Energy Levels of the Finite Square Well

L. Nathan. L. Ngqwebo, NGQLIN011[1]

[1] *University of Cape Town*
Woolsack Drive, Rondebosch, 7701
Cape Town, Western Cape, South Africa

ABSTRACT

A numerical solving of the energy eigenstates of the time-independent Schrodinger equation for the finite square well potential using root-finding techniques.

1. INTRODUCTION

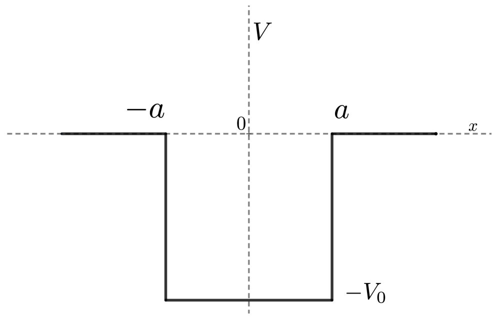

Figure 1. *Finite Square Well Potential*

From the Time Independent Schrodinger Equation (TISE) it follows that:

$$\frac{-\hbar^2}{2m}\frac{d^2}{dt^2}\psi(x) + V(x)\psi(x) = E\psi(x) \tag{1}$$

$$\frac{d^2\psi}{dt^2} = \frac{-2m}{\hbar^2}\left(E - V(x)\right)\psi(x) \tag{2}$$

By noting the symmetry of the potential, the wave function's form can be defined in all three regions of the well shown in Fig.1 where $V(x) = \begin{cases} -V_0\,, & |x| > a \\ 0\,, & -a < x < a \end{cases}$

Taking care to trim the general solutions of non-physical terms, as shown in Griffiths & Schroeter (2018), the **even** solutions in the right half of the well are:

$$\psi(x) = \begin{cases} Fe^{-\kappa x} \; , & (x > a) \\ D\cos(lx) \; , & (0 < x < a) \\ \psi(-x) \; , & (x < 0) \end{cases} \tag{3}$$

where the constants κ and l are defined as follows:

$$\kappa \equiv \sqrt{\frac{-2mE}{\hbar^2}} \tag{4}$$

$$l \equiv \sqrt{\frac{2m\,(E + V_0)}{\hbar^2}} \tag{5}$$

By applying continuity constraints for $\psi(x)$ and $\psi'(x)$ in the even and odd cases on both sides of the boundary at $x = a$, we obtain Eq 6 and Eq 7 whose roots yield the allowed energies for the even and odd wave functions, respectively:

$$\tan(la) - \frac{\kappa}{l} = 0 \tag{6}$$

$$\cot(la) + \frac{\kappa}{l} = 0 \tag{7}$$

For further numerical analysis, the following system constants are provided:

Table 1. *Table of System Constants*

Quantity	Description	Value	Units
a	*half well width*	0.05	[nm]
$-V_0$	*minimum potential*	-40	[eV]
mc^2	*particle's reduced mass*	0.511	[MeV]
$\hbar c$	*physical constant*	197.3	[eV·nm]

TASK 1: ROOT FINDING

Using the bisection method[1], the bound state energy corresponding to the even wave functions within the well was found to be $-26.34\,\text{eV}$. This value represents the midpoint of an interval whose width was required to be at most $0.001\,\text{eV}$. The algorithm used is laid out in Listing 1

[1] The upper and lower bounds for the search interval were informed by plotting the homogeneous form of Eq 6

Listing 1. Bisection Method Root Finding Implementation

```python
import numpy as np

def bisection(func, x_l, x_u) -> tuple:
    threshold = 1e-3
    if abs(x_u - x_l) < threshold:
        print('Root␣Constrained!')
        return(x_l, x_u)
    else:
        x_mid = (x_l + x_u) / 2
        if func(x_l) * func(x_mid) < 0:
            print('going␣left', x_l, x_u)
            return bisection(func, x_l, x_mid)

        elif func(x_mid) * func(x_u) < 0:
            print('going␣right', x_l, x_u)
            return bisection(func, x_mid, x_u)
        else:
            print('Root␣Not␣Found')
            return None


def F(E: float) -> float:
    """
    Definition of the function whose roots are to be found
    """
    l = np.sqrt((2 * mc2 / hc**2) * (E + V_0))
    kappa = np.sqrt((-2 * mc2 / hc**2) * E)

    return l * np.tan(l * a) - kappa
```

