AEP 4380: Homework 7

Gregory Kaiser

November 8th, 2019

1 Problem Background and Solution Overview

The one-dimensional time-dependent Schrödinger equation is important to understanding quantum systems in general. In many introductory-level quantum mechanics courses, the interaction of a quantum mechanical plane wave with a step function potential is discussed in order to motivate transmission and reflection across a potential barrier. By solving the Schrödinger equation numerically, one has the freedom not only to manipulate the potential barrier but also the initial wavefunction itself. This report investigates the interaction of a gaussian quantum-mechanical wave packet with a finite potential step.

The time-dependent Schrödinger equation, with a potential function $V(x)$, is:

$$i\hbar\frac{\partial}{\partial t}\psi = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\psi(x,t) + V(x)\psi(x,t) \tag{1}$$

Where $\hbar = 6.5821 \times 10^{-16} eVs$ and $\frac{\hbar}{2m} = 3.801 eVA^2$ for an electron[2]. The potential step function is an imperfect one, that is the potential function is not a sudden Heaviside step, but rather a very sharp S-shaped curve:

$$V(x) = \frac{V_0}{1 + exp[\frac{0.5L - x}{\omega_v}]} \tag{2}$$

where $L = 1000$ Angstroms, $V_0 = 3.90$eV and $\omega_v = 7$ Angstroms. The initial condition for the wavefunction, $\psi(x,0)$, is a Gaussian packet which travels to the right as time progresses:

$$\psi(x, t = 0) = exp\left[-\left(\frac{x - 0.3L}{s}\right)^2 + ixk_0\right] \tag{3}$$

where $s = 20$ Angstroms and $k_0 = 1$ Angstrom^{-1}. This packet begins its journey to the left of the potential step, where $V(x) = 0$:

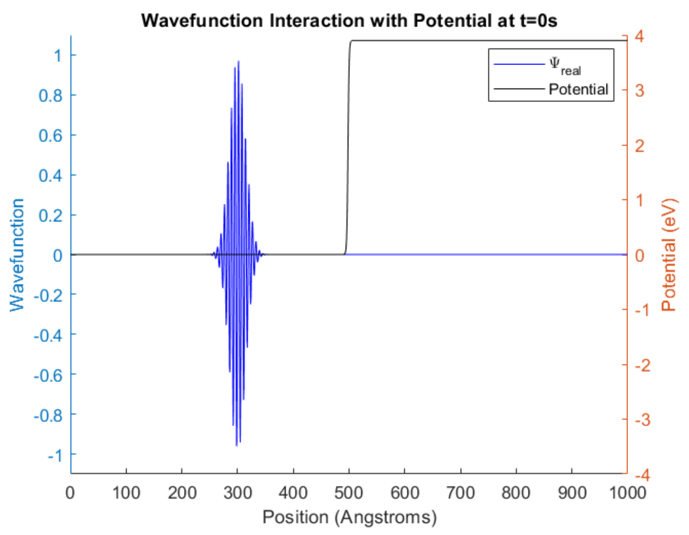


Figure 1: The real component of ψ at t=0.

Intuitively, it should then collide with the potential barrier and transmit/reflect depending on the wavepacket strength and potential barrier height. The solution was calculated along a range of 1000 Angstroms and for times between 1e-14 and 5e-14 seconds.

As a convention, $\psi(x,t)$ will now be denoted ψ_x^t or ψ_i^j.

2 Solution Description

In order to solve this problem, the function ψ must be solved over a range in space and propagated to the next point in time. Since wavefunction solutions to Schrödinger's equation are complex in general, a complex data type now in C++ must be used. In tandem with the arrayt class, arrays of complex numbers are generated easily [2][3].

First ψ is initialized as a complex array, along with appropriate constant values, and filled with ψ_x^0 with a spacing of dx. Since the complex and real parts of ψ are hidden in their exponential form, Euler's Identity breaks up ψ_x^0 into its component real and imaginary components. Since the average wavenumber is 1 Angstrom^{-1}, and since $k_0 = \frac{2\pi}{\lambda_0}$, $\frac{\lambda}{10} \approx 0.6$. By overlaying plots of ψ_x^0, and ensuring that dx was less than the smallest wavelength component of ψ, dx was maintained at 0.1 Angstroms to be on the safe side.

