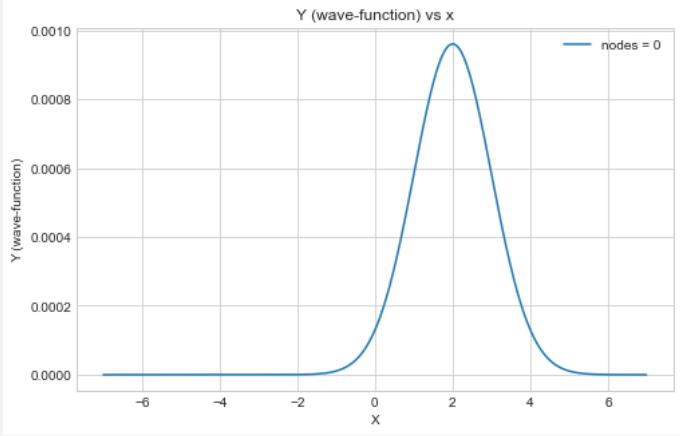
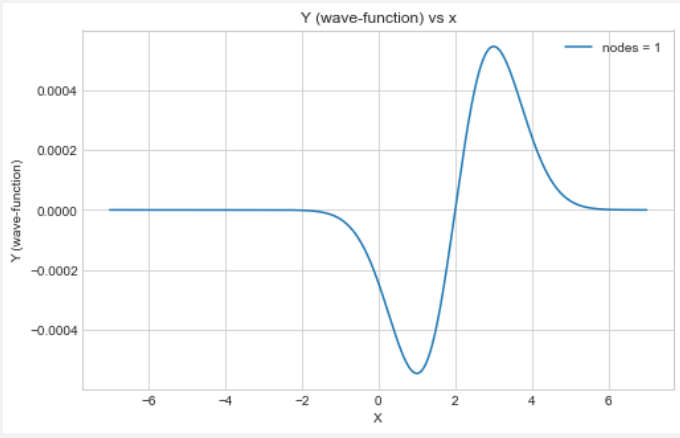
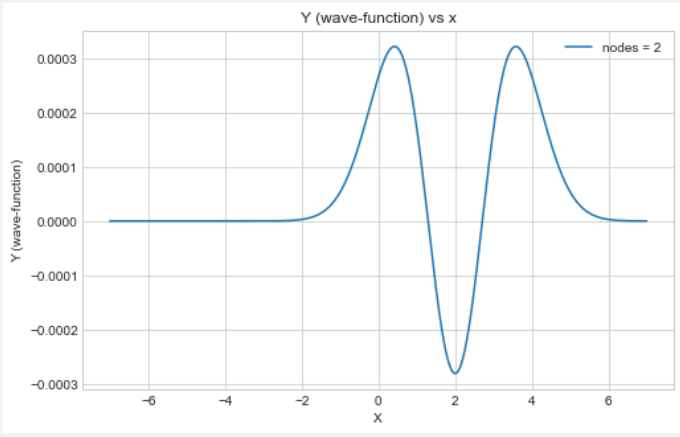
2. $\psi_n(x)$ s with double-well potential ($V(x) = 2x^2 - 8x^4$)

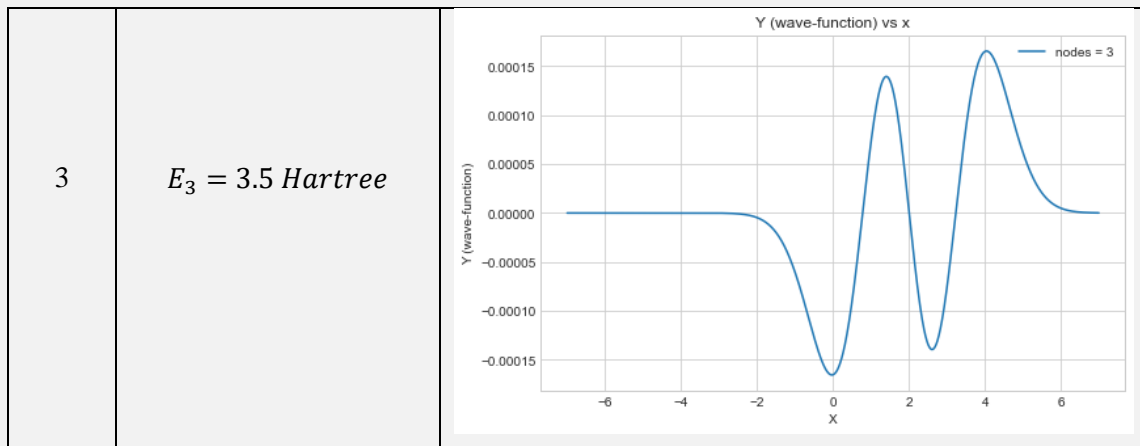


Double well Potential

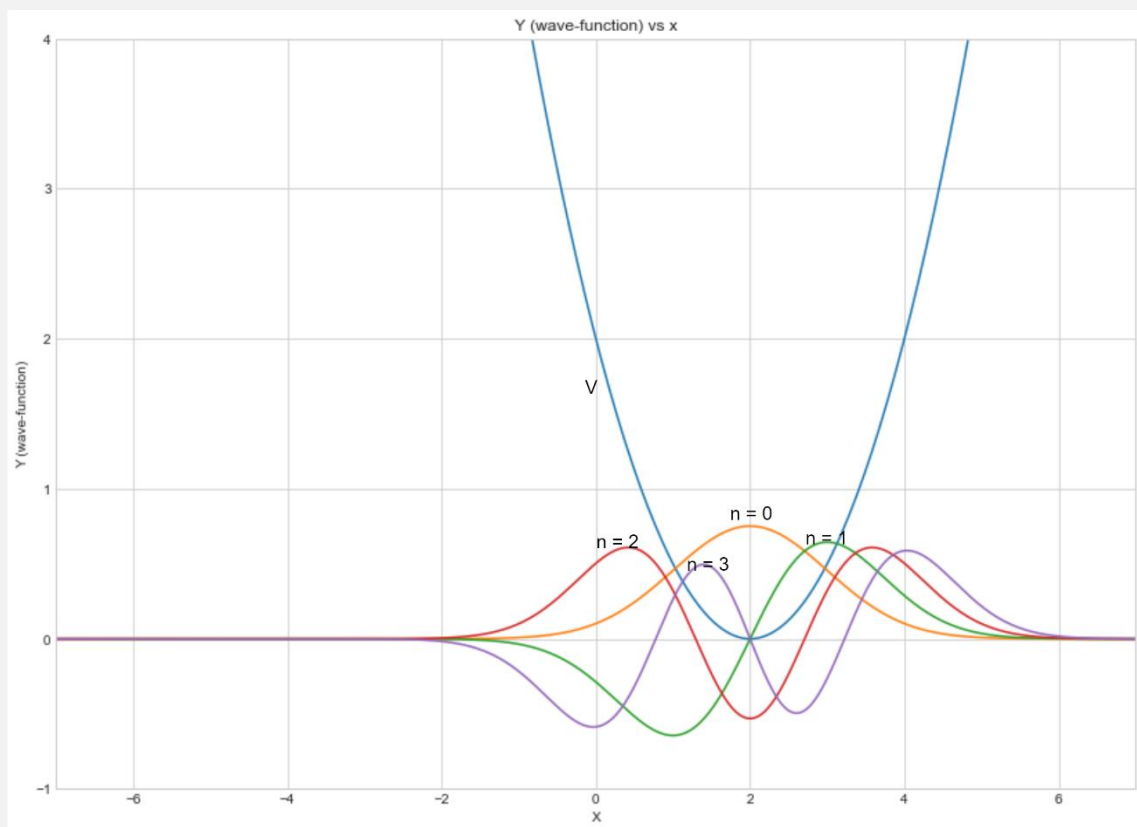
4.3 Simple Harmonic Oscillator at $x = 2$ ($V(x) = \frac{1}{2}(x - 2)^2$)

3. E_n and $\psi_n(x)$

nodes	Energy	Eigen wave function
0	$E_0 = 0.5 \text{ Hartree}$	
1	$E_1 = 1.5 \text{ Hartree}$	
2	$E_2 = 2.5 \text{ Hartree}$	



4. $\psi_n(x)$ s with double-well potential $\left(V(x) = \frac{1}{2}(x - 2)^2\right)$



Simple Harmonic Oscillator at $x = 2$

5 Discussion

The code implemented for the double-shooting method has been tested for solving Schrödinger equation for the SHO at $x = 0$. We can confirm that the solutions generated by the code are correct since we already have the analytical solution of SHO at $x = 0$. Also, the analytical values of eigen-energies i.e. $E_n = 0.5, 1.5, 2.5, 3.5, \dots$ *Hartree* in atomic units matches with the energies output of the simulation code. So, we can say that the code works (at least for the SHO).

Further, the eigen wave functions for the double well potential has been generated. The solutions are alternatively even and odd solutions which we can confirm by their number of nodes which was also expected since our potential was symmetric.

Solutions are also generated for $V(x)$ for SHO around at $x = 2$. The solutions are identical to the solutions for SHO at $x = 0$ except that they are shifted to right by 2 which was also expected.

To conclude, from the results of this project I can say with confidence that the numerical double shooting method using 4th order Runge-Kutta integration method has been successfully implemented for solving Schrödinger equation.

6 Code

Please see the next page for the code used in this project.

Code

```
In [25]: import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
```

```
In [26]: import scipy.optimize
def V(x): return 2*x**4-8*x**2
max_x = scipy.optimize.fmin(lambda x: V(x), 0)
```

Optimization terminated successfully.
Current function value: -8.000000
Iterations: 26
Function evaluations: 52

4th order Runge Kutta method

```

In [27]: #####
##### RK_4th #####
def RK_4th(a, b, h, d2y, ya, d1ya, plot_enabled = True, normalized = False, label = None)

    """
    General info:
        This function solves 2nd order differential equation using 4th order
        runge kutta method.
    Arguments:
        a      : lower limit
        b      : higher limit
        h      : interval length (dx)
        d2y    : function handle to 2nd derivative of y
        ya     : value of y at starting point a
        d1ya   : value of d1y at starting point a
    """

    import numpy as np
    import matplotlib.pyplot as plt
    plt.style.use('seaborn-whitegrid')

    def d1y(x, y, z):
        return z

    xpoints = np.arange(a,b,h)
    ypoints = []
    zpoints = []

    y = ya
    z = d1ya

    for x in xpoints:
        ypoints.append(y)
        zpoints.append(d1y)

        k1 = h * d1y(x, y, z)
        l1 = h * d2y(x, y, z)

        k2 = h * d1y(x+0.5*h, y+0.5*k1, z+0.5*l1)
        l2 = h * d2y(x+0.5*h, y+0.5*k1, z+0.5*l1)

        k3 = h * d1y(x+0.5*h, y+0.5*k2, z+0.5*l2)
        l3 = h * d2y(x+0.5*h, y+0.5*k2, z+0.5*l2)

        k4 = h * d1y(x+h, y+k3, z+l3)
        l4 = h * d2y(x+h, y+k3, z+l3)

        y = y + (k1 + 2*k2 + 2*k3 + k4) / 6

```

```

        z = z + (l1 + 2*l2 + 2*l3 + l4) / 6

    if normalized:

        ypoints_sqr = np.square(ypoints)
        # Calculating integration >>
        f_a = ypoints_sqr[0]
        f_b = ypoints_sqr[-1]
        I_n = (f_a + f_b) / 2

        # Loop for adding n-1 terms >>
        for ypoint_sqr in ypoints_sqr[1:-1]:
            I_n = I_n + ypoint_sqr * abs(h)
        ypoints = ypoints / np.sqrt(I_n)
        y        = y        / np.sqrt(I_n)
        z        = z        / np.sqrt(I_n)

    if plot_enabled == True:

        # Plotting R_average vs N for many trials >>
        fig = plt.figure(figsize = (8, 5))
        axes = plt.gca()
        if label: axes.plot(xpoints, ypoints, label = label); plt.legend()
        else: axes.plot(xpoints, ypoints)

        # Setting plot elements >>
        axes.set_title("Y (wave-function) vs x")
        axes.set_xlabel("X")
        axes.set_ylabel("Y (wave-function)")
        plt.show()
        return y, z, axes

    else:
        return y, z, None

##### RK_4th #####
#####

