# Finite Difference Method for Time-Dependent Schrödinger Equation

Kevin Juan (kj89)

October 31st 2018

## The Numerical Methods and Algorithms

The objective of this assignment was to implement a finite difference method on the Time-Dependent Schrödinger Equation (TDSE) for a wave packet through a potential field, $V(x,t)$. Schrödinger's Equation in 1D Cartesian coordinates is shown below:

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

The TDSE is an example of a parabolic partial differential equation, meaning it's first order in time and second order in space. A problem like this combines the Runge-Kutta method along with the finite difference method for boudnary value problems to obtain a solution. The particular method of interest for this assignment is the Crank-Nicholson Method, which averages the solutions given implicitly and explicitly. The global error for this method is $\mathcal{O}(\Delta t^2)$ globally, and the solution is always stable. The finite difference form of the TDSE is given below:

$$\begin{aligned}\psi(x-\Delta x, t+\Delta t) + \left[\frac{2mwi}{\hbar} - 2 - \frac{2m\Delta x^2}{\hbar^2}V(x)\right]\psi(x,t+\Delta t) + \psi(x+\Delta x, t+\Delta t) = \\ \psi(x-\Delta x,t) + \left[\frac{2mwi}{\hbar} + 2 + \frac{2m\Delta x^2}{\hbar^2}V(x)\right]\psi(x,t) + \psi(x+\Delta x,t)\end{aligned} \tag{2}$$

In the above equation, $w = \frac{2\Delta x^2}{\Delta t}$, where $\Delta x$ is the sampling size in space and $\Delta t$ is the sampling size in time. Equation 2 can be rewritten to have a form solvable by a matrix equation. The form is shown below:

$$a_j\psi(x_{j-1}, t_{n+1}) + b_j\psi(x_j, t_{n+1}) + c_j\psi(x_{j+1}, t_{n+1}) = d_j \tag{3}$$

This equation can be put into tri-diagonal matrix form as follows:

$$\begin{bmatrix} b_0 & c_0 & & & & & & \\ a_1 & b_1 & c_1 & & & & & \\ & & a_2 & b_2 & c_2 & & & \\ & & & & \ddots & & & \\ & & & & a_{N_x-2} & b_{N_x-2} & c_{N_x-2} & \\ & & & & & & a_{N_x-1} & b_{N_x-1} \end{bmatrix} \begin{bmatrix} \psi_0 \\ \psi_1 \\ \psi_2 \\ \vdots \\ \psi_{N_x-2} \\ \psi_{N_x-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{N_x-2} \\ d_{N_x-1} \end{bmatrix} \tag{4}$$

Elements not on the diagonals of the matrix are zero. The elements $\psi_i = \psi(x_i, t_{n+1})$ are the unkown future values to be found. Additionally, $\psi_{-1}$ and $\psi_{N_x}$ are fixed at zero. The following algorithm is used to solve the tri-diagonal matrix equation at each time step:

- Step 1: $c_0 \leftarrow c_0/b_0$ and $d_0 \leftarrow d_0/b_0$
- Step 2: $c_i \leftarrow c_i/(b_i - a_i c_{i-1})$ for $i = 1, 2, 3, ...N_x - 2$
- Step 3: $d_i \leftarrow (d_i - a_i d_{i-1})/(b_i - a_i c_{i-1})$ for $i = 1, 2, 3, ...N_x - 1$
- Step 4: $\psi_{N_x-1} \leftarrow d_{N_x-1}$
- Step 5: $\psi_i \leftarrow d_i - c_i \psi_{i+1}$ for $i = N_x - 2, ..., 0$

Steps 1-2 convert the matrix into upper diagonal form while Steps 4 and 5 perform back substitution to find the new $\psi$.

More on explicit, implicit, and Crank-Nicholson methods can be found in §20.2 in Press et al[1]. Additional information on tri-diagonal matrices can be found in §2.4 in Press et al[1].

# Electron Wave Packet Through Potential Barrier

When a wave encounters a potential barrier, it can exhibit a phenomena known as quantum tunneling. Classical mechanics predicts that a particle with energy lower than the potential barrier will be forbidden from passing through. However, in quantum mechanics, waves have the probability of passing through the barrier. Similarly the entire wave, or a portion of it, can be reflected by the barrier. The TDSE described by Equation 1 gives the wave function $\psi$ for a particle of mass $m$ moving through a potential field $(x, t)$. The value $\hbar = 6.5821 \times 10^{-16}$ eV-sec is Planck's constant, and $\frac{\hbar^2}{2m} = 3.801$ eV-Å$^2$ for an electron. The potential field that will be used in this assignment is described by the equation below:

$$V(x) = V_1 \exp\left[ -\left(\frac{x - x_1}{w_1}\right)^2 \right] + V_2 \exp\left[ -\left(\frac{x - x_2}{w_2}\right)^2 \right] + V_3 \exp\left[ -\left(\frac{x - x_3}{w_3}\right)^2 \right] \tag{5}$$

$V_1 = 3.8$eV, $V_2 = 4.0$eV, $V_3 = 4.2$eV, $x_1 = 0.5L$, $x_2 = 0.55L$, $x_3 = 0.6L$ and

$w_1 = w_2 = w_3 = 10$ Å. The calculation will be done over a region $0 < x < L$, where $L = 500$ Å. The wave function at $t = 0$ is given by the following equation:

$$\psi(x, t = 0) = \exp\left[ -\left(\frac{x - 0.3L}{s}\right)^2 + ixk_0 \right] \tag{6}$$

The width $s = 15$ Å and the average wavenumber $k_0 = 1$ Å$^{-1}$.

