Square Well Potential

Ben Pearson

November 16, 2021

Contents

1 Section I Physics

The square well potential is an analysis of the wave function of a particle in one dimension near a well with a negative potential. Splitting the problem in two, we can analyze this case in two cases:

1.1 Even Parity

In this situation $\psi(-x) = \psi(x)$ Here we aim to solve the Schrodinger equation as

$$\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + V(x)\Psi = E\Psi$$

We can break this down as

$$\begin{aligned} \frac{d^2\psi}{dx^2} + \frac{2m}{\hbar^2}\left(E + V_0\right)\Psi = 0, |x| \leq a \\ \frac{d^2\psi}{dx^2} + \frac{2m}{\hbar^2}E\Psi = 0, |x| > a \end{aligned} \tag{1}$$

The solution to 1 can be broken down as

$$\begin{aligned} \Psi(x) &= ce^{\beta x}, && x < -a \\ &= Bcos\alpha x, && -a \leq x \leq a, && \alpha = \sqrt{\frac{2m(E+V_0)}{\hbar^2}} \\ &= ce^{-\beta x}, && x > a, && \beta = \sqrt{\frac{-2mE}{\hbar^2}} \end{aligned}$$

In order to solve 2 we need to implement some boundary conditions on the problem:

$$\Psi(a) \Rightarrow Bcos(\alpha a) = Ce^{-\beta a} \tag{3}$$

$$\frac{d\Psi(a)}{dx} \Rightarrow -\alpha Bsin(\alpha a) = \beta Ce^{-\beta a} \tag{4}$$

In 3 and 4 we enforce continuity for the bounds on x. Dividing 3 by 4 we get that

$$tan\alpha a - \frac{\alpha}{\beta} = 0 \tag{5}$$

We use 5 to declare two new variables $\zeta = \alpha a$ and $\eta = \beta a$. We can now define a new variable k so that

$$k \equiv \zeta^2 + \eta^2 = \left(\alpha^2 + \beta^2\right)a^2 = \frac{2mV_0a^2}{\hbar^2} \tag{6}$$

Which leads to the transcendental equation:

$$\zeta tan\zeta = \sqrt{k - \zeta^2} \tag{7}$$

1.2 Odd Parity

With odd parity $\psi(-x) = -\psi(x)$ This results in many of the same equations with the exception of 2 and it's derivative equations.

$$\begin{aligned}
\Psi(x) &= ce^{\beta x}, && x < -a \\
&= Bsin\alpha x, && -a \leq x \leq a, && \alpha = \sqrt{\frac{2m(E+V_0)}{\hbar^2}} \\
&= ce^{-\beta x}, && x > a, && \beta = \sqrt{\frac{-2mE}{\hbar^2}}
\end{aligned}$$

Which leads to

$$\Psi(a) \Rightarrow Bsin(\alpha a) = Ce^{-\beta a} \tag{9}$$

$$\frac{d\Psi(a)}{dx} \Rightarrow \alpha Bcos(\alpha a) = \beta Ce^{-\beta a} \tag{10}$$

And, following the same steps as before, we end up with the equation

$$tan\zeta = -\frac{\zeta}{\sqrt{k-\zeta^2}} \tag{11}$$

1.3 Transcendental equations

The solution for each parity state is what is known as a transcendental equation, and this particular breed cannot be solved analytically. However, we can implement a numerical solution for it.

2 Section II Numerical Solution

There are a number of ways to numerically solve transcendental equations, we need ones that are iterative in nature so that they can be implemented simplistically in a program.