Since we are given a starting function for ψ_x^0 we can use the Crank-Nicholson method to approximate the solution for $\psi_x^{\Delta t}$ at the next time step. In finite difference form, the Schrödinger equation is then given by:

$$\psi_{x-\Delta x}^{t+\Delta t} + \left[\frac{2mi\omega}{\hbar} - 2 - \frac{2m\Delta x^2}{\hbar^2}V(x)\right]\psi_x^{t+\Delta t} + \psi_{x+\Delta x}^{t+\Delta t} = -\psi_{x-\Delta x}^t + \left[\frac{2mi\omega}{\hbar} + 2 + \frac{2m\Delta x^2}{\hbar^2}V(x)\right]\psi_x^t - \psi_{x+\Delta x}^t \tag{4}$$

where $w = 2\Delta x^2/\Delta t$ [2]. This equation can be written as a tridiagonal system of equations:

$$\begin{bmatrix} b_0 & c_0 & & & & \\ a_1 & b_1 & c_1 & & 0 & \\ & a_2 & b_2 & c_2 & & \\ & & & \ddots & & \\ & 0 & & a_{N_x-2} & b_{N_x-2} & c_{N_x-2} \\ & & & & a_{N_x-1} & b_{N_x-1} \end{bmatrix} \begin{bmatrix} \psi_0 \\ \psi_1 \\ \psi_2 \\ \vdots \\ \psi_{N_x-2} \\ \psi_{N_x-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{N_x-2} \\ d_{N_x-1} \end{bmatrix}$$

[2] such that:

$$a_i\psi_{i-1}^{n+1} + b_i\psi_i^{n+1} + c_i\psi_{i+1}^{n+1} = d_i \tag{5}$$

A tridiagonal matrix can be solved by converting the matrix to upper diagonal form and then substituting to solve for unknown values:

```
//step 1, do the special first row case of the process
c(0) = c(0)/b(0);
d(0) = d(0)/b(0);

//step 2, replace all the c values with their value if a<--0 and b<--1 "ZIP"
for(i=1; i<=n-2; i++){
    c(i) = c(i)/(b(i)-a(i)*c(i-1));
}
//step 2, replace all the d values with their value if a<--0 and b<--1 "ZIP"
for(i=1; i<=n-1; i++){
    d(i) = (d(i)-a(i)*d(i-1))/(b(i)-a(i)*c(i-1));
}

//step 3, set last solution to d
soln(n-1) = d(n-1);

//backpropagate the solution "ZAP"
for(i=n-2; i>=0; i--){
    soln(i) = d(i)-c(i)*soln(i+1);
}
```

where the above was made as a template so that any arbitrary matrix can be solved in this manner[2]. Since a_i, b_i, c_i, and d_i are complex valued, arithmetic works normally using the C++ complex type.

However, the values of a_i, b_i, c_i, and d_i must be recalculated at each step in time. Following equation 4 it is clear that $a_i = 1 + 0i$ and $c_i = 1 + 0i$ for each time step. The complex arrays b_i and d_i take on more complicated values based on the complex and real parts of the multiplicative factor on $\psi(x, t + \Delta t)$ and the terms on the right hand side of equation 4. These terms also involve the potential function at that value of x.

$$b_i = \left[\frac{2mi\omega}{\hbar} - 2 - \frac{2m\Delta x^2}{\hbar^2}V(i)\right] \tag{6}$$

$$d_i = -\psi^j_{i-1} + \left[\frac{2mi\omega}{\hbar} + 2 + \frac{2m\Delta x^2}{\hbar^2}V(i)\right]\psi^j_i - \psi^j_{i+1} \tag{7}$$

To keep the algorithm simpler, a_i, b_i, c_i, and d_i were made to be the same length. In tridiag, a_0 and c_{N_x-1} are ignored, though they are set to 1 by default while being calculated.

In order to account for the fact that ψ_{-1} and ψ_{N_x} are held at zero, those terms are omitted at the boundary while calculating d_0 and d_{N_x-1}.