TASK 2: BOUND STATES

The odd solutions of the wave function have bound states represented by the roots of Eq 7. Plotting the κ/l term against the *tangent* and *cotangent* terms, reveals the second and final bound state found just below $V(a) = 0$. Figure 2 illustrates the positions of the two energies. Using the Newton-Raphson method outlined in Task 4, the value of this second energy level was found to be $-0.090\,\mathrm{eV}$ (rounded off to three decimal places). Further more, the 'deeper' the potential well, the more bound states are expected to become available. Figure 3 illustrates this nicely for the case of a minimum potential equal to $-350\,\mathrm{eV}$

TASK 3: BINDING ENERGY

By re-purposing the function defined in Listing 1 to vary the minimum potential V_0, for a **fixed** binding energy of $13.6\,\mathrm{eV}$ corresponding to that of the hydrogen atom, the minimum potential was found to be $-24.416\,\mathrm{eV}$. The implementation is the same as shown in Listing 1 except for the function passed to the `bisection` method shown in Listing **??**:

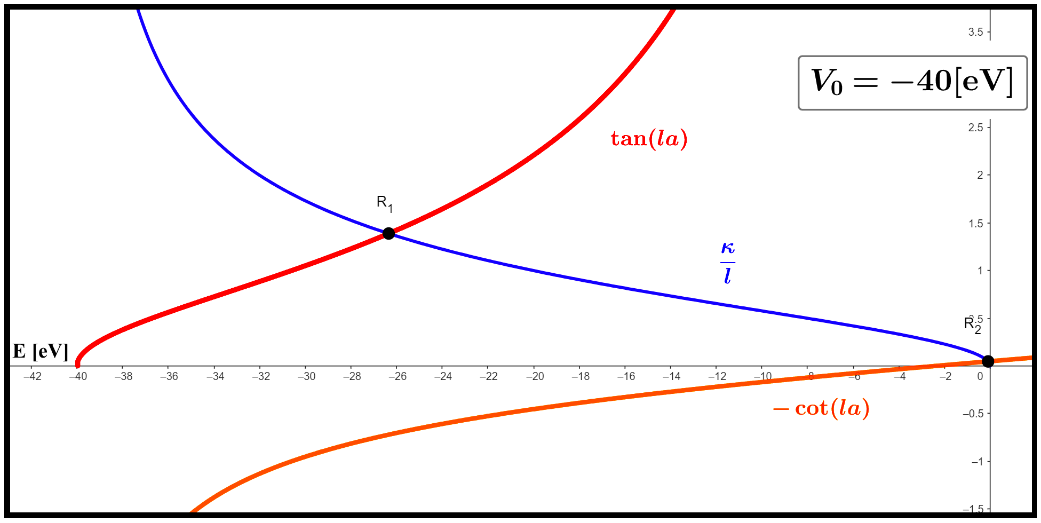

Figure 2. Line Plots of Each term in the Root Finding Equations 6 & 7

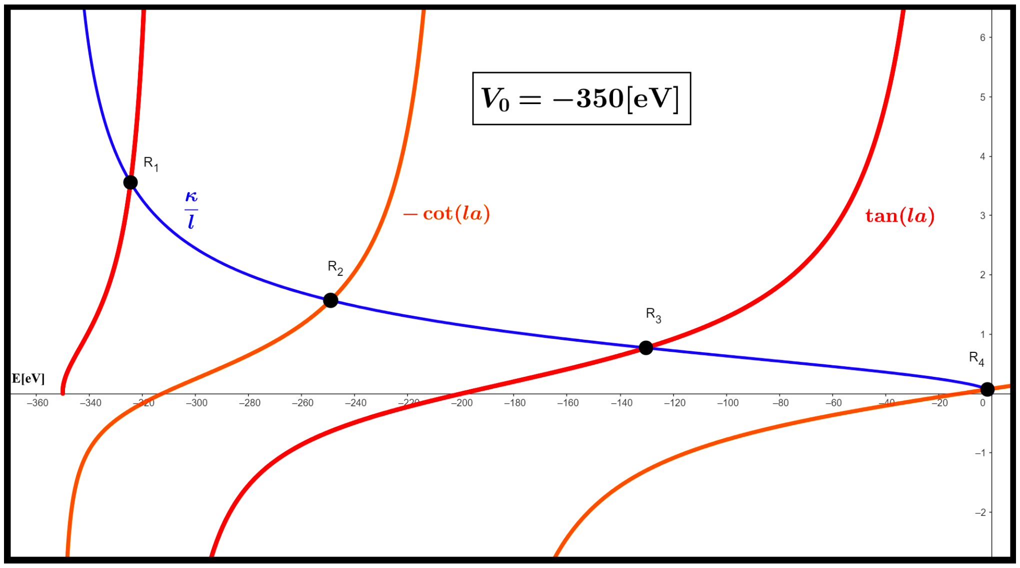

Figure 3. Line Plots Showing More Bound States for a Deeper Potential Well

Listing 2. Function for Finding Minimum Potential

```
def G(v_0: float) -> float:
    E = -13.6
    l = np.sqrt((2 * mc2 / hc**2) * (E + v_0))
    kappa = np.sqrt((-2 * mc2 / hc**2) * E)

    return l * np.tan(l * a) - kappa
```

TASK 4: NEWTON-RAPHSON METHOD