2.1 Functional iteration

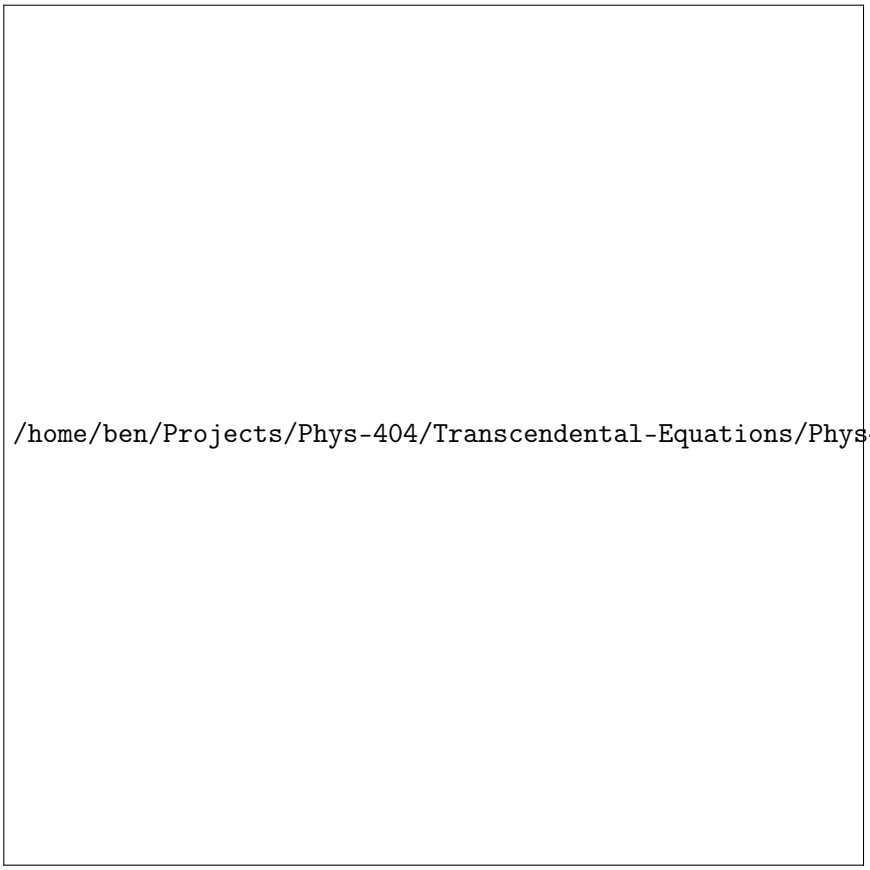
Functional iteration is a relatively simple idea. First we take our equation and solve it for x, that is $g(x) = x$. Then we can say that $x_{i+1} = g(x_i)$. We essentially use the function $g(x)$ as a recursive method to find a good value for x - efficiency depends upon the initial guess, but should be able to converge to a single value regardless of the initial value (barring any out of bounds conditions).

2.2 Newton's Method

Newton's method is slightly more complicated, but more intuitive. Instead of solving for x we solve for 0 so that $f(x) = 0$. We then find the derivative of $f(x)$, $f'(x)$. We then iterate to find values of x through $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$. The second term, $\frac{f(x_i)}{f'(x_i)}$, is essentially steering the value of x_{i+1} through the tangent of the curve. By doing so, it converges faster because it modifies it's iteration depending on the tangent at that point. However, it can run into the issue of getting stuck around local minimums due to this methodology.

3 Section III Program

3.1 Flow chart



3.2 Constants.py

This file contains the constants used in this program across files.

```
# Written by Ben Pearson
# V0.0.1
"""
This file is to store the constants for my program
"""
# Imports
import numpy as np
# Constants
K = 16.032280
# Number of iterations for function
N = 20
```

3.3 Functions.py

This represented the bulk of the work for this project. Finding the right formulas so that they converge properly was primarily done through trial and error of various configurations of the given formulas. For even parity Newton's method ($f(x)$) and Functional Iteration ($g(x)$) are

$$f(\zeta) = \sqrt{k} \times |cos(\zeta)| - \zeta f'(\zeta) = -\sqrt{k} \times \frac{cos(\zeta) \times sin(\zeta)}{|cos(\zeta)|} - 1 g(\zeta) = arccos\left(\frac{\zeta}{\sqrt{k}}\right)$$

And for odd parity it is

$$f(\zeta) = \sqrt{k} \times |sin\zeta| - zeta f'(\zeta) = \sqrt{k} \times \frac{cos(\zeta) \times sin(\zeta)}{|sin\zeta|} - 1 g(\zeta) = arcsin(\frac{\zeta}{\sqrt{k}})$$

```
# Written by Ben Pearson
# V0.1.1

"""
This file will hold the primary functions to be used in the program
"""

# Modules
import numpy as np
import constants as c
```

```
# Newton's method

# Test function
def n_test(x):
    return 3 * (1 - np.exp(-x)) - x


def d_n_test(x):
    return 3 * np.exp(-x) - 1


# Even parity function
def n_even(x):
    return np.sqrt(c.K) * np.abs(np.cos(x)) - x


def d_n_even(x):
    return -np.sqrt(c.K) * np.cos(x) * np.sin(x) / np.abs(np.cos(x)) - 1


# Odd parity funciton
def n_odd(x):
    return np.sqrt(c.K) * np.abs(np.sin(x)) - x


def d_n_odd(x):
    return np.sqrt(c.K) * np.cos(x) * np.sin(x) / np.abs(np.sin(x)) - 1


# Functional Iteration

# Test function
def f_test(x):
    return 3 * (1 - np.exp(-x))


# Even parity function
def f_even(x):
    return np.arccos(x / np.sqrt(c.K))
```

```
# Odd parity function
def f_odd(x):
    return np.arcsin(x / np.sqrt(c.K))
```