# Implementation and Results

Because the wavefunctions have real and imaginary parts, the STL complex class template had to be used. This allows for complex valued arithmetic to be done easily. Using the complex data type along with array class `arrayt`, a complex type array was created for arrays $a, b, c, d$, and $\psi$.

A template called `tridiag` was written to solve the tridiagonal matrix equation for every time step. The template takes in the five complex array and performs the algorithm listed above to obtain updated $\psi$ values.

The potential field was plotted as shown below:

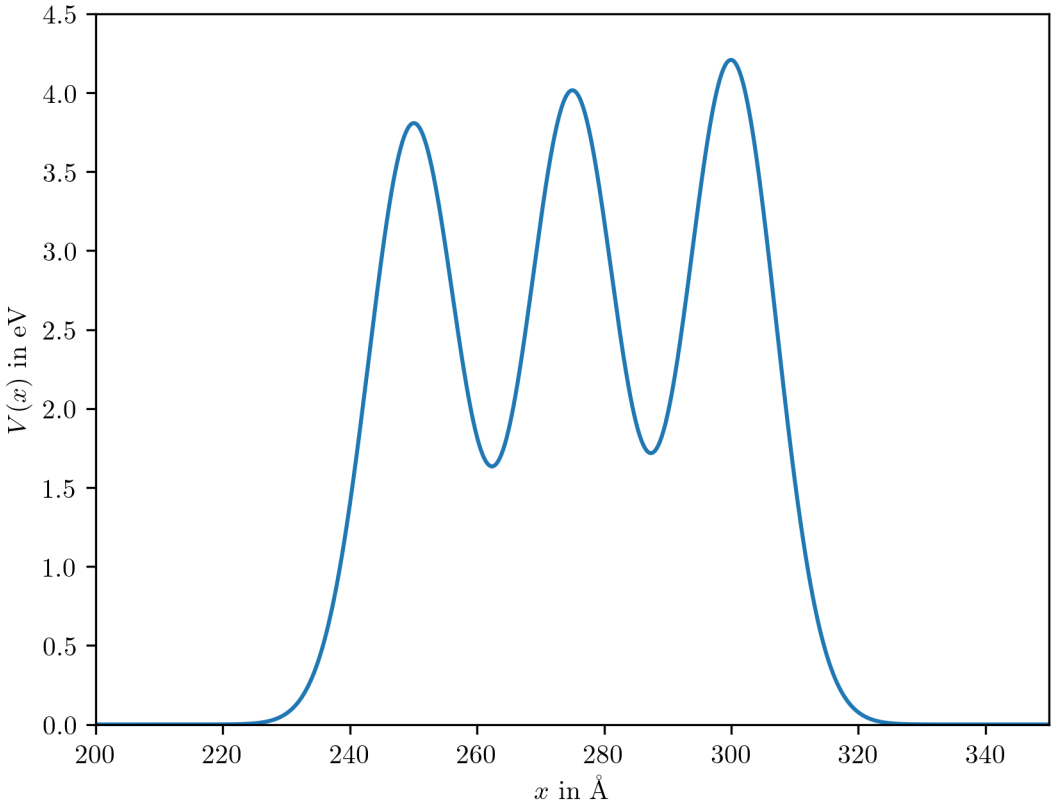


Figure 1: Potential field used in the simulation.

The first part of the assignment was to find an adequate sampling size for space and time. As shown below, $\Delta x$ was reduced until $|\psi|^2$ curves were identical.