Leveraging, prior knowledge of the analytical forms of both the function of interest (Eq 6) and it's derivative, the Newton-Raphson method can be employed to cut down computational time in the root-finding. The SymPy implementation shown in Listing 3 took only five iterations to converge to a root which agreed with the previous iterations to within 13 decimal places, whereas the root found using the bisection method took 14 iterations for its intervals to agree within just 2 decimal places. The implementation went as follows:

Listing 3. Newton-Raphson Root Finding Implementation

```python
import sympy as sp

mc2, V_0, E, h_c, f, F, a = \
sp.symbols('mc^2␣V_0␣E␣(\\hbar*c)␣f␣F␣a', real=True)
k, l = sp.symbols('kappa␣l', cls=sp.Function, real=True)
k = k(E)
l = l(E)

# Function definition
l = sp.sqrt((2 * mc2 / h_c**2) * (E + V_0))
k = sp.sqrt(((-2 * mc2 / h_c**2) * E))
F = sp.tan(l * a) - k / l
f = sp.diff(F, E)
G = sp.cot(l * a) + k / l
g = sp.diff(G, E)

E_i_e = -39
E_i_o = -0.01
n = 20

# Define specific values for the variables
values = {mc2: 0.511e6 , V_0: 40, h_c: 197.3, a: 0.05}

for i in range(n):
    E_i_e = E_i_e - float(F.subs({E: E_i_e, **values})) \
    / float(f.subs({E: E_i_e, **values}))
    print(f"even␣root:␣{E_i_e}")
print()
for i in range(n):
    E_i_o = E_i_o - float(G.subs({E: E_i_o, **values})) \
    / float(g.subs({E: E_i_o, **values}))
    print(f"odd␣root:␣{E_i_o}")
```

The roots converged to are: $E_0 = -26.343\,\text{eV}$ and $E_1 = -0.091\,\text{eV}$ (rounded off). The initial value guesses were, again, informed by plotting the functions but the guesses were purposefully placed somewhat far from the root to better compare performance between the two root-finding algorithms.

TASK 5: GROUND STATE WAVE FUNCTION

By working through the normalisation of the wave function defined in Eq 3, the constants F and D could be found:

$$F = e^{\kappa a}\cos(la)\,(a + 1/\kappa)^{-1/2}$$
$$D = (a + 1/\kappa)^{-1/2}$$