Once those arrays are generated, the solver described above, ghk_tridiag, is called, which takes arrayt's of arbitrary type and fills the final argument with $\psi_x^{t+\Delta t}$ (the solution at the next timestep).

A for loop controls the propagation in time, so by tuning the time t_{max}, the algorithm ends at any specified time for printing to an output file.

By generating a few plots with different time steps dt, but while holding the final t constant, a dt of 1e-16 was determined to be plenty precise enough to be safe for production of final results, while keeping compute time at a reasonable level of a couple seconds.

3 Results and Interpretation

The imaginary and real parts of ψ and $V(x)$ are plotted for t=0 to show all initial conditions:

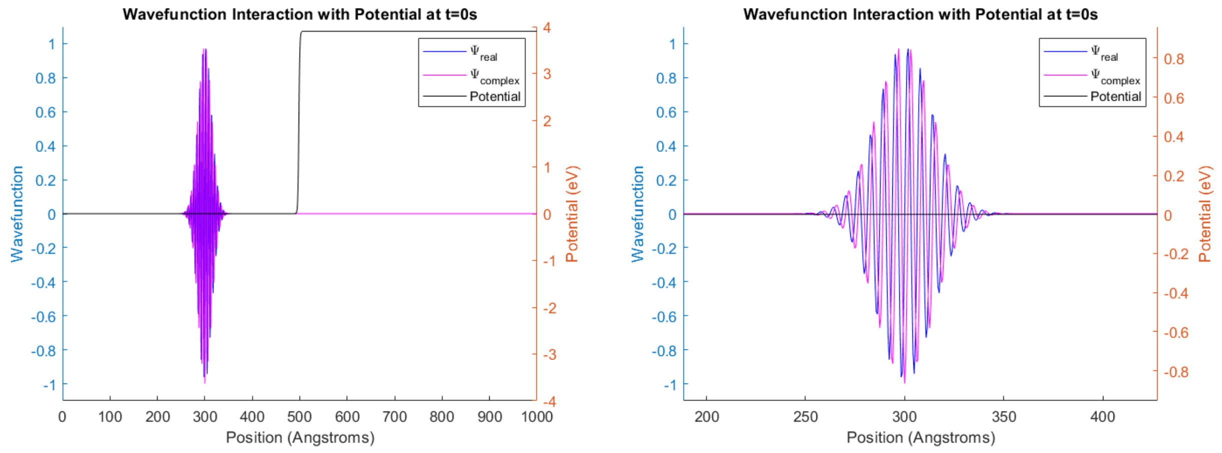


Figure 2: Left: The real and complex parts of ψ at t=0. Right: A detailed view of the real and complex parts.

A plot of $|\psi|^2$ at t=0 can be found in Figure 3.

The following plots show $|\psi|^2$ plotted at times t=0, 1e-14, 2e-14, 3e-14, 4e-14, and 5e-14 seconds of propagation. The integral of $|\psi|^2$ remained constant around 25.