3.4 Functional iteration.py

This file was in charge of handling functional iteration of the functions found in functions.py

```
# Written by Ben Pearson
# V0.0.1
"""
This file contains the functional iteration methods
"""
# import modules
import constants as c

def functional_iteration(func, var):
    """Run functional iteration with a starting value of x on the function func"""
    return func(var)

# Take function and iterate it c.N times
def itor(func, var):
    """Run functional iteration with a starting value of x on the function func c.N ti
    arr = [var]
    for i in range(c.N):
        var = functional_iteration(func, var)
        arr.append(var)
    return arr
```

3.5 Newton.py

This file took care of applying Newton's method.

```
# Written by Ben Pearson
# V0.0.1

"""
This module contains newton's methods
```

```
"""
# import modules
import constants as c

def newton_method(func, d_func, var):
    """Run newton's method on the function func with a starting value of var"""
    return var - func(var) / d_func(var)

def itor(func, d_func, var):
    """Run newton's method on the function func with a starting value of x c.N times"""
    arr = [var]
    for i in range(c.N):
        var = newton_method(func, d_func, var)
        arr.append(var)
    return arr
```

3.6 Main.py

This file brought together the various modules and implementing them as well as data output.

```
# Written by Ben Pearson
# V0.0.3

"""
This module implements the modules funcitonal_iteration and newton.
"""

# Modules
import constants as c
import functions as f
import newton
import functional_iteration
import pandas as pd


def print_arrs(f_arr, n_arr):
    print("\tFuncitonal Iteration\tNewton's Method\tDifference")
    for i in range(c.N):
        if c.N > 100:
```

```
            if i % 20 == 0:
                print(
                    str(i)
                    + "\tx =\t"
                    + str(f_arr[i])
                    + ",\tx =\t"
                    + str(n_arr[i])
                    + ",\tdelta =\t"
                    + str(f_arr[i] - n_arr[i])
                )
        else:
            print(
                "x =\t"
                + str(f_arr[i])
                + ",\tx =\t"
                + str(n_arr[i])
                + ",\tdelta =\t"
                + str(f_arr[i] - n_arr[i])
            )


x = 3
data = {
    "f_test": functional_iteration.itor(f.f_test, x),
    "n_test": newton.itor(f.n_test, f.d_n_test, x),
    "f_even": functional_iteration.itor(f.f_even, x),
    "n_even": newton.itor(f.n_even, f.d_n_even, x),
    "f_odd": functional_iteration.itor(f.f_odd, x),
    "n_odd": newton.itor(f.n_odd, f.d_n_odd, x),
}
print_arrs(data["f_test"], data["n_test"])
print_arrs(data["f_even"], data["n_even"])
print_arrs(data["f_odd"], data["n_odd"])

df = pd.DataFrame(data)
df.to_csv("./data.csv")
```

4 Section IV Analysis

Below are the values found for ζ. As you can see, despite having the same initial functions, Newton's method and functional iteration are finding different roots. Newton's method finds a stable minimum locally, so the root it finds is mostly dependent upon the starting point. For example, a starting point of 1 quickly converges to the same value as functional iteration for even parities. Similarly, for Odd parity, placing the initial value too close to 0 results in a convergence on zero, hence why 3 was chosen to avoid such a case.

	Functional Even	Newton's Even	Functional Odd	Newton's Odd
0	3	3	3	3
1	0.7238755966	5.21625644	0.8469207	2.509474
2	1.3890101	6.52611136	0.2131267	2.475512
3	1.216529546	5.18151211	0.0532532	2.475178
4	1.262090332	6.49254737	0.0133003	2.475178
5	1.250123900	5.08660202	0.0033217	2.475178
6	1.253271385	6.41518729	0.00082956	2.475178
7	1.252443831	4.81336863	0.0002072	2.475178
8	1.252661438	6.29134034	5.174531e-5	2.475178
9	1.252604219	4.07622917	1.2923298e-5	2.475178
10	1.25261926	3.67407813	3.227571e-6	2.475178
11	1.252615309	3.60008202	8.060800e-7	2.475178
12	1.252616349	3.59662727	2.013170e-7	2.475178
13	1.252616075	3.59661951	5.02785556e-8	2.475178
14	1.25261614	3.5966195	0	2.475178
15	1.25261612	3.5966195	0	2.475178
16	1.252616133	3.5966195	0	2.475178
17	1.252616132	3.5966195	0	2.475178
18	1.252616132	3.5966195	0	2.475178
19	1.252616132	3.5966195	0	2.475178
20	1.252616132	3.5966195	0	2.475178