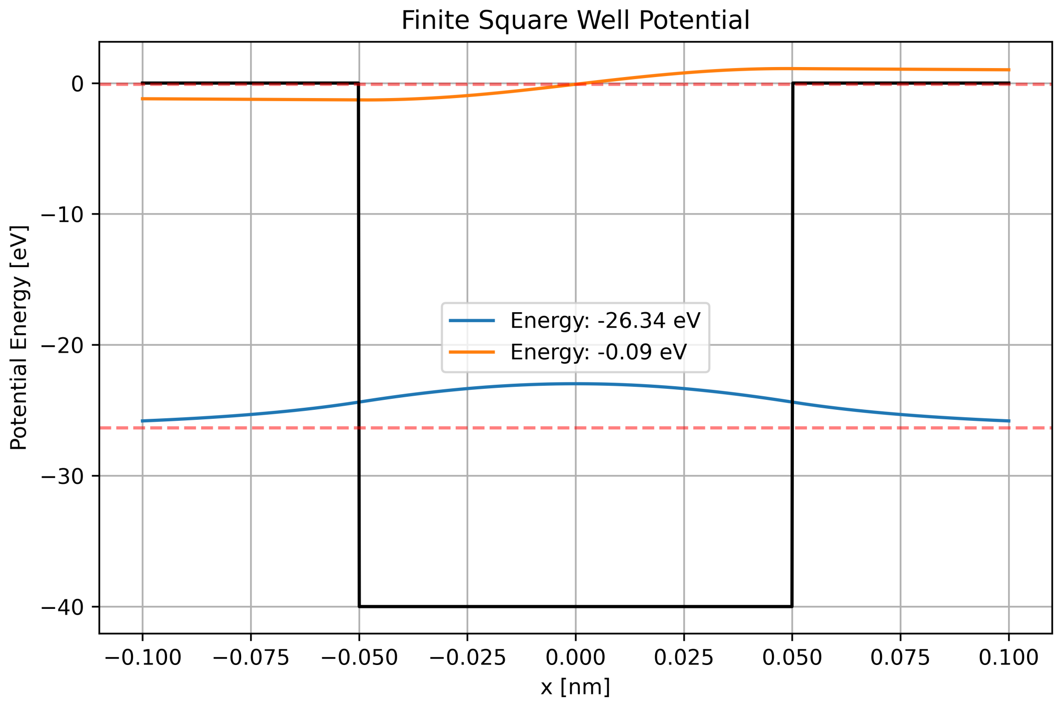

Figure 4. Plot of Two Bound State Wave Functions in the Finite Square Well

Thus, the even wave function solutions become:

$$\psi(x) = \begin{cases} (a+1/\kappa)^{-1/2}\cos(la)\; e^{-\kappa(x-a)}\ , & (x > a) \\ (a+1/\kappa)^{-1/2}\cos(lx)\ , & (0 < x < a) \\ \psi(-x)\ , & (x < 0) \end{cases} \tag{8}$$

and the odd solutions become:

$$\psi(x) = \begin{cases} (a+1/\kappa)^{-1/2}\sin(la)\; e^{-\kappa(x-a)}\ , & (x > a) \\ (a+1/\kappa)^{-1/2}\sin(lx)\ , & (0 < x < a) \\ -\psi(-x)\ , & (x < 0) \end{cases} \tag{9}$$

Finally, plotting these out 8 and 9 produce Figure 4 Listing 4 describes the plotting procedure:

Listing 4. Wave Function Plotting Procedure

```
def wave_function(x, E:float, parity:str) -> float:
    """
    Takes an array of positions and energy for a bound state.
    Returns an array of wave function values for those positions.
    checks parity of bound state and adjusts output accordingly
    """
    l = np.sqrt((2 * mc2 / hc**2) * (E + V_0))
    kappa = np.sqrt((-2 * mc2 / hc**2) * E)
    A = 1 / np.sqrt(a + 1 / kappa)

    # Region IIII
    mask_IIII = x > a
    # Region III
    mask_III = np.logical_and(x >= 0, x <= a)
    # Region II (By Symmetry)
    mask_II = np.logical_and(x >= -a, x <= 0)
    # Region I (By Symmetry)
    mask_I = x < -a

    if parity == 'even':
        return np.piecewise(x=x,
            condlist=[mask_IIII, mask_III, mask_II, mask_I],
            funclist=[
                lambda x: A * np.cos(l * a) * np.exp(-kappa * (x - a)),
                lambda x: A * np.cos(l * x),
                lambda x: A * np.cos(l * -x),
                lambda x: A * np.cos(l * a) * np.exp(-kappa * (-x - a)\
                )])
    elif parity == 'odd':
        return np.piecewise(x=x,
            condlist=[mask_IIII, mask_III, mask_II, mask_I],
            funclist=[
                lambda x: A * np.sin(l * a) * np.exp(-kappa * (x - a)),
                lambda x: A * np.sin(l * x),
                lambda x: -A * np.sin(l * -x),
                lambda x: -A * np.sin(l * a) * np.exp(-kappa * (-x - a)\
                )])
    else:
        print("Neither???")
        return None
```

Software: Python (Van Rossum & Drake 2009), SymPy (Meurer et al. 2017)

REFERENCES

Griffiths, D., & Schroeter, D. 2018, Introduction to Quantum Mechanics (Cambridge University Press), 93–94. https://books.google.co.za/books?id=82FjDwAAQBAJ

Meurer, A., Smith, C. P., Paprocki, M., et al. 2017, PeerJ Computer Science, 3, e103, doi: 10.7717/peerj-cs.103

Van Rossum, G., & Drake, F. L. 2009, Python 3 Reference Manual (Scotts Valley, CA: CreateSpace)