```

```

In [28]: #####
##### plot_wavefunc_with_V #####
def plot_wavefunc_with_V( func_V,
                        axes_wavefunc,
                        figsize = (8, 6),
                        rangex_V = (-2.5, 2.5),
                        xlim     = None,
                        ylim     = None
                        ):

    import numpy as np

    # Making new figure for V >>
    fig     = plt.figure(figsize = figsize)
    axes_V = plt.gca()
    xpoints_V = np.linspace(rangex_V[0], rangex_V[1], num = 200)
    ypoints_V = func_V(xpoints_V)
    axes_V.plot(xpoints_V, ypoints_V)

    if type(axes_wavefunc) == list:
        for ax_wavefunc in axes_wavefunc:
            # Adding wave-functions plot to V figure >>
            xydata_wavefunc = ax_wavefunc.get_lines()[0].get_xydata()
            xpoints_wavefunc = xydata_wavefunc[:, 0]
            ypoints_wavefunc = xydata_wavefunc[:, 1]
            axes_V.plot(xpoints_wavefunc, ypoints_wavefunc)

    else:
        ax_wavefunc = axes_wavefunc
        xydata_wavefunc = ax_wavefunc.get_lines()[0].get_xydata()
        xpoints_wavefunc = xydata_wavefunc[:, 0]
        ypoints_wavefunc = xydata_wavefunc[:, 1]
        axes_V.plot(xpoints_wavefunc, ypoints_wavefunc)

    # Setting plot elements >>
    axes_V.set_title("Y (wave-function) vs x")
    axes_V.set_xlabel("X")
    axes_V.set_ylabel("Y (wave-function)")
    if xlim: axes_V.set_xlim(xlim)
    if ylim: axes_V.set_ylim(ylim)
    plt.show()

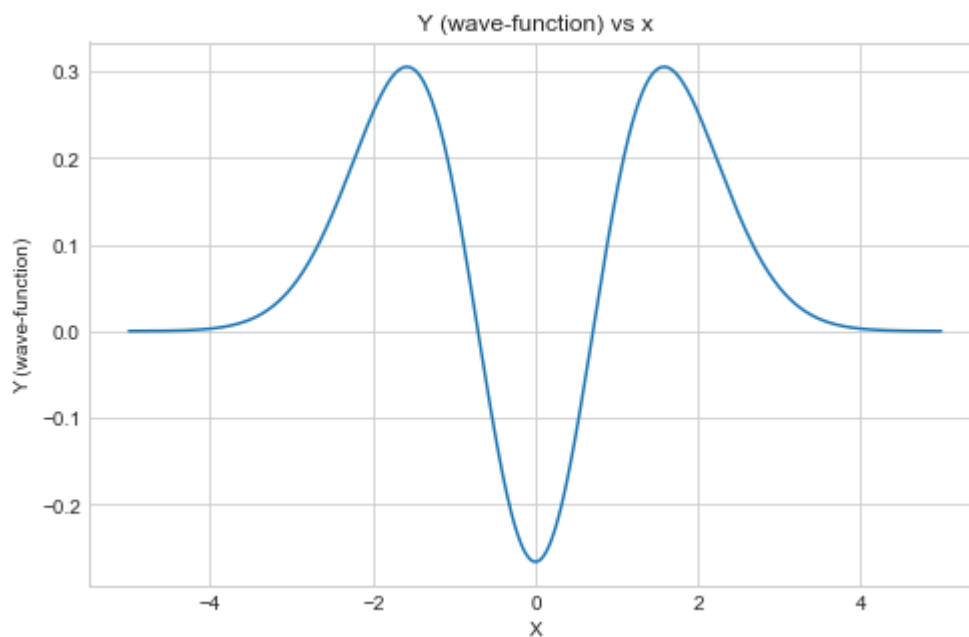
    return None
##### plot_wavefunc_with_V #####
#####

```

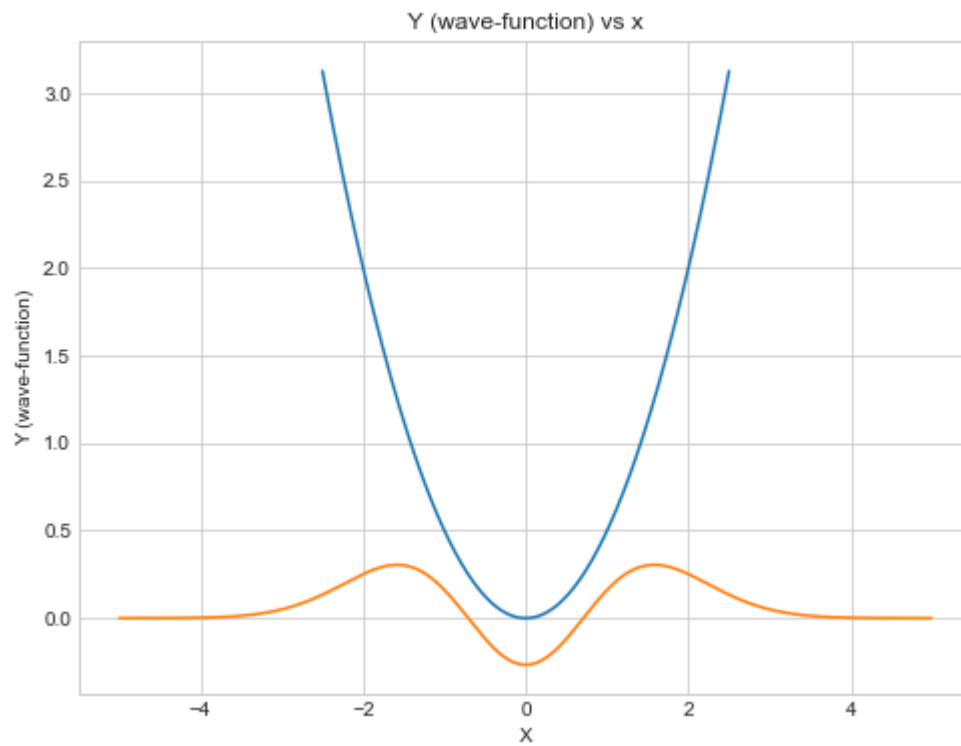
```
In [29]: def V(x):  
         return 1/2*x**2  
  
         def d2y(x, y, z):  
             n = 2; h = 1; w = 1  
             E = (n + 0.5) * h * w  
             return 2*(V(x)-E) * y
```

Not normalized

```
In [30]: _, _, axes_sh = RK_4th( a      = -5,  
                                b      = 5,  
                                h      = 0.01,  
                                d2y    = d2y,  
                                ya     = 0.0001,  
                                d1ya   = 0,  
                                normalized = False )
```

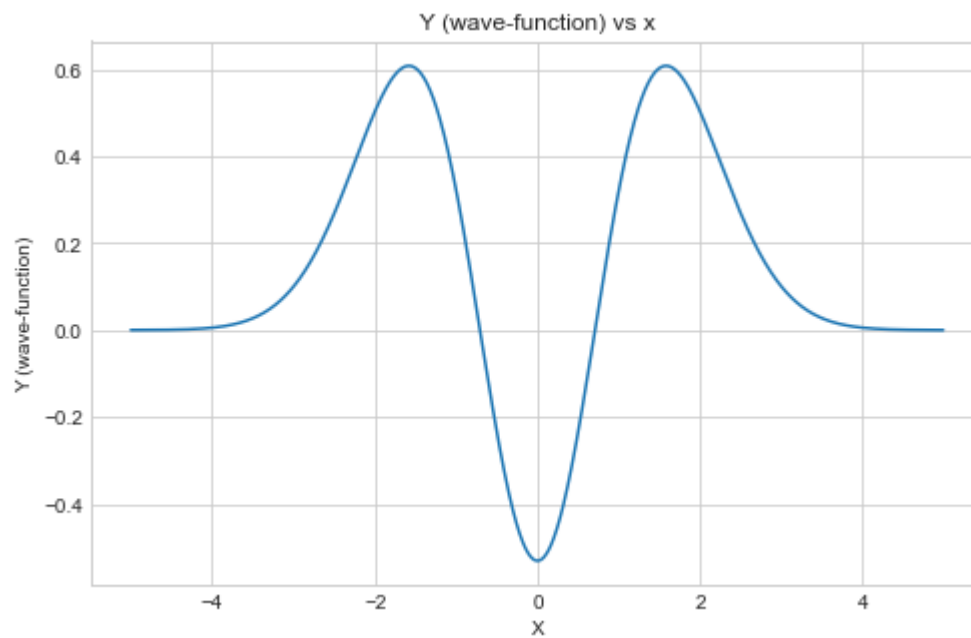


```
In [31]: plot_wavefunc_with_V( func_V      = V,  
                               axes_wavefunc = axes_sh,  
                               figsize      = (8, 6),  
                               rangex_V    = (-2.5, 2.5) )
```

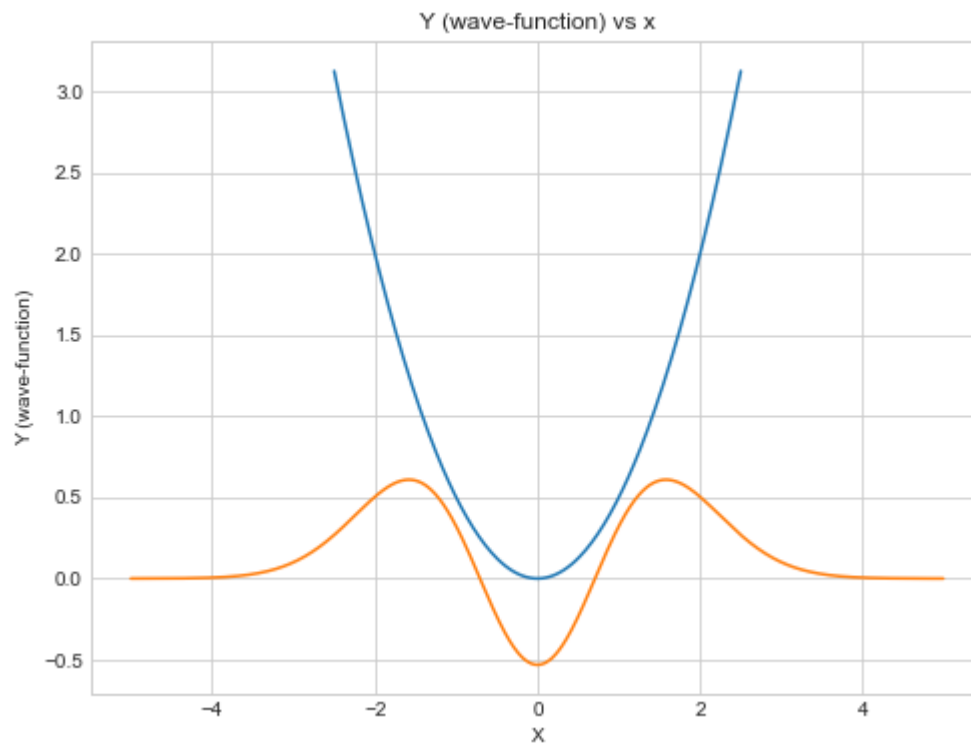


Normalized

```
In [32]: _, _, axes_sh_norm = RK_4th( a      = -5,  
                                     b      = 5,  
                                     h      = 0.01,  
                                     d2y    = d2y,  
                                     ya      = 0.0001,  
                                     d1ya    = 0,  
                                     normalized = True )
```




```
In [33]: plot_wavefunc_with_V( func_V      = V,  
                               axes_wavefunc = axes_sh_norm,  
                               figsize      = (8, 6),  
                               rangex_V    = (-2.5, 2.5))
```