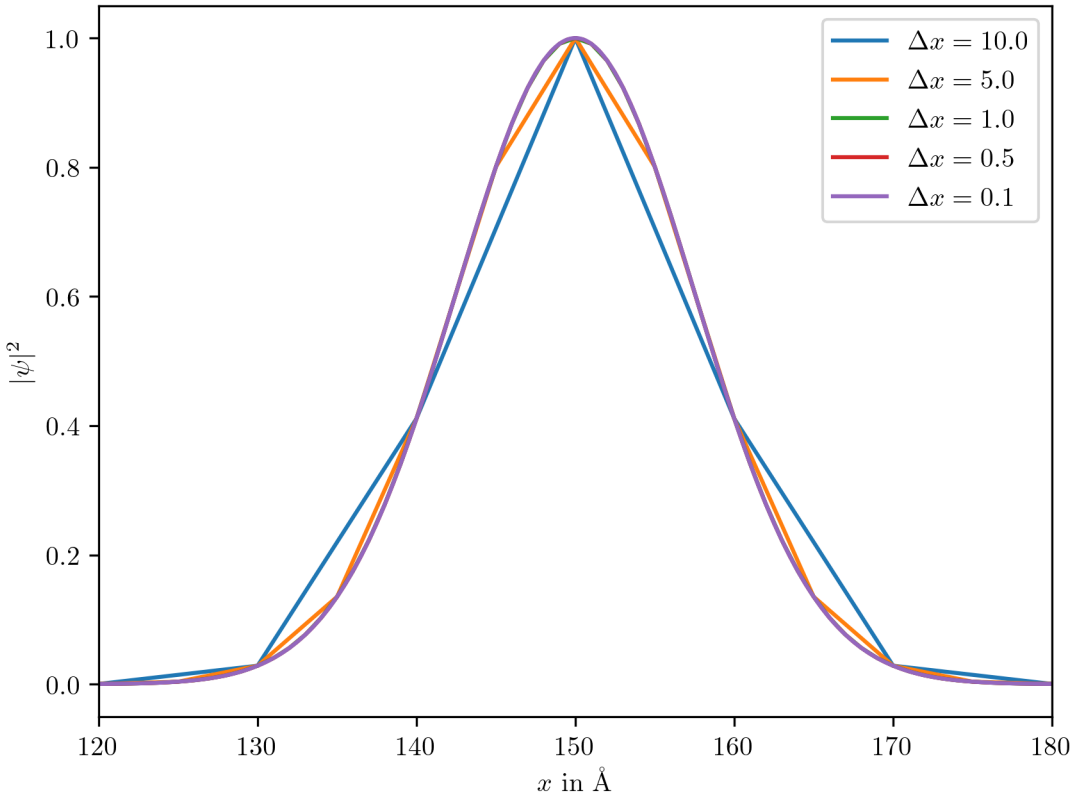


Figure 2: $|\psi|^2$ for different values of $\Delta x$ to find an adequate sample size for space.

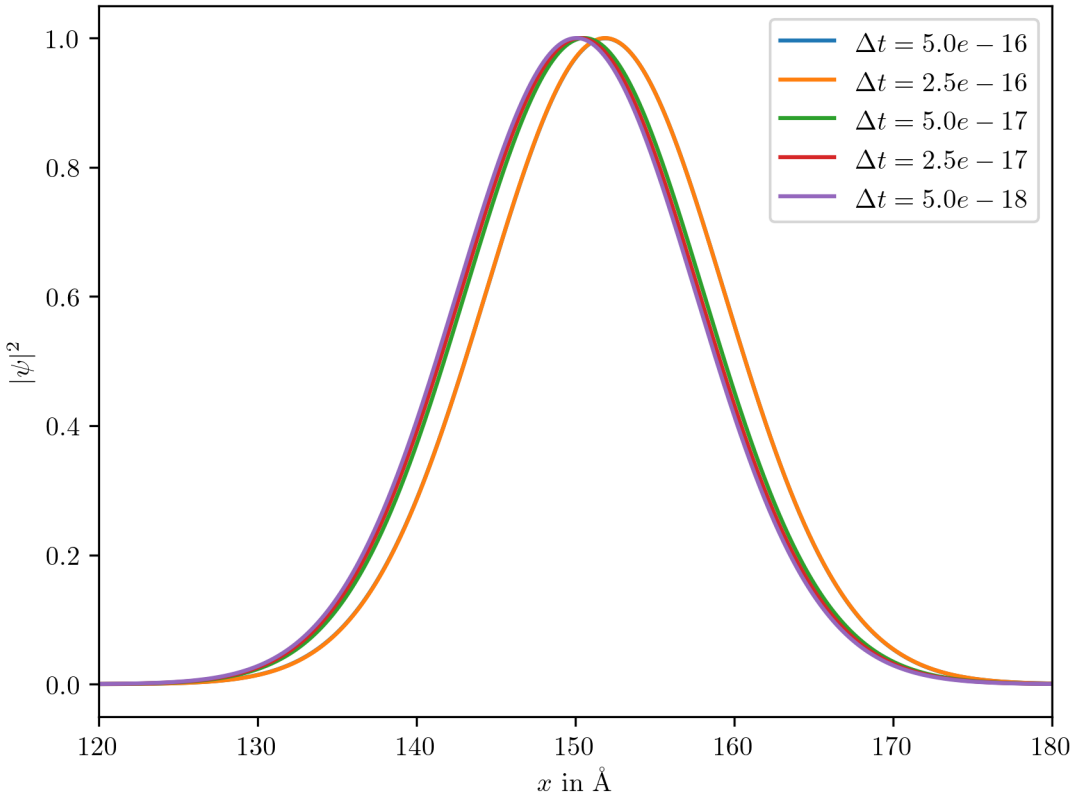


Figure 3: $|\psi|^2$ for different $\Delta t$ to find an adequate sample size for time.