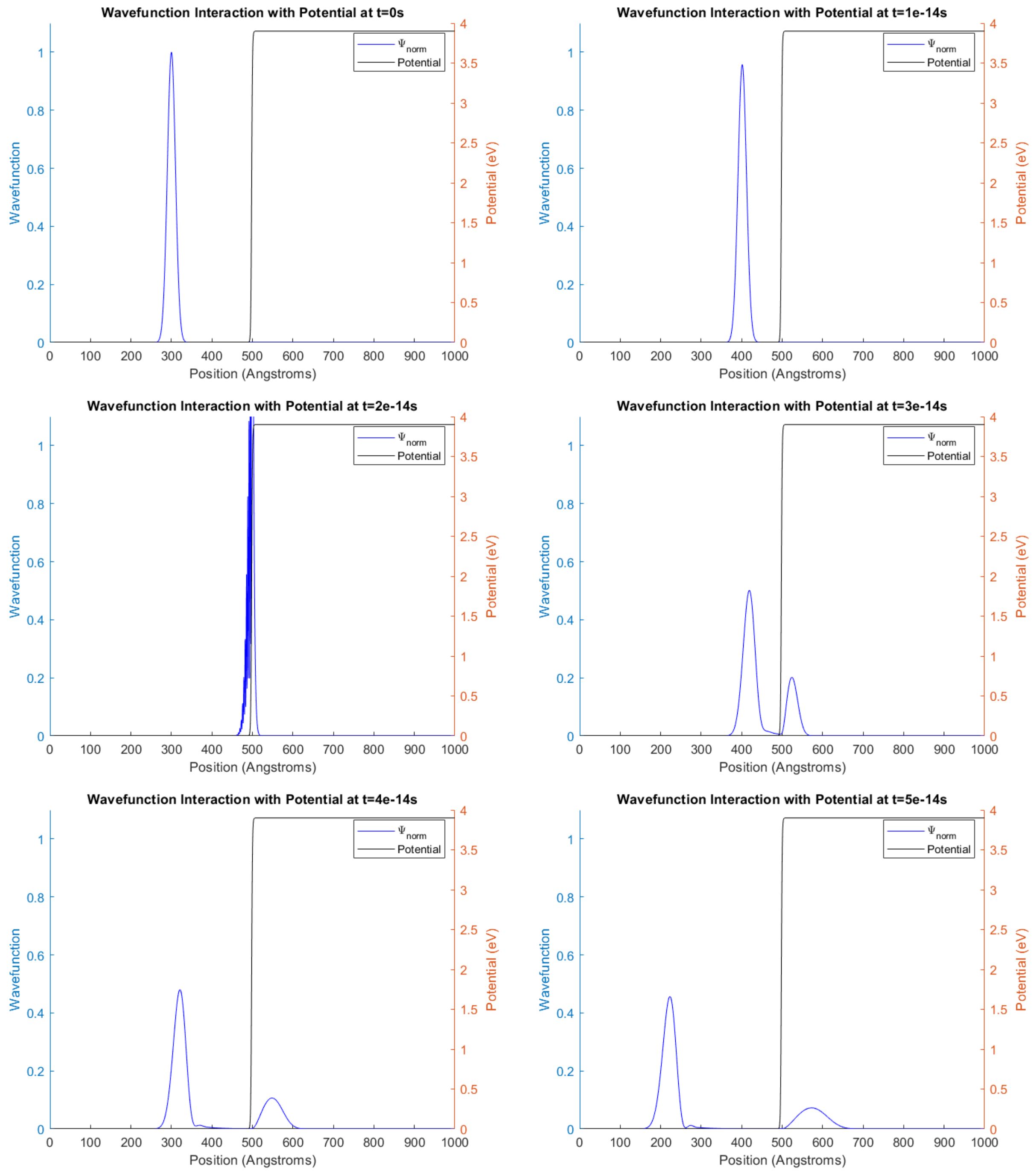


Figure 3: The norm of the wavefunction $|\psi|^2$ plotted at different times to show time-dependent propagation

These results make physical sense, as it is clear that some of the wavefunction propagates into the finite step, as a quantum wave should, while some significant chunk is reflected backwards. The most interesting part of these results is the secondary "hump" on the right tail of the reflected wave. In order to check that this isn't an error, this algorithm was tested using a "perfect" Heaviside step function (instantaneous change) with much higher potential, which showed a more clean reflection of the wave as expected:

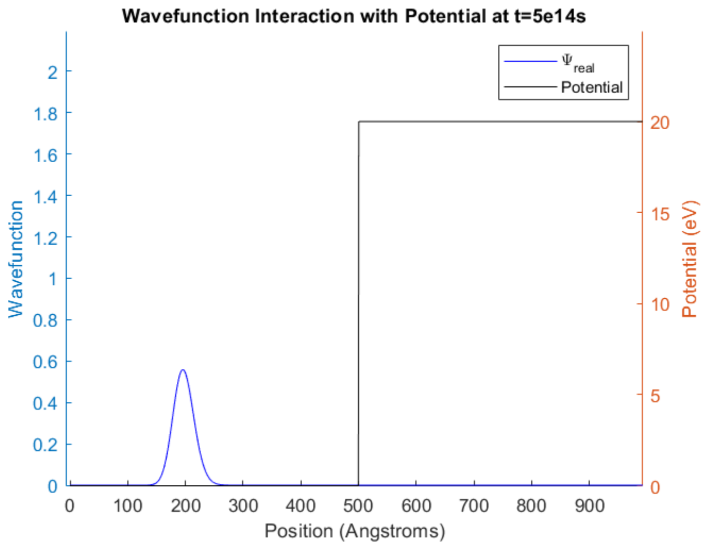
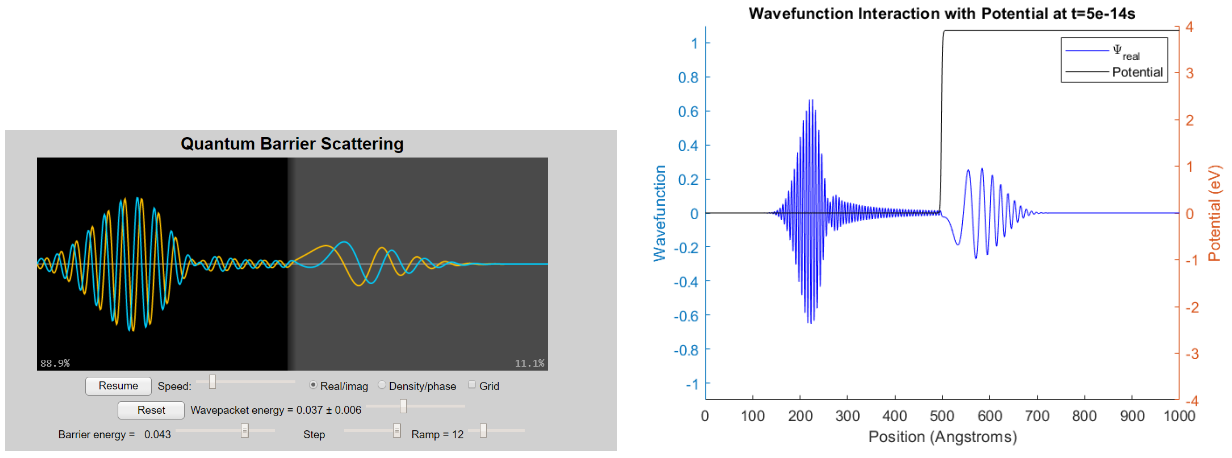


Figure 4: After 5e-15 seconds, it's clear that the wavepacket has bounced almost entirely off of the sharper boundary.

In addition, an online simulator shows that, given the right parameters for a step function potential, a quantum wave can develop this second hump given a non-zero "ramp" value, which allows one to change the slope of the rise to high potential. Looking more closely at the potential given in the assignment, it is possible that the curvature at the corners of the potential cause this second hump to appear naturally during reflection.



Wavepacket energy = 0.037 ± 0.006
Barrier energy = 0.043
Ramp = 12

Figure 5: Comparing an online simulator with the results for the real part of ψ matches this secondary bump in the wavefunction.[4]

This simulator could be extrapolated into more dimensions with more complicated formulas, but as it stands could be used to produce a more intuitive picture of quantum mechanical wave propagation for undergraduate classwork. Unbounded by the blackboard, simulations like those found on the internet can provide a much more valuable sense of physical intuition and reality.

To investigate the dynamics a bit further, the wavepacket is sent through a potential well of -30eV between 500 and 700 Angstroms. The following plots show the interaction of the packet with the new potential:

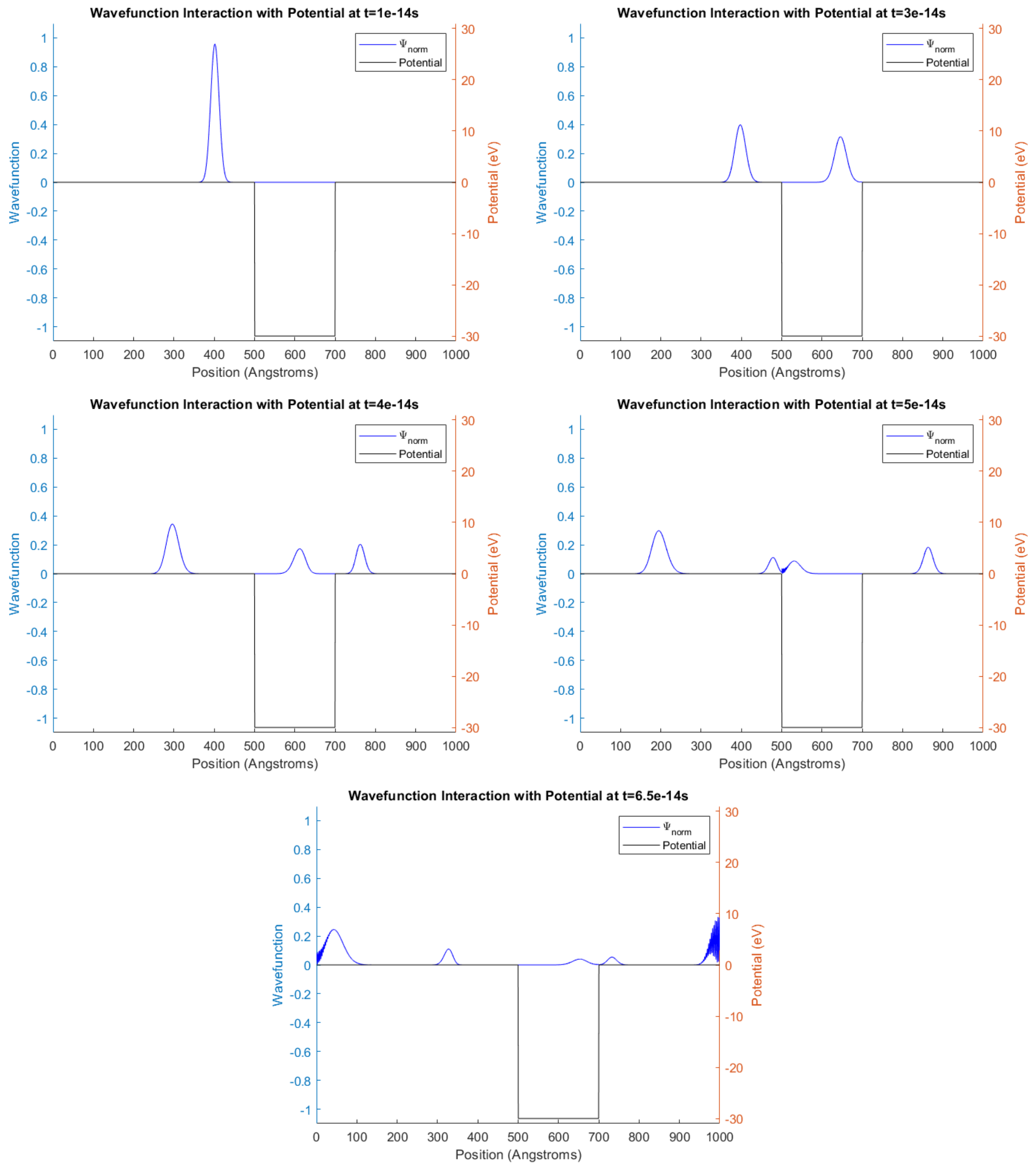


Figure 6: Interaction with a potential well at t=1e-14s, 3e-14s, 4e-14s, 5e-14s, 6.5e-14s.

These results are also quite interesting since there is a trapped piece of the pulse hitting alternate walls and sending pieces of itself through the barrier with each impact. This wave also dies away in magnitude as the energy is split between the spawned pulses.