Double shooting method using RK_4th

```

In [34]: #####
##### solve_schrod_RK_4th #####
def solve_schrod_RK_4th(a, b, h, d2y, ya, d1ya, El, Eh, dE, xlim = None, ylim = None, ma

    """
    General info:
        This function solves schrodinger's equation using 4th order
        runge kutta method.
    Arguments:
        a      : lower limit of x
        b      : upper limit of x
        h      : interval length (dx)
        d2y    : function handle to 2nd derivative of wave function y
        ya     : value of wave function y at starting point a
        d1ya   : value of 1st derivative of wave function y at starting point a
        El     : Lower limit of energy
        Eh     : Upper limit of energy
    """

    import numpy as np
    from progressbar import ProgressBar
    import matplotlib.pyplot as plt
    plt.style.use('seaborn-whitegrid')

    global E
    energies = np.arange(El, Eh, dE)
    axes_sol = []

    diff_y1_y_ratios = []
    eigen_values      = []

    pbar = ProgressBar()
    i = 0
    for E in pbar(energies):
        # i = i + 1

        ya_left = (-1)**(i)*ya
        y_L, z_L, _ = RK_4th( a, (b - (a+b)/2) * 0.05, h, d2y, ya_left, d1ya, plot_enab
        y_R, z_R, _ = RK_4th( b, (b - (a+b)/2) * 0.05, -h, d2y, ya, d1ya, plot_enab

        y1_y_ratio = abs(z_L/y_L - z_R/y_R)

        if abs(z_L/y_L - z_R/y_R) < match_ratio:
            print("")
            print("i          =", i)
            print("E          =", E)
            y, _, ax_sol = RK_4th( a, b, h, d2y, ya_left, d1ya, plot_enabled = True, nor
            axes_sol.append(ax_sol)

```

```

        eigen_values.append(E)
        i = i + 1

    diff_y1_y_ratios.append(y1_y_ratio)

fig = plt.figure(figsize = (8, 6))
axes = plt.gca()
if xlim: axes.set_xlim(xlim)
if ylim: axes.set_ylim(ylim)
axes.plot(energies, diff_y1_y_ratios)
axes.set_title("(y1/y)L - (y1/y)R vs Eigen-energies")
axes.set_xlabel("Eigen-energies")
axes.set_ylabel("(y1/y)L - (y1/y)R")

return eigen_values, axes_sol
##### solve_schrod_RK_4th #####
#####

```

Testing double shooting method on SHO

```

In [35]: def V(x):
          return 1/2*x**2

def d2y(x, y, z):
    return 2*(V(x)-E) * y

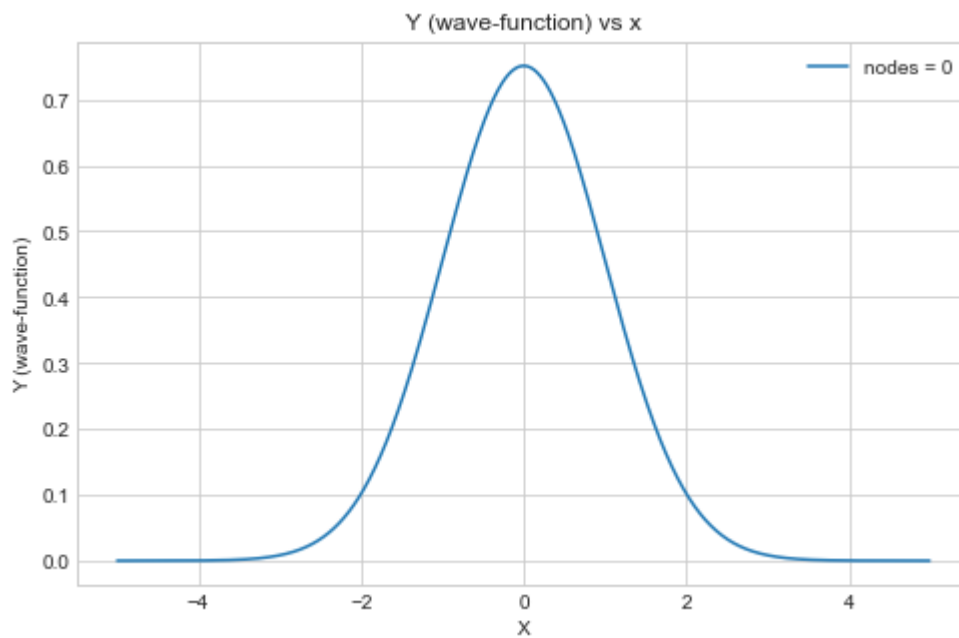
```

```
In [36]: eigen_values, axes_sh = solve_schrod_RK_4th( a    = -5,
                                                    b    = 5,
                                                    h    = 0.01,
                                                    d2y  = d2y,
                                                    ya   = 0.0001,
                                                    d1ya  = 0,
                                                    El    = 0,
                                                    Eh    = 5,
                                                    dE    = 0.1,
                                                    match_ratio = 0.01,
                                                    normalized = True )
                                                    # xlim = None,
                                                    # ylim = (0, 10) )

print("eigen_values: ", eigen_values)
```