An adequate sample size in space found by Figure 2 was $\Delta x = 0.1$ Å. This value gave smooth $|\psi|^2$ and $\psi$ curves. Figure 3 shows that an adequate sample size for time is $\Delta t = 2.5 \times 10^{-17}$s. The wavefunction at $t = 0$ was also plotted as shown below using $\Delta x = 0.1$:

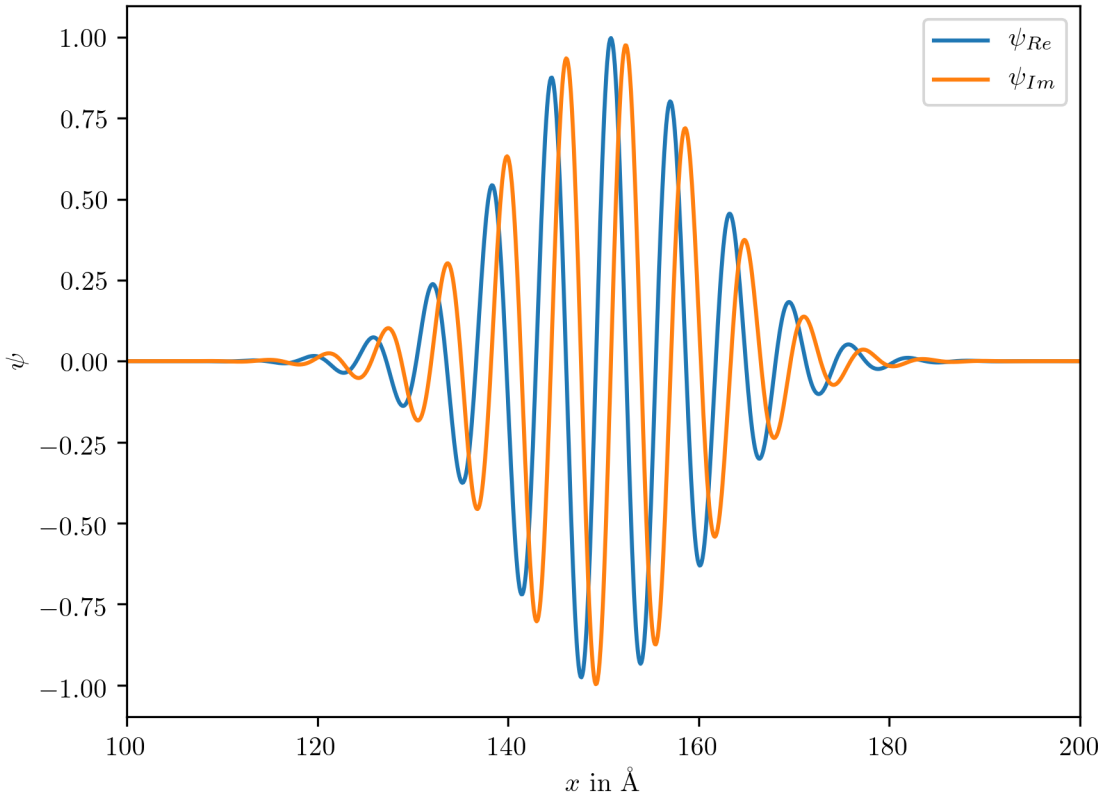


Figure 4: $\psi(x,t)$ vs. $x$, showing both the imaginary and real parts.

The wave propagation was tracked at five different times as shown below:

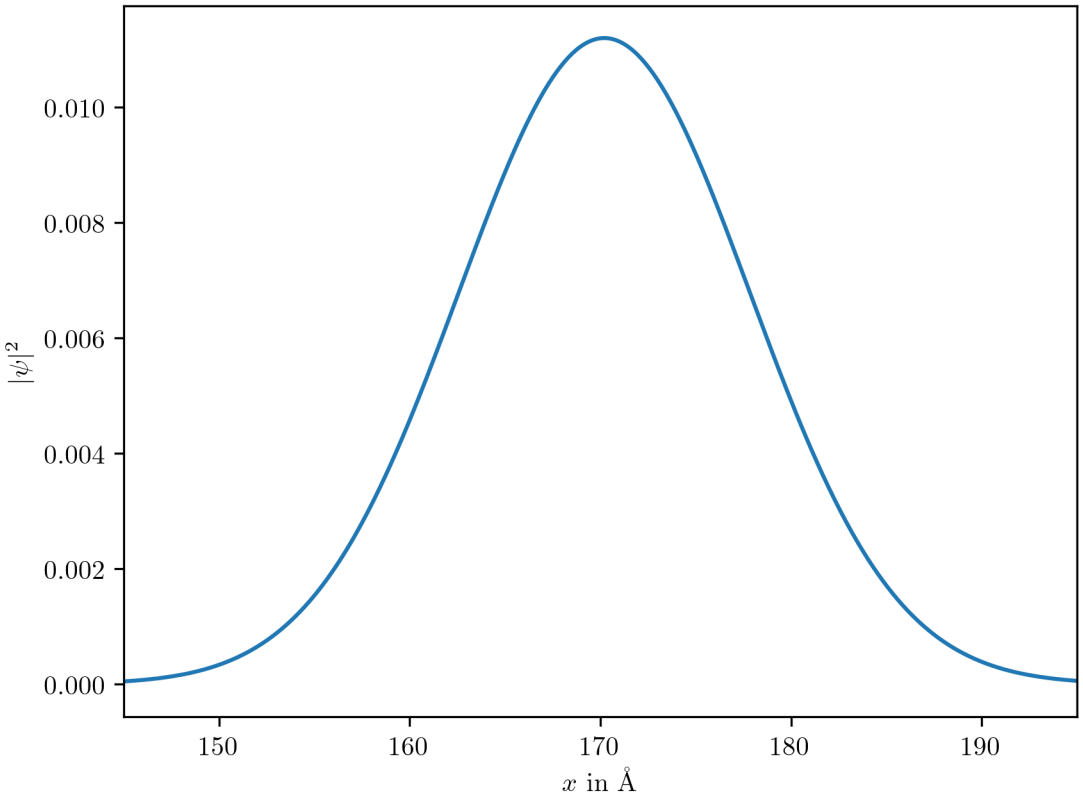


Figure 5: $|\psi|^2$ vs. $x$ at $t = 0.5 \times 10^{-14}$s.

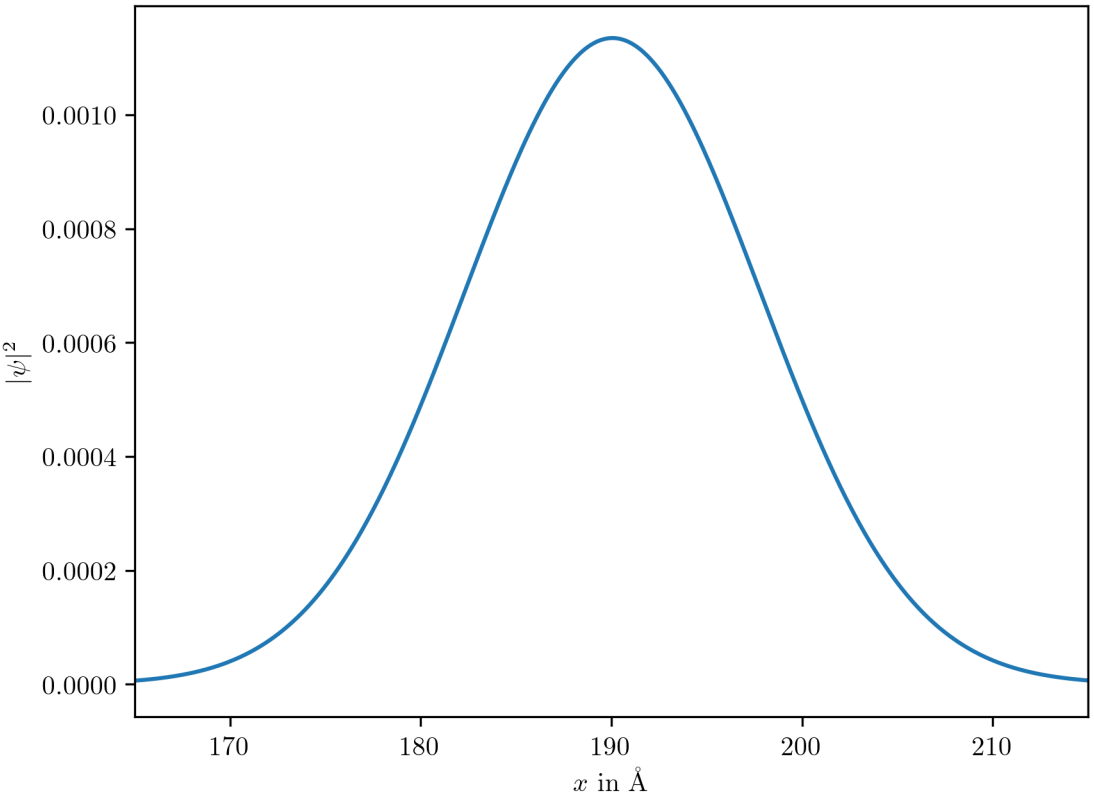


Figure 6: $|\psi|^2$ vs. $x$ at $t = 1.0 \times 10^{-14}$s.