References

[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes, The Art of Scientific Computing.* (3rd edit.), Cambridge Univ. Press, 2007, (ISBN 978-0-521-88068- 8, QA297 .N866 2007)

[2] Kirkland, Earl. *Time Dependent Schrödinger Equation.* AEP 4380 Fall 2019 Assignment 7. https://courses.cit.cornell.edu/aep4380/secure/hw07f19.pdf

[3] Kirkland, Earl. *Array Class Objects in C/C++ for Vectors and Matrices* AEP 4380 Fall 2019. https://courses.cit.cornell.edu/aep4380/secure/arrayt.hpp

[4] Schroeder, Daniel V. *Quantum Barrier Scattering* Physics Department of Weber State University. https://physics.weber.edu/schroeder/software/BarrierScattering.html

Source Code

```
/*  AEP 4380 Assignment #7
    Time Dependent Schrodinger Equation

    Run on Windows core i7 with gcc version 8.2.0 (MinGW.org GCC-8.2.0-3)

    Gregory Kaiser November 6th 2019

*/
#include <cstdlib>
#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>
#define ARRAYT_BOUNDS_CHECK
#include "arrayt.hpp"//from Prof. Kirkland’s https://courses.cit.cornell.edu/aep4380/secure/arrayt.hpp
#include <complex> //complex data type in C++
using namespace std; //makes writing code easier
typedef complex<double> Complex; //shorthand for declaring a complex variable
typedef arrayt<Complex> arrayc; //shorthand for declaring a complex array
int i,j;//loop variable
//----------physical constants-----------
double hbar = 6.5821e-16;//eVs
double hbar2_2m_elec = 3.801;//eV(A^2)

template <class T>
void ghk_tridiag(arrayt<T>& a, arrayt<T>& b, arrayt<T>& c, arrayt<T>& d,
                arrayt<T>& soln){//a version of tridiag
    int n =  a.n1();
    //step 1, do the special first row case of the process
    c(0) = c(0)/b(0);
    d(0) = d(0)/b(0);

    //step 2, replace all the c values with their value if a<--0 and b<--1 "ZIP"
    for(i=1; i<=n-2; i++){
        c(i) = c(i)/(b(i)-a(i)*c(i-1));
    }
    //step 2, replace all the d values with their value if a<--0 and b<--1 "ZIP"
    for(i=1; i<=n-1; i++){
        d(i) = (d(i)-a(i)*d(i-1))/(b(i)-a(i)*c(i-1));
    }
    //step 3, set last solution to d
    soln(n-1) = d(n-1);
    //backpropagate the solution "ZAP"
    //cout<<soln(300)<<endl;
    for(i=n-2; i>=0; i--){
        soln(i) = d(i)-c(i)*soln(i+1);
    }
}

int main(){

    ofstream fp;//output for psi real
    fp.precision(8);

    fp.open("hw7_1.dat");
    if(fp.fail()){
        cout<<"cannotopenfile"<<endl;
        return(EXIT_SUCCESS);
    }

    ofstream fp2;//output for potential function
    fp2.precision(9);

    fp2.open("hw7_2.dat");
    if(fp2.fail()){
        cout<<"cannotopenfile"<<endl;
        return(EXIT_SUCCESS);
    }

    ofstream fp3;//output for complex part
    fp3.precision(9);

    fp3.open("hw7_3.dat");
    if(fp3.fail()){
        cout<<"cannotopenfile"<<endl;
        return(EXIT_SUCCESS);
    }
    //HW Assignment
    //-----define bounds and spacing---
    double xmin=0, xmax=1000, dx=.1;//bounds and step size in position
    double tmin=0, tmax=5e-14, dt=1e-16;//bounds and step size in time
    double w = 2*dx*dx/dt;//omega definition for use in calculation later

    //calculate number of steps needed for while loop
    int Nx = (int) ((xmax-xmin)/dx) + 1; //number of points in x to calculate, rounds based on bounds
    int Nt = (int) ((tmax-tmin)/dt) + 1; //number of time steps to calculate, rounds based on bounds
    double t=tmin;

    //----problem initialization-----
    arrayc psi(Nx);//solution at a given time step
    arrayc Vtest(Nx); //a testing potential well, finite square well
    arrayc Vhw(Nx);//the homework potential well
    arrayc Vextra(Nx);//an extra potential

    //----define psi and V at t=0 as an initial condition
    double k_0 = 1;//Angstrom^-1 average wavenumber for gaussian starting shape