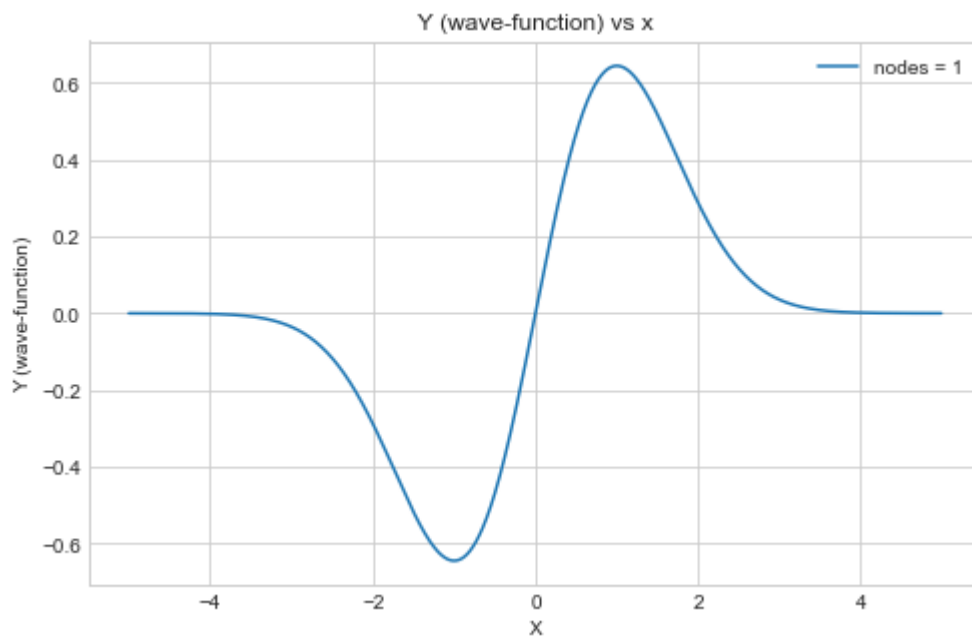
10% |#####

i = 0
E = 0.5



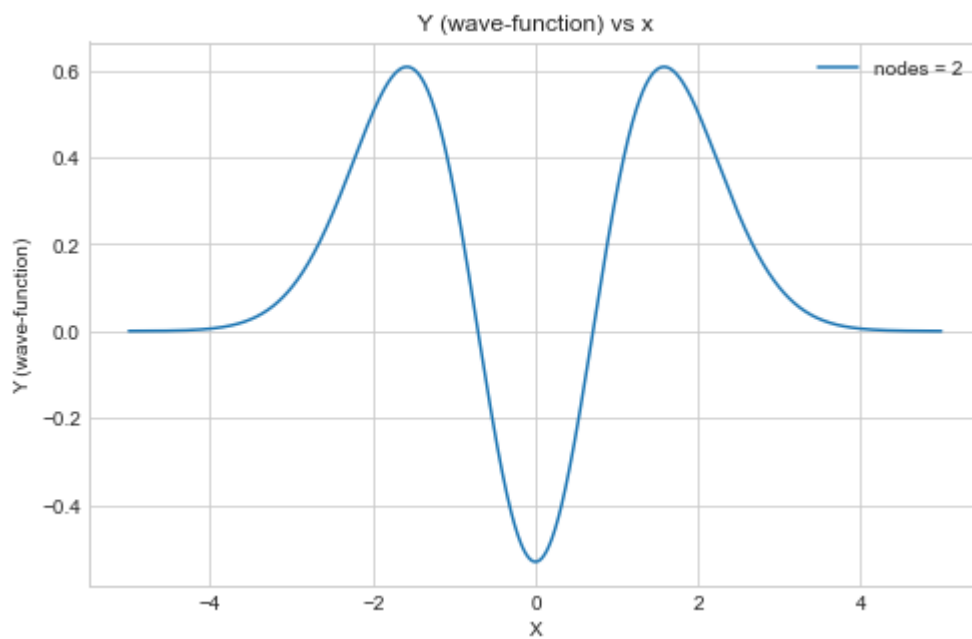
30% |#####

i = 1
E = 1.5



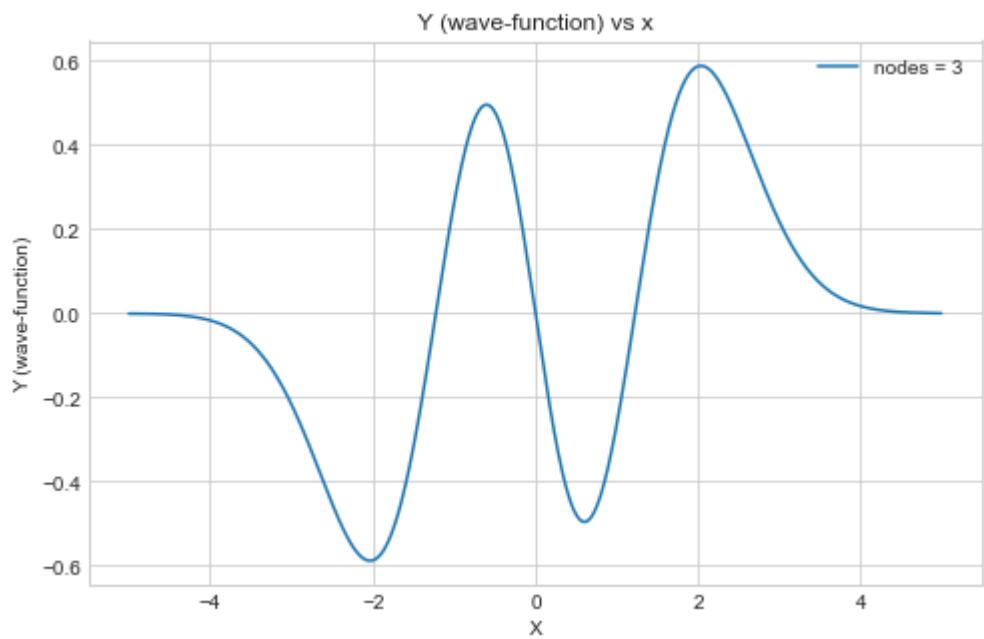
50% | #####

i = 2
E = 2.5



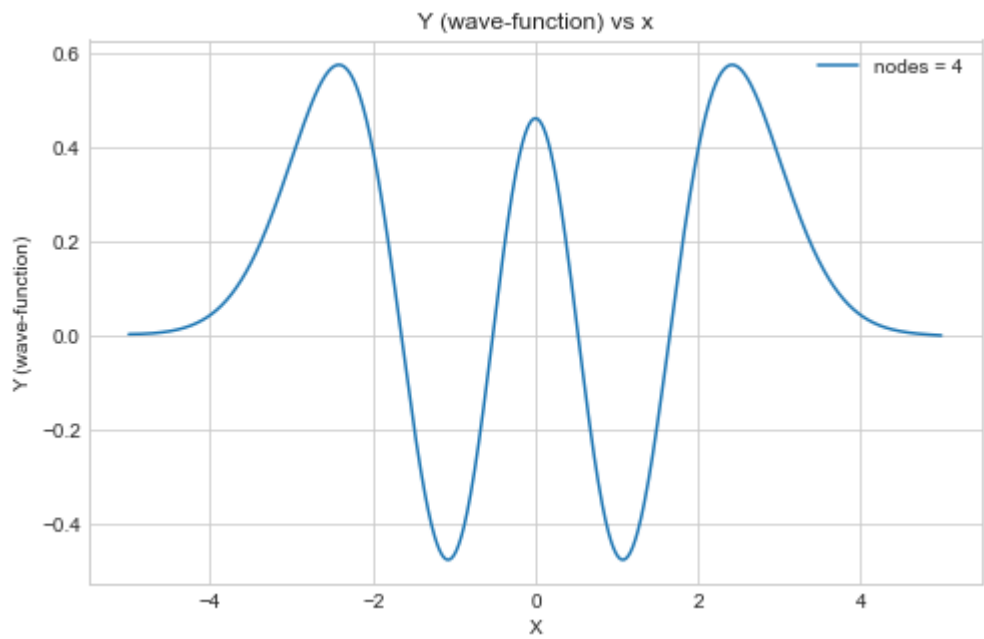
70% | #####

i = 3
E = 3.5



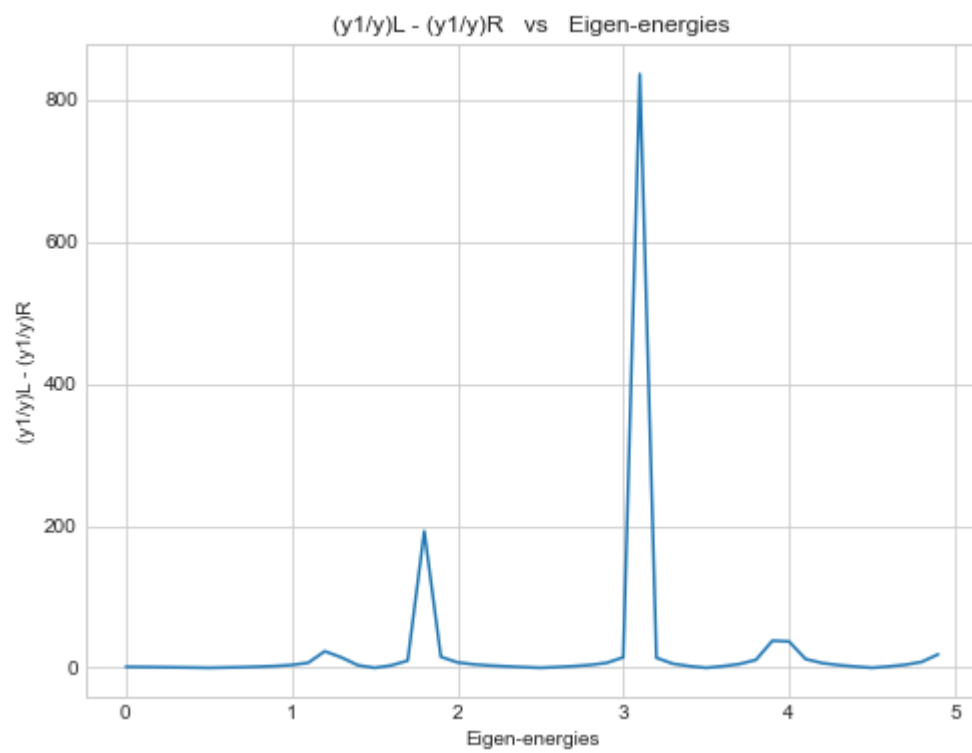
90% | ##### |

i = 4
E = 4.5

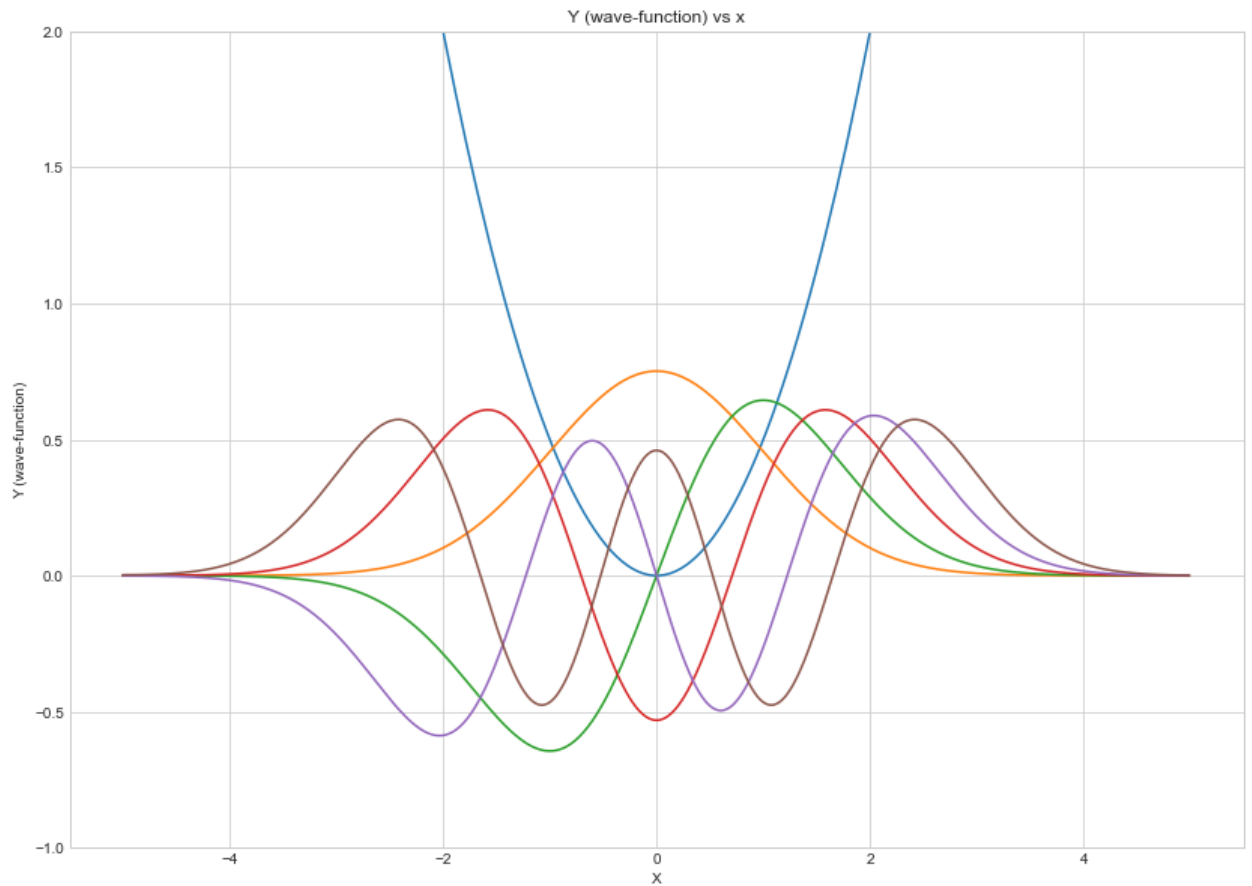


100% | ##### |

eigen_values: [0.5, 1.5, 2.5, 3.5, 4.5]



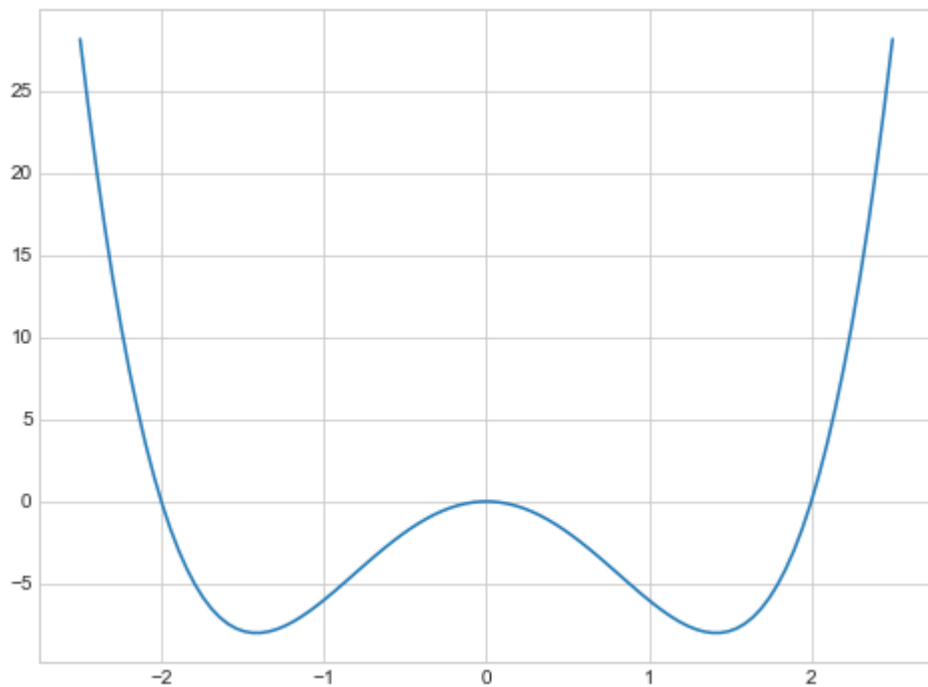
```
In [37]: plot_wavefunc_with_V( func_V      = V,  
                               axes_wavefunc = axes_sh,  
                               figsize      = (14, 10),  
                               rangex_V    = (-5, 5),  
                               ylim        = (-1, 2),  
                               )
```



Double Well Potential


```
In [39]: def V(x):  
         return 2*x**4 - 8*x**2  
  
         def d2y(x, y, z):  
             return 2*(V(x)-E) * y  
  
         fig = plt.figure(figsize = (8, 6))  
         axes = plt.gca()  
         xpoints = np.linspace(-2.5, 2.5, num = 100)  
         ypoints = V(xpoints)  
         axes.plot(xpoints, ypoints)
```

Out[39]: [matplotlib.lines.Line2D at 0x20006057d30>]



First attempt to get solution for double Well Potential

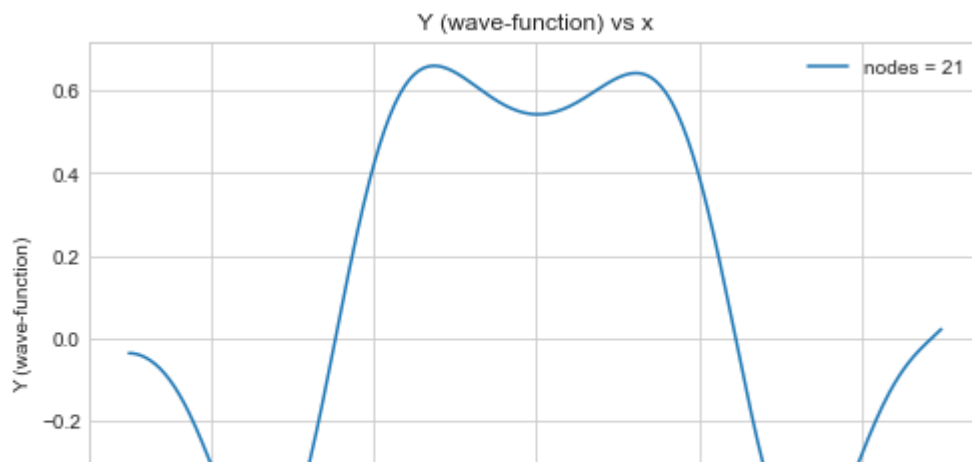
```

In [20]: def V(x):
          return 2*x**4 - 8*x**2

def d2y(x, y, z):
    return 2*(V(x)-E) * y

eigen_values, axes_doublewell = solve_schrod_RK_4th( a    = -2.5,
                                                    b    = 2.5,
                                                    h    = 0.01,
                                                    d2y   = d2y,
                                                    ya    = 0.0001,
                                                    d1ya  = 0,
                                                    E1    = -8,
                                                    Eh    = 0,
                                                    dE    = 0.001,
                                                    match_ratio = 0.1,
                                                    # xlim = None,
                                                    ylim = (0, 10) )