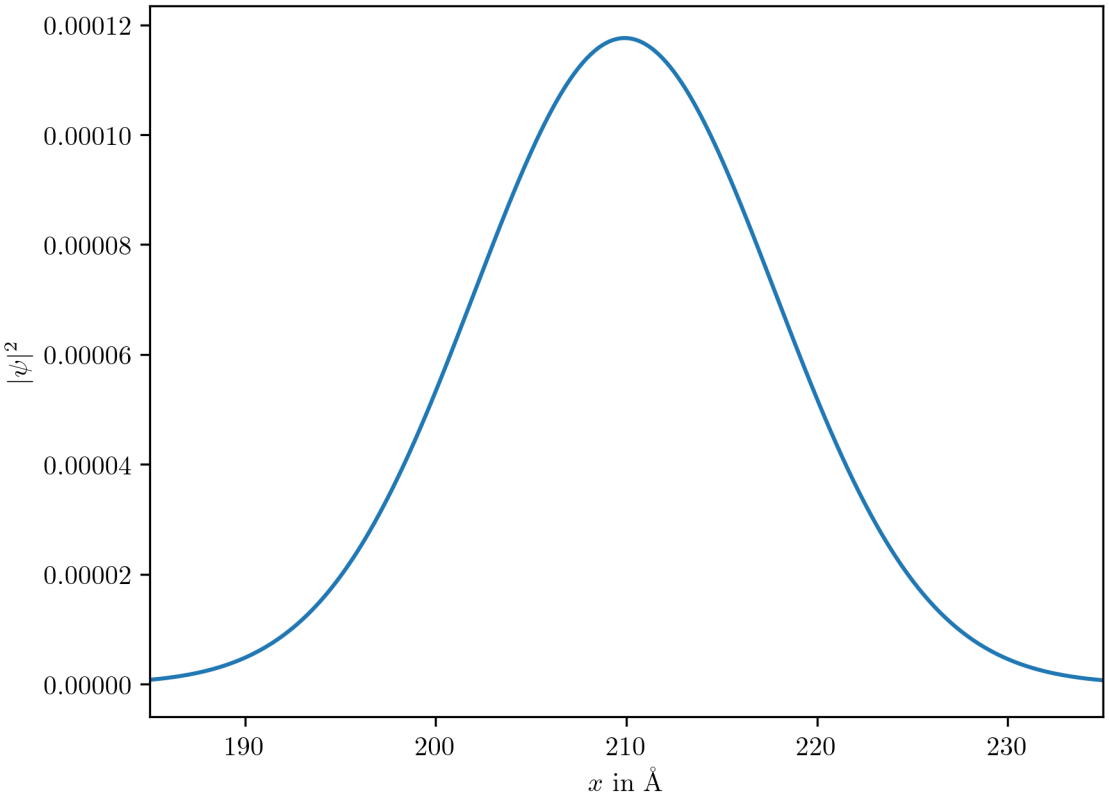


Figure 7: $|\psi|^2$ vs. $x$ at $t = 1.5 \times 10^{-14}$s.

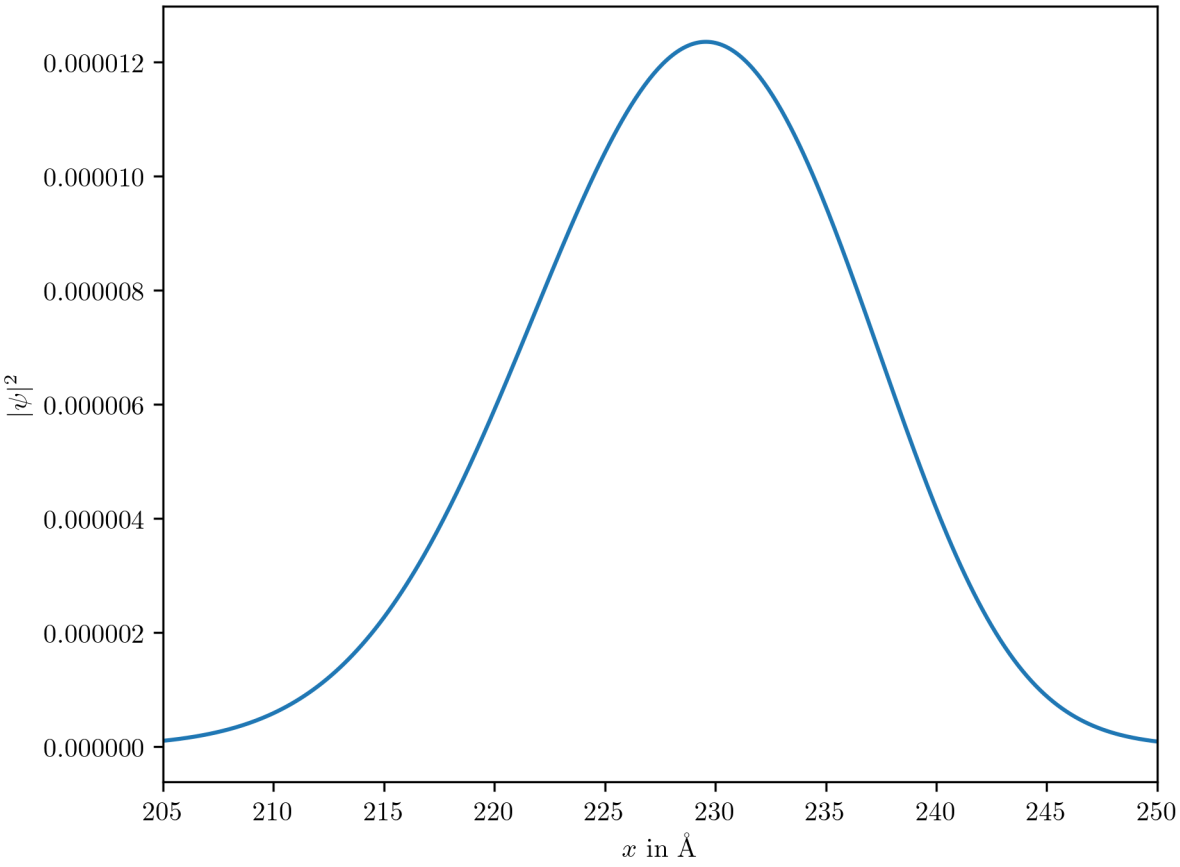


Figure 8: $|\psi|^2$ vs. x at $t = 2.0 \times 10^{-14}$s.