    double s = 20;//width of the gaussian starting shape
    double V_0 = 3.9;//eV height of potential well
    double w_v = 7;//Angstroms parameter of potential well

    //----intialize Vtest and Vhw and psi for all x---
    double curx; //just convenience
    for(i=0;i<Nx;i++){
        curx = i*dx;
        //testing potential
        if(curx>500){
            Vtest(i) = 20;
        }
        else{
            Vtest(i) = 0.0;
        }
        //extra potential
        if(curx>500&&curx<700){
            Vextra(i) = -30;
        }
        else{
            Vextra(i) = 0.0;
        }
        //hw potential
        Vhw(i) = Complex(V_0/(1+exp(0.5*(xmax-xmin)-curx)/w_v),0);
        //psi initialization
        psi(i) = Complex(exp(-((curx-0.3*(xmax-xmin))/s)*((curx-0.3*(xmax-xmin))/s))*cos(curx*k_0),
            exp(-((curx-0.3*(xmax-xmin))/s)*((curx-0.3*(xmax-xmin))/s))*sin(curx*k_0));
        fp<<psi(i).real()<<setw(20)<<psi(i).imag()<<setw(20)<<norm(psi(i))<<setw(15)<<
            curx<<setw(15)<<0<<endl;//output initial condition of psi
        fp2<<Vhw(i).real()<<setw(15)<<curx<<endl;//output the potential for display
        //fp2<<Vtest(i).real()<<setw(15)<<curx<<endl;//output the potential for display
        //fp2<<Vextra(i).real()<<setw(15)<<curx<<endl;//output the potential for display
    }

    //initialize the arrays that we will feed to tridiag
    arrayc a(Nx);
    arrayc b(Nx);
    arrayc c(Nx);
    arrayc d(Nx);
    //constants associated with the psi(x,t) term
    //(2mwi/hbar)-2-(2mdx^2/hbar^2)V(x)
    double complex_term = w/hbar2_2m_elec*hbar;
    double pot_factor = dx*dx/hbar2_2m_elec;
    double integral=0;
    Complex multiply;
    for(j=0;j<Nt;j++){
        t=j*dt;//constant time steps means we can do this more precise calculation of the time
        for(i=0;i<Nx;i++){
            //compute the value of a, b, c for every position of psi
            //with the Crank-Nicholson formulae
            a(i) = Complex(1,0);
            c(i) = Complex(1,0);

            //-------use the below for using the actual assigned hw potential
            b(i) = Complex(-2-pot_factor*Vhw(i).real(), complex_term);
            multiply = Complex(2+pot_factor*Vhw(i).real(), complex_term);

            //-----use the below for testing with the "perfect" step function
            // b(i) = Complex(-2-pot_factor*Vtest(i).real(), complex_term);
            // multiply = Complex(2+pot_factor*Vtest(i).real(), complex_term);

            //-----use the below for testing with the "perfect" step function
            // b(i) = Complex(-2-pot_factor*Vextra(i).real(), complex_term);
            // multiply = Complex(2+pot_factor*Vextra(i).real(), complex_term);

            if(i==0){//special case since psi(-1)=0 doesn't exist in the array but is a boundary
                d(i) = (multiply)*psi(i)-psi(i+1);
            }
            else if (i==Nx-1){//special case since psi(Nx)=0 doesn't exist but is a boundary
                d(i) = -psi(i-1)+(multiply)*psi(i);
            }
            else{
                d(i) = -psi(i-1)+(multiply)*psi(i)-psi(i+1);
            }
        }
        //at this point, all of a, b, c, d are calculated for a single time step
        //so feed into tridiag to get the next step of psi
        ghk_tridiag(a, b, c, d, psi);
        //then move to the next time step with this modified psi
    }
    for(i=0;i<Nx;i++){
        //print psi(i,Nt), the final value of psi at all positions
        curx=i*dx;
        fp3<<psi(i).real()<<setw(20)<<psi(i).imag()<<setw(20)<<norm(psi(i))<<
            setw(15)<<curx<<setw(15)<<t<<endl;
        integral+=norm(psi(i));
    }
    cout<<integral*dx<<endl;
    fp.close();
    fp2.close();
    fp3.close();
    return(EXIT_SUCCESS);
} //end main
```