```



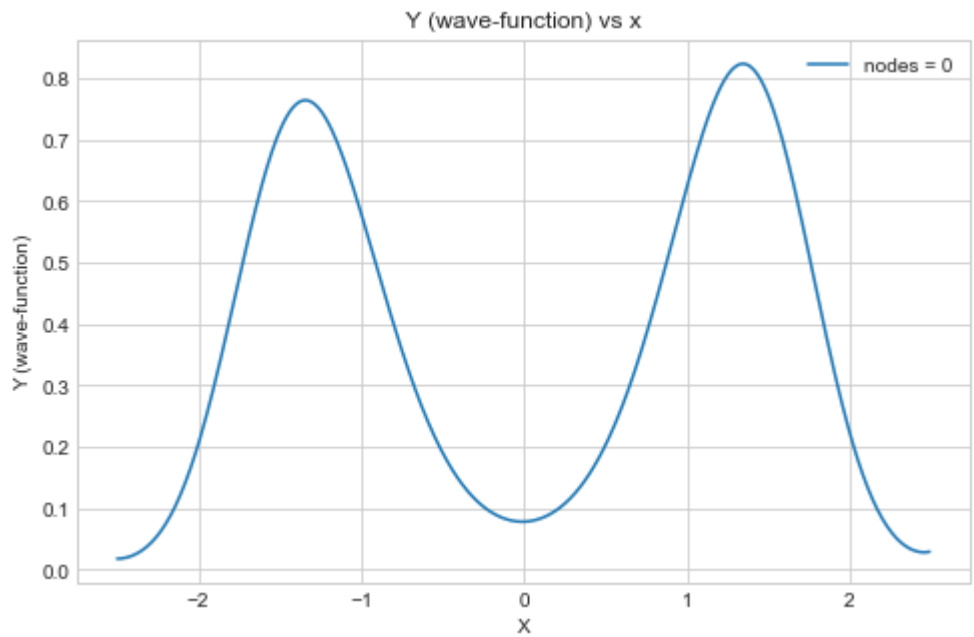
1st two solutions in range (-5.5, -5.2)

```
In [42]: eigen_values, axes_doublewell_1_2 = solve_schrod_RK_4th( a    = -2.5,
                                                                    b    =  2.5,
                                                                    h    =  0.01,
                                                                    d2y  = d2y,
                                                                    ya    =  0.0001,
                                                                    d1ya =  0,
                                                                    E1    = -5.5,
                                                                    Eh    = -5.2,
                                                                    dE    =  0.005,
                                                                    match_ratio = 0.4,
                                                                    ylim  = (0, 10) )
```

eigen_values

58% | #####

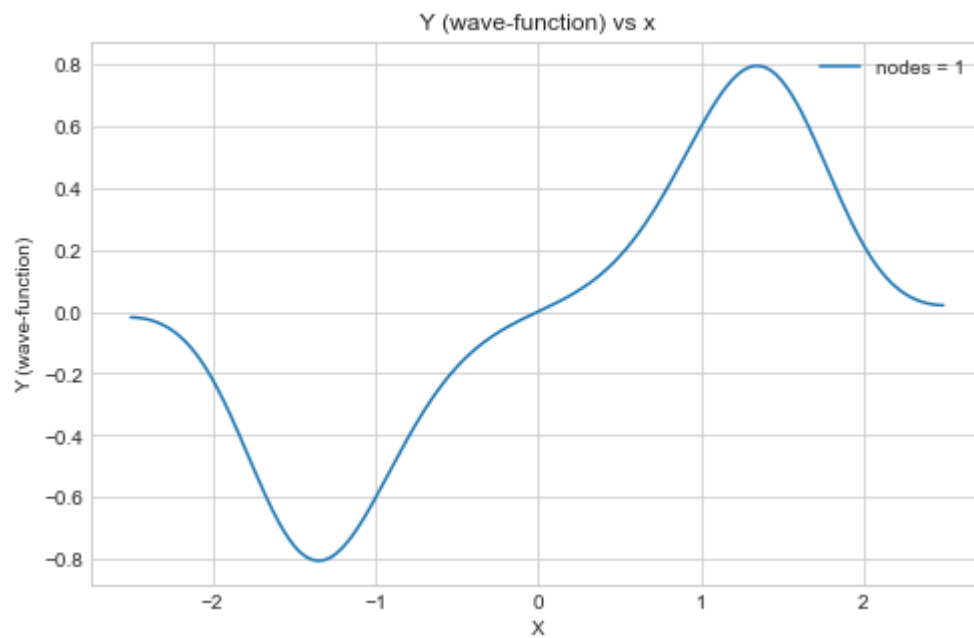
i = 0
E = -5.325000000000004



65% | #####

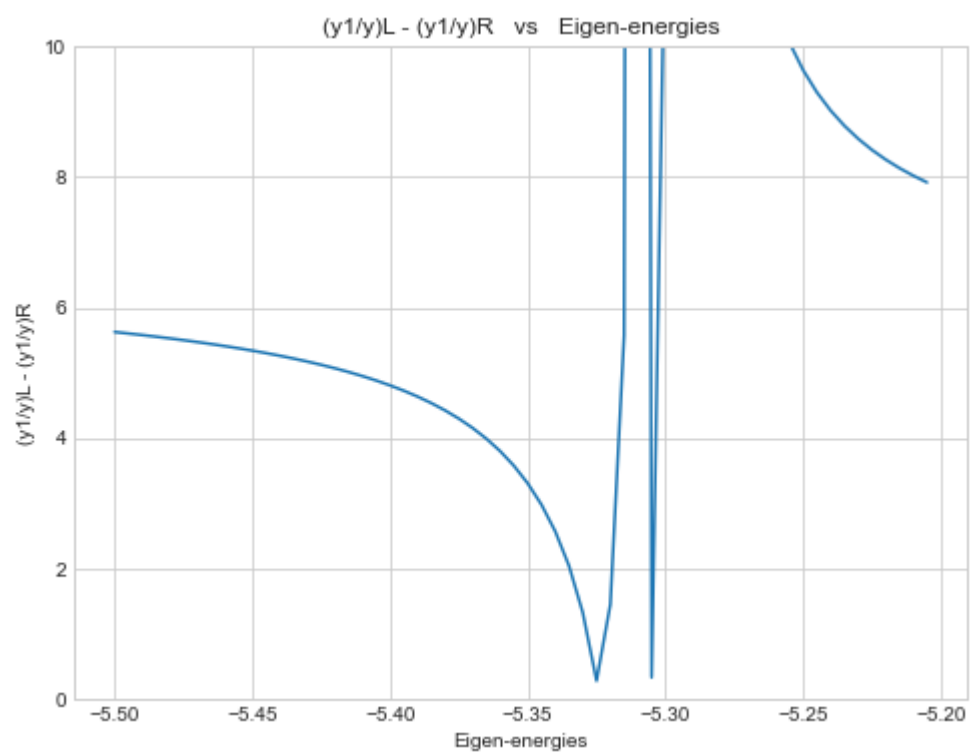


i = 1
E = -5.305000000000004

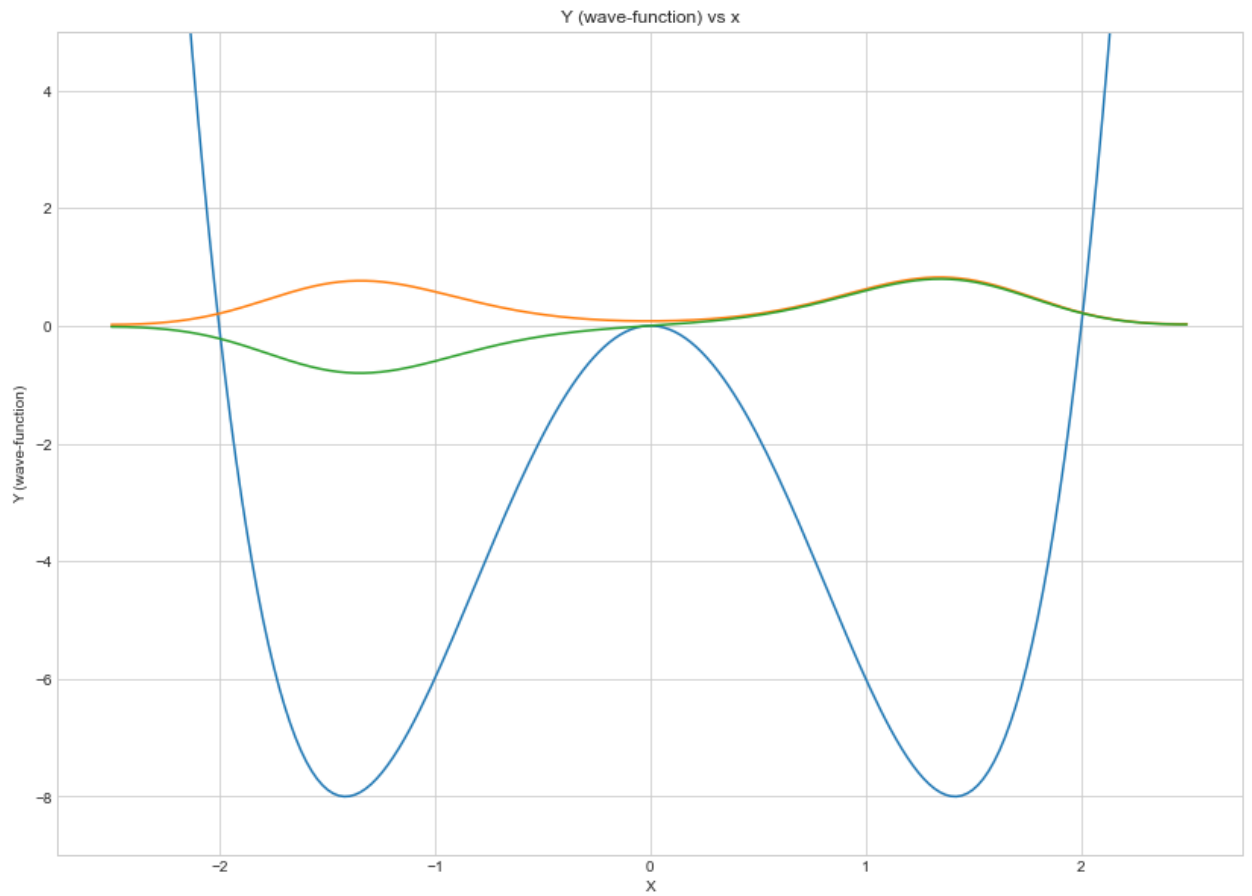


100% | #####

Out[42]: [-5.325000000000004, -5.305000000000004]



```
In [43]: plot_wavefunc_with_V( func_V      = V,  
                               axes_wavefunc = axes_doublewell_1_2,  
                               figsize      = (14, 10),  
                               rangex_V    = (-2.5, 2.5),  
                               ylim        = (-9, 5)  
                               )
```