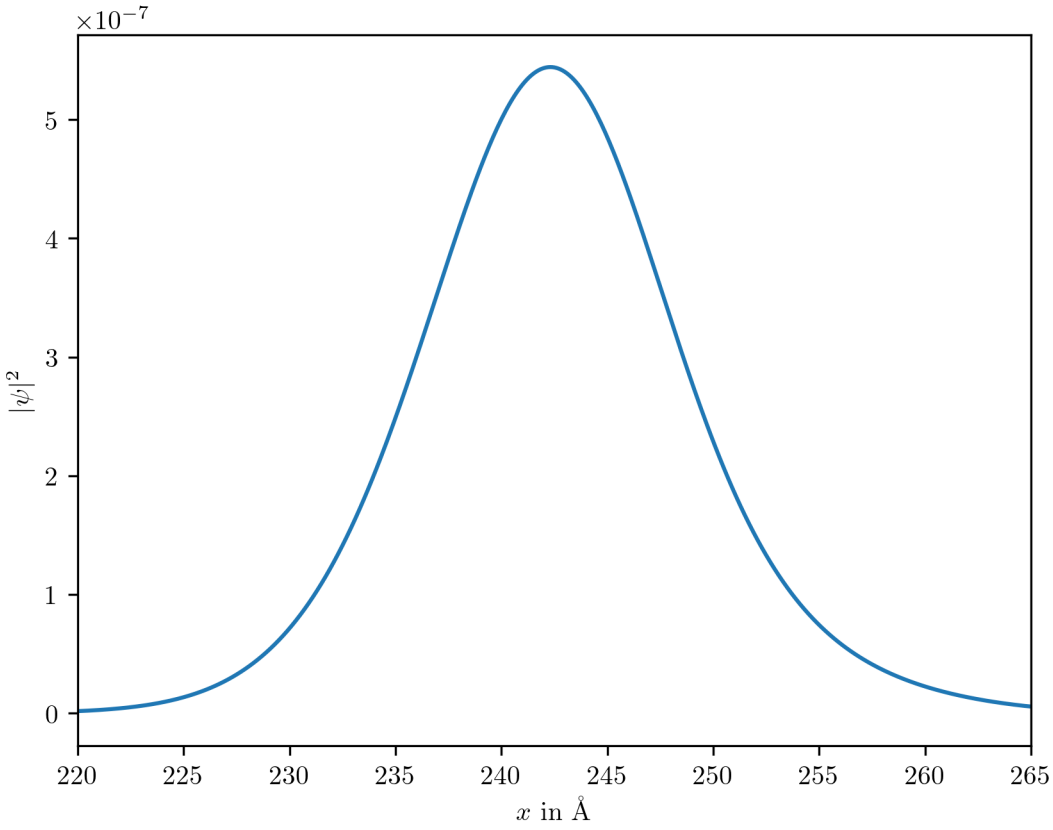


Figure 9: $|\psi|^2$ vs. $x$ at $t = 2.5 \times 10^{-14}$s.