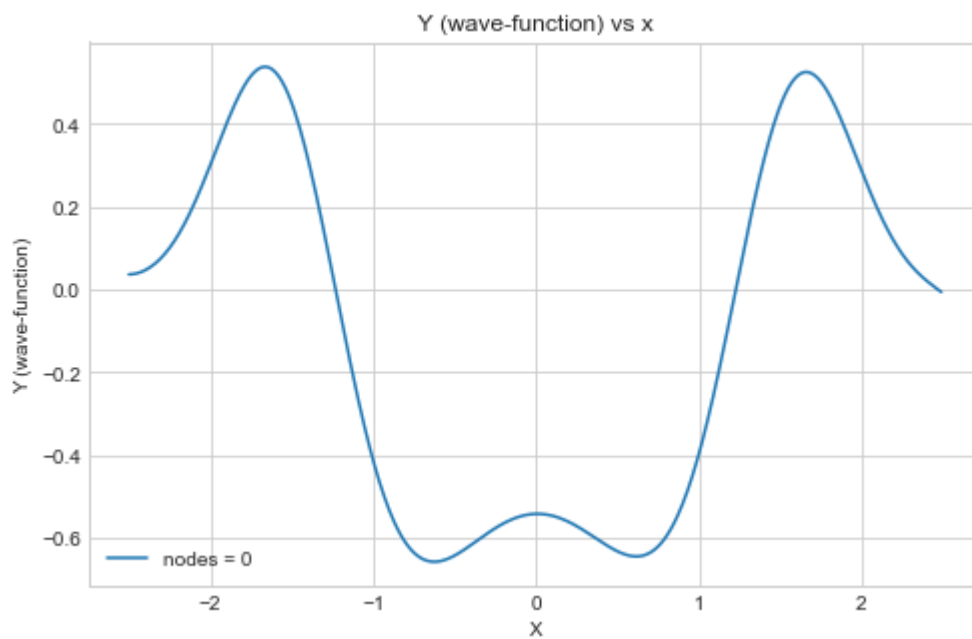
3rd solutions in range (-1.5, -1)

```
In [46]: eigen_values, axes_doublewell_3 = solve_schrod_RK_4th( a    = -2.5,
                                                                b    =  2.5,
                                                                h    =  0.01,
                                                                d2y  = d2y,
                                                                ya   =  0.0001,
                                                                d1ya =  0,
                                                                El   = -1.5,
                                                                Eh   = -1,
                                                                dE   =  0.01,
                                                                match_ratio = 0.04,
                                                                ylim  = (0, 10) )
```

eigen_values

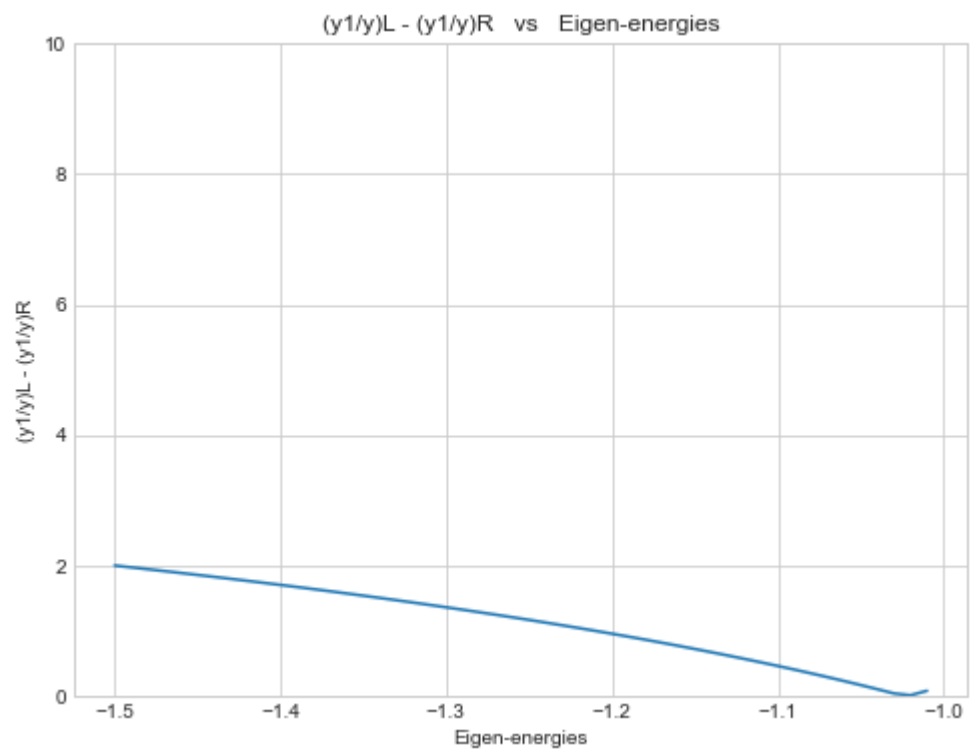
```
96% | #####
```

```
i      = 0
E      = -1.0199999999999996
```

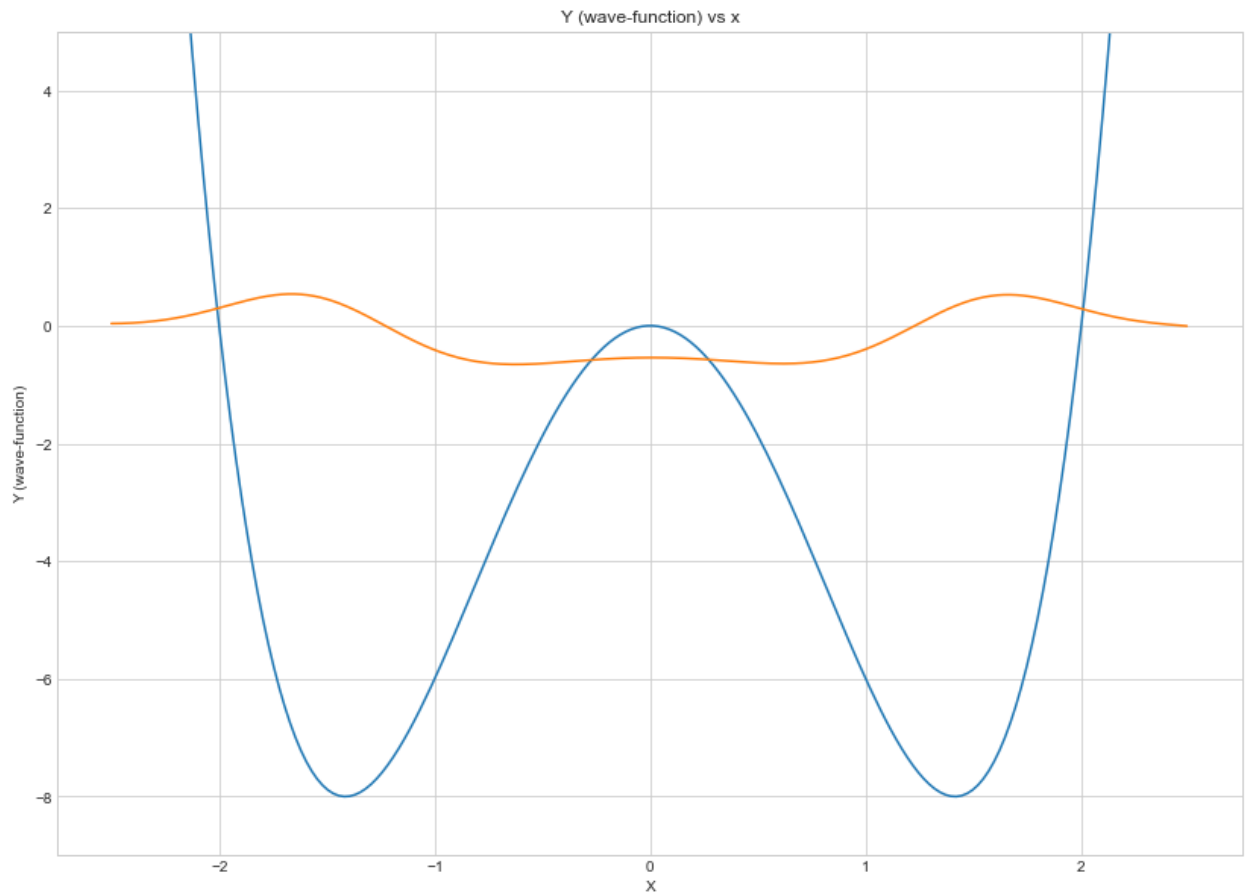


100% | ##### |

Out[46]: [-1.0199999999999996]




```
In [47]: plot_wavefunc_with_V( func_V      = V,  
                               axes_wavefunc = axes_doublewell_3,  
                               figsize      = (14, 10),  
                               rangex_V    = (-2.5, 2.5),  
                               ylim        = (-9, 5)  
                               )
```

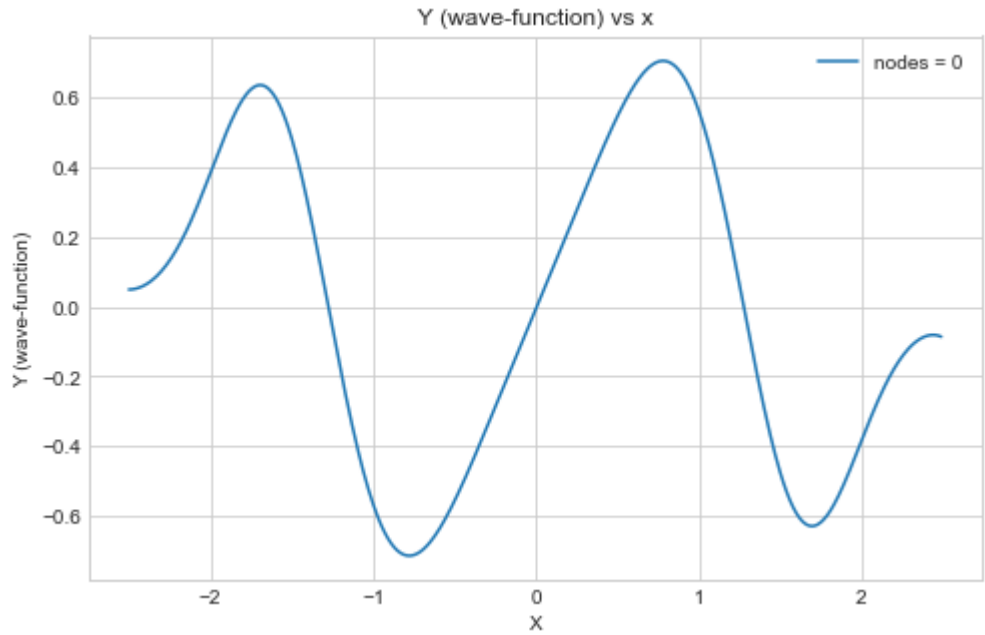


4th solutions in range (-0.5, 0)

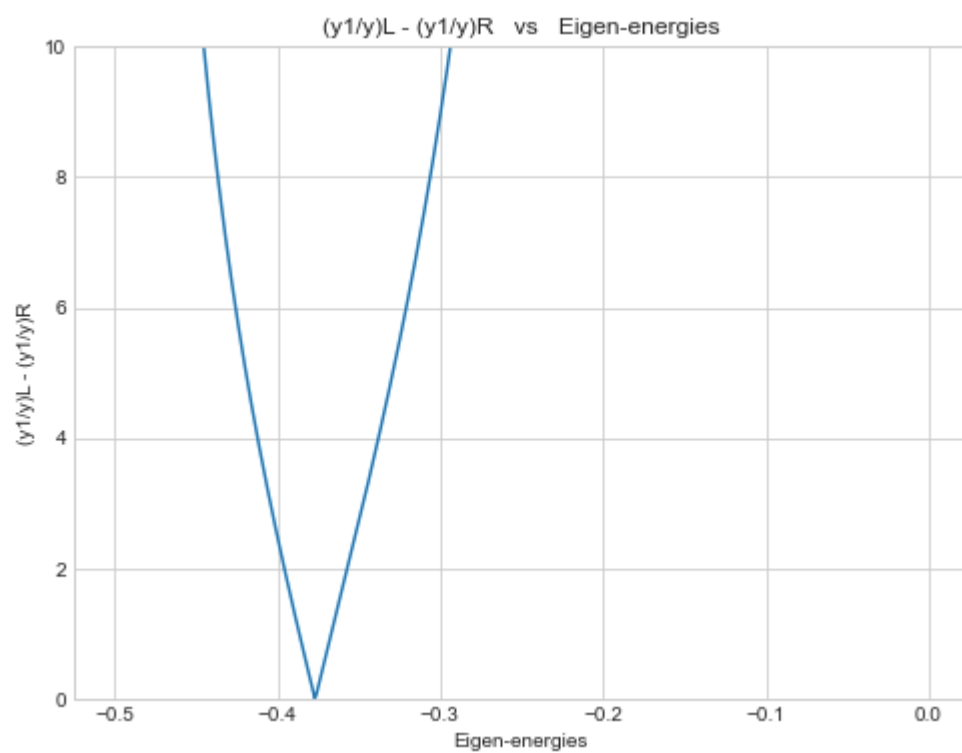
```
In [48]: eigen_values, axes_doublewell_4 = solve_schrod_RK_4th( a      = -2.5,
                                                                b      = 2.5,
                                                                h      = 0.01,
                                                                d2y    = d2y,
                                                                ya     = 0.0001,
                                                                d1ya   = 0,
                                                                E1     = -0.5,
                                                                Eh     = 0,
                                                                dE     = 0.001,
                                                                match_ratio = 0.02,
                                                                # xlim = None,
                                                                ylim    = (0, 10) )
```

24% |#####

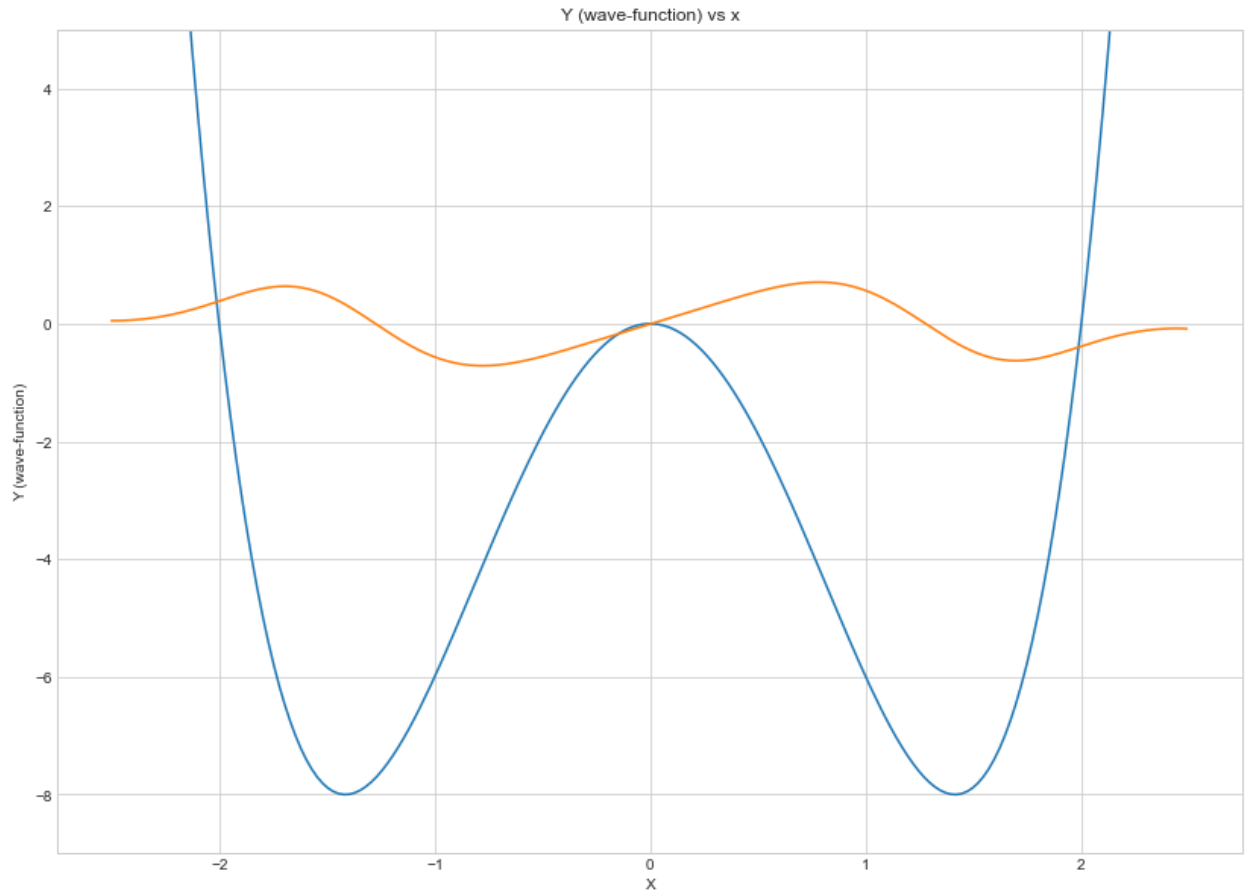
i = 0
E = -0.3769999999999999



100% |#####



```
In [50]: plot_wavefunc_with_V( func_V      = V,
                               axes_wavefunc = axes_doublewell_4,
                               figsize      = (14, 10),
                               rangex_V     = (-2.5, 2.5),
                               ylim         = (-9, 5)
                               )
```



Final Solution Problem 1

Eigen-Values for double wave potential

$$E_0 = -5.325000000000004$$

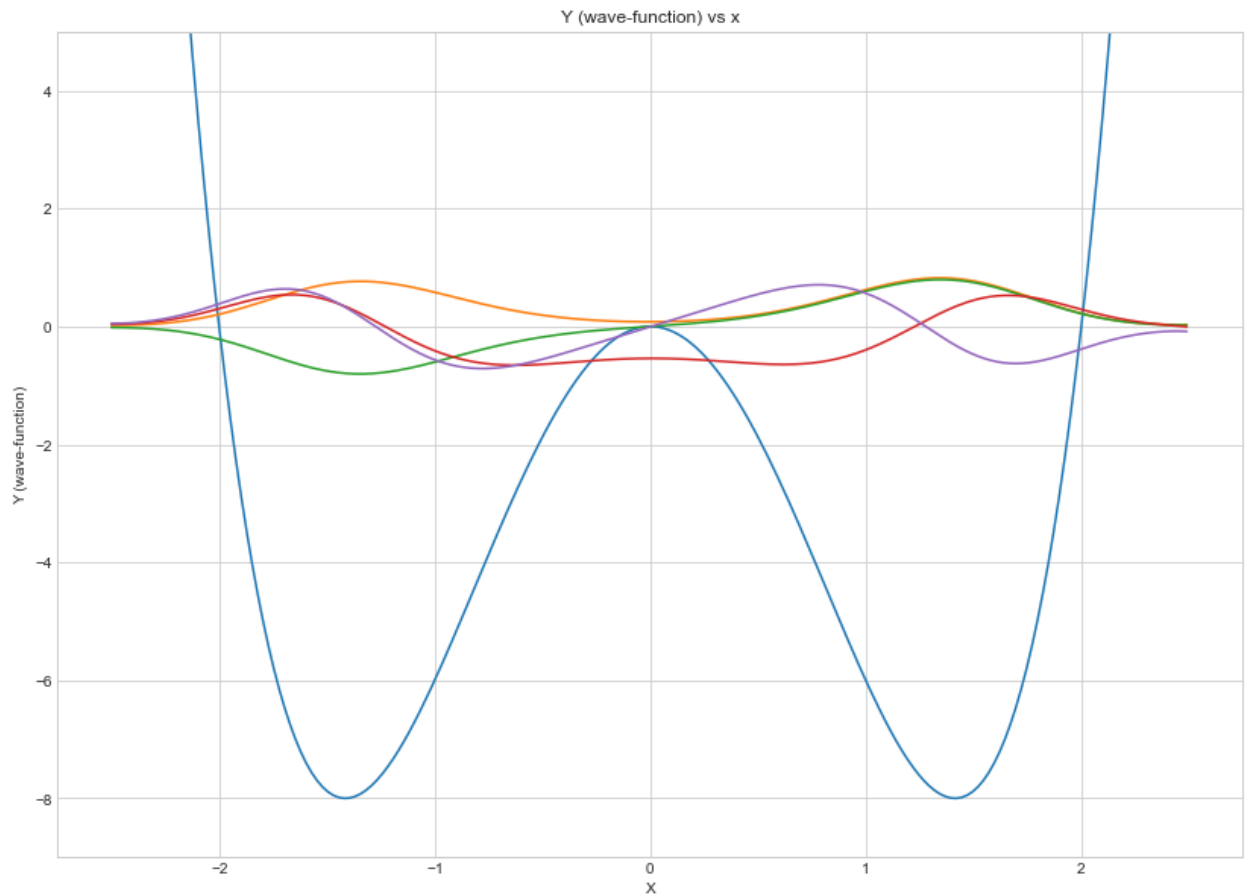
$$E_1 = -5.305000000000004$$

$$E_2 = -1.0199999999999996$$

$$E_3 = -0.3769999999999999$$

Eigen wave functions

```
In [51]: plot_wavefunc_with_V( func_V      = V,
                               axes_wavefunc = axes_doublewell_1_2 + axes_doublewell_3 + axes_doublewell_4,
                               figsize      = (14, 10),
                               rangex_V    = (-2.5, 2.5),
                               ylim        = (-9, 5)
                               )
```



In []:

In []:

Simple Harmonic Oscillator at $x = 2$

```
In [60]: def V(x):
          return 1/2*(x-2)**2

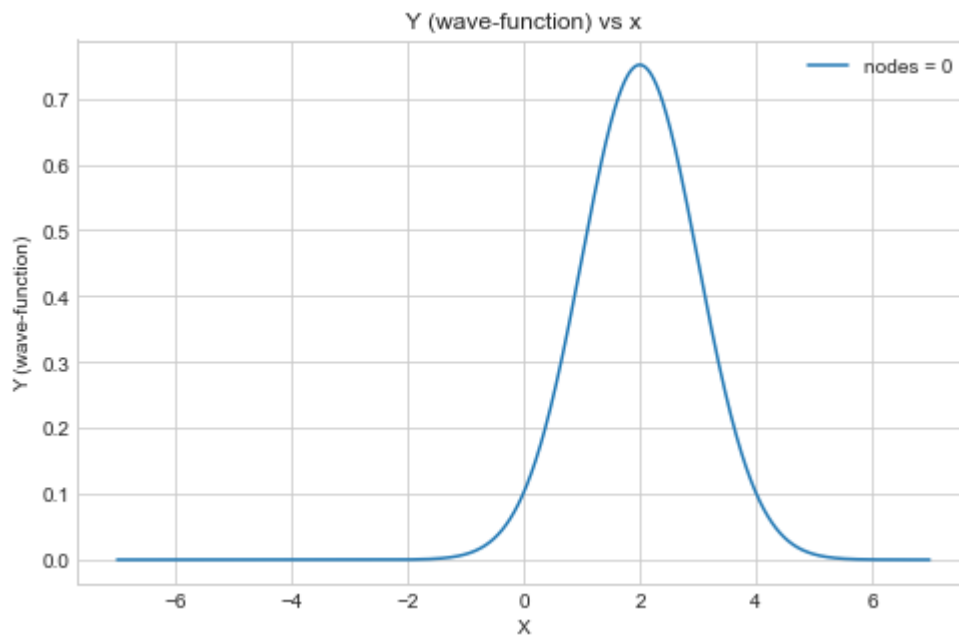
          def d2y(x, y, z):
              return 2*(V(x)-E) * y
```

```
In [61]: eigen_values, axes_sh = solve_schrod_RK_4th( a    = -7,
                                                    b    = 7,
                                                    h    = 0.01,
                                                    d2y  = d2y,
                                                    ya   = 0.0001,
                                                    d1ya  = 0,
                                                    El    = 0,
                                                    Eh    = 4,
                                                    dE    = 0.1,
                                                    match_ratio = 0.01,
                                                    normalized = True )
                                                    # xlim = None,
                                                    # ylim = (0, 10) )

print("eigen_values: ", eigen_values)
```