# Source Code

```
/* AEP 4380 Homework #6

Crank-Nicholson method is used
Tested on time-dependent Schrodinger Equation

Run on a core i7 using clang 1000.11.45.2 on macOS Mojave

Kevin Juan 31 October 2018
*/

#include <cstdlib>  // plain C
#include <cmath>    // use math package
#include <iostream> // stream IO
```

```
#include <fstream>  // stream file IO
#include <iomanip>  // to format the output
#define ARRAYT_BOUNDS_CHECK
#include "arrayt.hpp" // to use arrays
#include <complex>

using namespace std;

typedef complex<double> I;
typedef arrayt<I> arrayc;

template <class T>
void tridiag(arrayc &a, arrayc &b, arrayc &c, arrayc &psi, arrayc &d)
{
    int j, jMax = b.n1();
    I bac;

    c(0) = c(0) / b(0);
    d(0) = d(0) / b(0);
    for (j = 1; j < jMax - 1; j++)
    {
        bac = b(j) - a(j) * c(j - 1);
        if (j < jMax - 2)
        {
            c(j) = c(j) / bac;
        }
        d(j) = (d(j) - a(j) * d(j - 1)) / bac;
    }

    psi(jMax - 1) = d(jMax - 1);
    for (j = jMax - 2; j >= 0; j--)
    {
        psi(j) = d(j) - c(j) * psi(j + 1);
    }
}

double Vx(double x, double x1, double x2, double x3, double w1, double w2,
double w3, double V1, double V2, double V3)
{
    double x1Sq = (x - x1) * (x - x1) / (w1 * w1);
    double x2Sq = (x - x2) * (x - x2) / (w2 * w2);
    double x3Sq = (x - x3) * (x - x3) / (w3 * w3);
    double Vx = V1 * exp(-x1Sq) + V2 * exp(-x2Sq) + V3 * exp(-x3Sq);

    return Vx;
```

```
}

I psiInit(double x, double L, double s, double k0)
{
    I psiInit = exp(I(-(x - 0.3 * L) * (x - 0.3 * L) / (s * s), x * k0));
    return psiInit;
}

int main()
{
    const static double hbar = 6.5821e-16, hBarSq2m = 3.801;
    const static double V1 = 3.8, V2 = 4.0, V3 = 4.2;
    const static double L = 500.0, x1 = 0.5 * L, x2 = 0.55 * L;
    const static double x3 = 0.6 * L, w1 = 10.0, w2 = 10.0, w3 = 10.0;
    const static double s = 15.0, k0 = 1.0, t = 2.5e-14;
    int jMax = 5001, nMax = 1001, j, n;
    double xDelta = L / (double)(jMax - 1), tDelta = t / (double)(nMax - 1);
    double poten, w = 2.0 * xDelta * xDelta / tDelta;

    arrayc a(jMax);
    arrayc b(jMax);
    arrayc c(jMax);
    arrayc d(jMax);
    arrayc psi(jMax);

    // output data for x
    ofstream fpx;
    fpx.open("x_vals.dat");
    if (fpx.fail())
    {
        cout << "cannot open file" << endl;
        return (EXIT_SUCCESS);
    }
    for (j = 0; j < jMax; j++)
    {
        fpx << xDelta * j << endl;
    }
    fpx.close();

    // output data for t
    ofstream fpt;
    fpt.open("t_vals.dat");
    if (fpt.fail())
    {
        cout << "cannot open file" << endl;
```

```
    return (EXIT_SUCCESS);
}
for (n = 0; n < nMax; n++)
{
    fpt << tDelta * n << endl;
}
fpt.close();

// output data for potential
ofstream fpPoten;
fpPoten.open("poten_vals.dat");
if (fpPoten.fail())
{
    cout << "cannot open file" << endl;
    return (EXIT_SUCCESS);
}
for (j = 0; j < jMax; j++)
{
    poten = Vx(j * xDelta, x1, x2, x3, w1, w2, w3, V1, V2, V3);
    fpPoten << poten << endl;
}
fpPoten.close();

// initialize a, b, c, and d at t = 0
for (j = 0; j < jMax - 1; j++)
{
    poten = Vx(j * xDelta, x1, x2, x3, w1, w2, w3, V1, V2, V3);
    a(j) = 1.0;
    b(j) = I(-2.0 - xDelta * xDelta * poten / hBarSq2m, hbar * w
    / hBarSq2m);
    c(j) = 1.0;
    d(j) = -psiInit((j - 1) * xDelta, L, s, k0) + I(2.0 + xDelta *
    xDelta * poten / hBarSq2m, hbar * w / hBarSq2m) * psiInit(j *
    xDelta, L, s, k0) - psiInit((1 + j) * xDelta, L, s, k0);
    psi(j) = 0.0;
}

// output data for t = 0
ofstream fpPsiReInit;
fpPsiReInit.open("PsiRe_t0_vals.dat");
if (fpPsiReInit.fail())
{
    cout << "cannot open file" << endl;
    return (EXIT_SUCCESS);
}
```

```
for (j = 0; j < jMax; j++)
{
    fpPsiReInit << psiInit(j * xDelta, L, s, k0).real() << endl;
}
fpPsiReInit.close();

ofstream fpPsiImInit;
fpPsiImInit.open("PsiIm_t0_vals.dat");
if (fpPsiImInit.fail())
{
    cout << "cannot open file" << endl;
    return (EXIT_SUCCESS);
}
for (j = 0; j < jMax; j++)
{
    fpPsiImInit << psiInit(j * xDelta, L, s, k0).imag() << endl;
}
fpPsiImInit.close();

ofstream fpPsiSqInit;
fpPsiSqInit.open("PsiSq_t0_vals.dat");
if (fpPsiSqInit.fail())
{
    cout << "cannot open file" << endl;
    return (EXIT_SUCCESS);
}
for (j = 0; j < jMax; j++)
{
    fpPsiSqInit << (psiInit(j * xDelta, L, s, k0) * conj(psiInit(j
    * xDelta, L, s, k0))).real() << endl;
}
fpPsiSqInit.close();

// output data for different times
ofstream fpPsiSqVals;
fpPsiSqVals.open("PsiSq_vals.dat");
if (fpPsiSqVals.fail())
{
    cout << "cannot open file" << endl;
    return (EXIT_SUCCESS);
}
for (n = 0; n < nMax; n++)
{
    tridiag<arrayc>(a, b, c, psi, d);
    for (j = 0; j < jMax; j++)
```

```
        {
            fpPsiSqVals << setw(15) << (psi(j) * conj(psi(j))).real();
        }
        fpPsiSqVals << endl;
    }
    fpPsiSqVals.close();
}
```

# References


[1] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, editors. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, Cambridge, UK ; New York, 3rd ed edition, 2007. OCLC: ocn123285342.