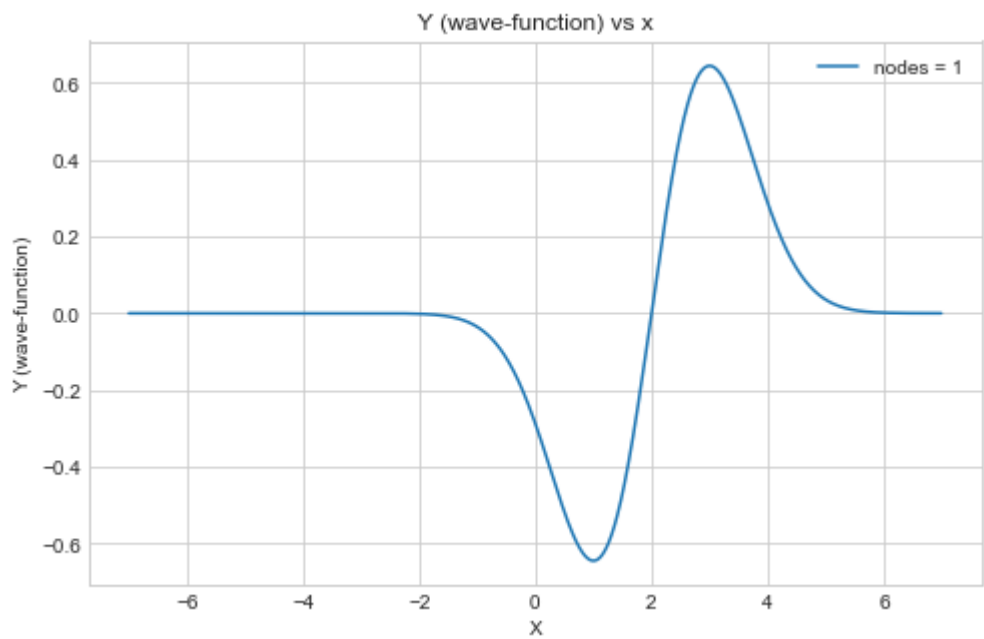
12% | #####

i = 0
E = 0.5



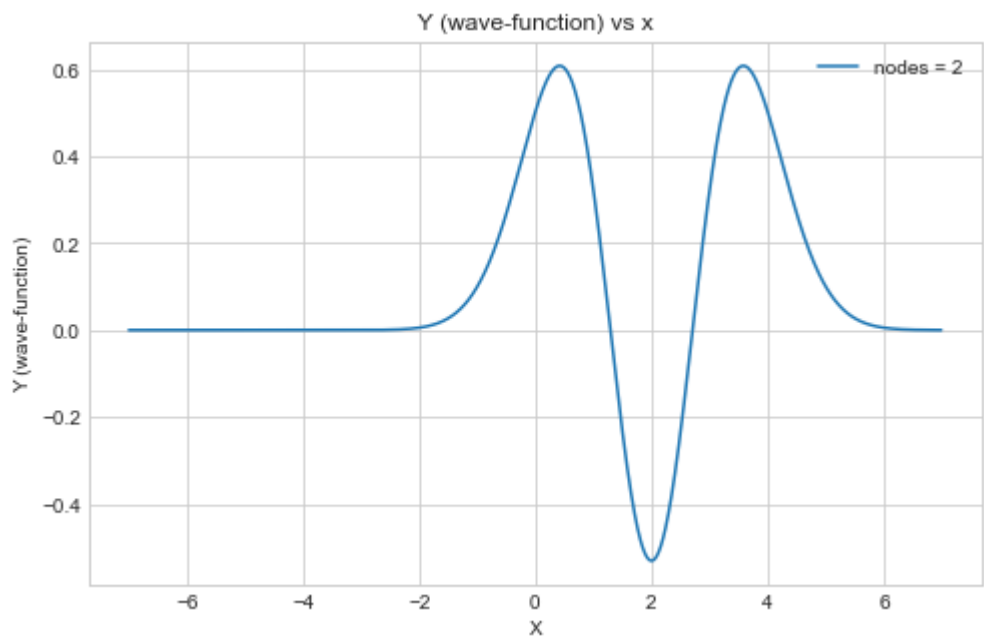
37% | #####

i = 1
E = 1.5



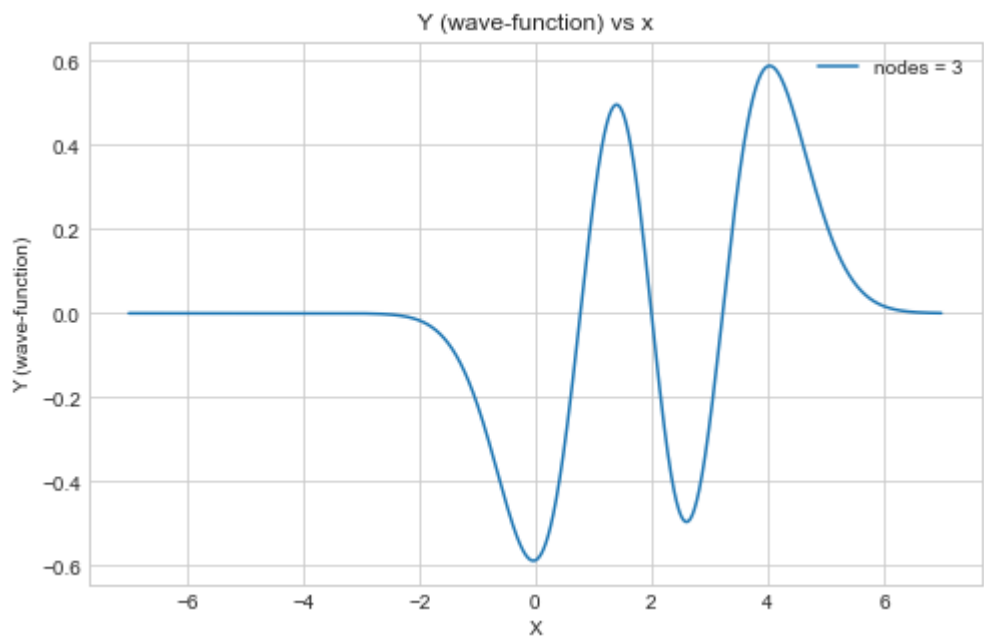
62% | #####

i = 2
E = 2.5



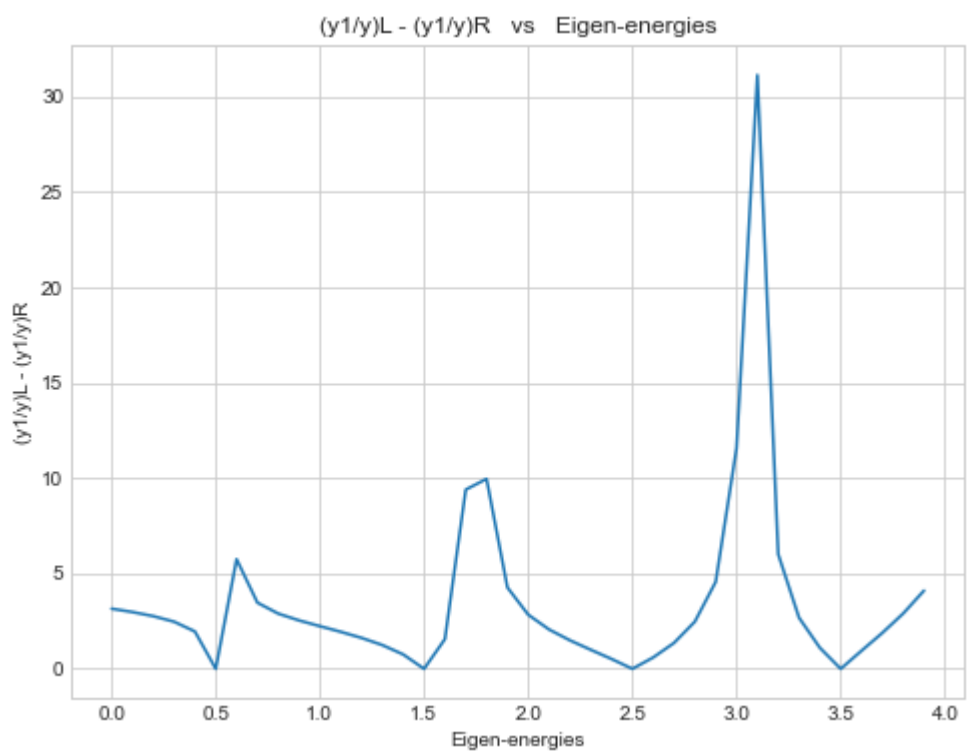
87% | #####

i = 3
E = 3.5

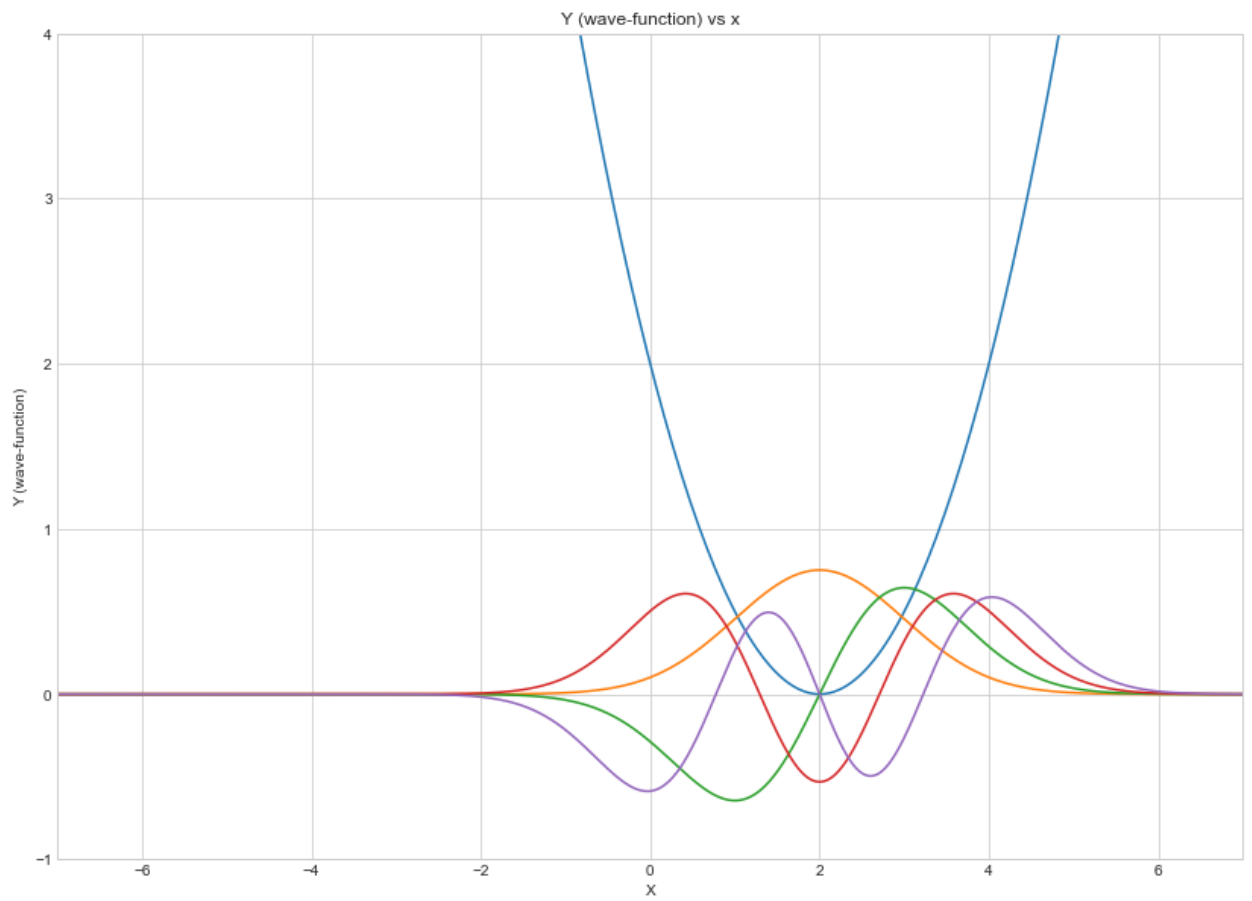


100% | ##### |

eigen_values: [0.5, 1.5, 2.5, 3.5]




```
In [62]: plot_wavefunc_with_V( func_V      = V,  
                                axes_wavefunc = axes_sh,  
                                figsize      = (14, 10),  
                                rangex_V    = (-3, 7),  
                                ylim        = (-1, 4),  
                                xlim        = (-7, 7)  
                                )
```



In []: