Time Dependent Schrödinger Equation

HW 7: Wednesday, Oct. 30, 2019
DUE: Wednesday, Nov. 6, 2019
READ: *Numerical Recipes in C++*, Section 20.2 on explicit, implicit and Crank-Nicholson
Section 2.4 on the solution of a tri-diagonal matrix
you may also want to read section 2.0-2.3 on matrix algebra
OPTIONAL: Landau, Paez, and Bordeianu: section 17.17,18,19, 18.5,6,7
has a little on explicit and implicit methods

OPTIONAL: Gerald and Wheatley, Applied Numerical Analysis, section 8.2, page 481-498, has a much better treatment of the Crank Nicholson method than Numerical Recipes.

OPTIONAL: Ammeraal: section 7.5 p. 243 on complex class template.

For your information the homework mean and standard deviation are 9.4 and 0.36 for homework number 2 through 5 (which is very good) excluding a few late ones.

PROBLEM (10 points):

The time dependent Schrödinger equation for the wave function ψ of a particle of mass m moving in a potential energy $V(x,t)$ is:

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

where $\hbar = 6.5821 \times 10^{-16}$ eV-sec is Planck's constant, and $\frac{\hbar^2}{2m}$=3.801 eV-A^2 for an electron. This homework will calculate the time dependent propagation of an electron wave packet through a potential barrier. The calculation should be performed in a region of length L=1000 Angstroms, $(0 < x < L)$. Start with an initial (complex valued) Gaussian wave function (for an electron) of:

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

with a width of s=20 Angstroms and average wavenumber k_0=1 Angstrom^{-1}. The potential energy $V(x)$ will be a boundary with a non-zero width:

$$V(x) = \frac{V_0}{1+\exp[(0.5L-x)/w_v]} \tag{3}$$

where V_0=3.90eV and w_v=7 Angstroms.

A finite difference form of the Schrödinger equation for use in the Crank-Nicholson method is:

$$\begin{aligned}\psi(x-\Delta x, t+\Delta t) + \left[\frac{2mwi}{\hbar} - 2 - \frac{2m\Delta x^2}{\hbar^2}V(x)\right]\psi(x,t+\Delta t) + \psi(x+\Delta x, t+\Delta t) \\ = -\psi(x-\Delta x,t) + \left[\frac{2mwi}{\hbar} + 2 + \frac{2m\Delta x^2}{\hbar^2}V(x)\right]\psi(x,t) - \psi(x+\Delta x,t)\end{aligned} \tag{4}$$

where $w = 2\Delta x^2/\Delta t$, Δx is the sampling size in space and Δt is the sampling size in time. All of the values at the old time are on the right hand side and the unknown values at the new time are on the left-hand side. This equation has the form:

$$a_j\psi(x_{j-1}, t_{n+1}) + b_j\psi(x_j, t_{n+1}) + c_j\psi(x_{j+1}, t_{n+1}) = d_j \tag{5}$$

which can be written as a tri-diagonal matrix equation:

$$\begin{bmatrix} b_0 & c_0 & & & & \\ a_1 & b_1 & c_1 & & & 0 \\ & a_2 & b_2 & c_2 & & \\ & & & \ddots & & \\ & 0 & & a_{N_x-2} & b_{N_x-2} & c_{N_x-2} \\ & & & & a_{N_x-1} & b_{N_x-1} \end{bmatrix} \begin{bmatrix} \psi_0 \\ \psi_1 \\ \psi_2 \\ \vdots \\ \psi_{N_x-2} \\ \psi_{N_x-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{N_x-2} \\ d_{N_x-1} \end{bmatrix} \tag{6}$$

where $\psi_i = \psi(x_i, t_{n+1})$ are the unknowns to be found. The two end points ψ_{-1} and ψ_{N_x} are fixed at zero. This tri-diagonal system may be solved (at each time step) with the following algorithm:

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 $i = 1, 2, 3, \cdots (N_x - 2)$ (in order)
$d_i \leftarrow (d_i - a_i d_{i-1})/(b_i - a_i c_{i-1})$ for $i = 1, 2, 3, \cdots (N_x - 1)$ (in order)

step 3: $\psi_{N_x-1} \leftarrow d_{N_x-1}$
$\psi_i \leftarrow d_i - c_i \psi_{i+1}$ for $i = (N_x - 2), \cdots, 0$

Steps 1 and 2 convert the matrix to upper diagonal form and Step 3 performs back substitution to solve for the unknowns. Note that you only need to store the diagonal elements and not the whole matrix. Complex arithmetic has to be written out the long way (i.e. separate real and imaginary parts) in the original standard C but can be done directly in C++ (see the HINT below). You should write a subroutine similar to tridiag() in section 2.4 of Numerical Recipes[2], but converted to use complex valued data. You might also want to consider using a template for tridiag().

Plot the potential, the real and imaginary part of ψ and $|\psi|^2$ at t=0. Next solve for the propagation of this wave packet as a function of time using the Crank-Nicholson method. Plot $|\psi|^2$ at times t=1e-14, 2e-14, 3e-14, 4e-14, and 5e-14 sec. You should remember to verify that you have adequate sampling in time and space. The sampling in space Δx should be much less than the smallest wavelength (by something like a factor of 10) and then guess at an initial Δt and decrease it until you get a consistent result (can be determined by plotting successive curves until they line up).

You may check that the integral of $|\psi|^2$ remains constant from start to finish to test your program (useful but not conclusive). This is not a bound state so this integral does not have to be unity (1).

OPTIONAL: Try different heights and/or widths of the potential (or different shapes of the potential).

References

[1] A. Goldberg, H. M. Schey and J. L. Schwartz, Amer. Journal of Physics 35 (1967) p. 177.

[2] W. H. Press, S. A, Teukolsky, W. T. Vetterling and B. P. Flannery *Numerical Recipes, The Art of Scientific Computing, 3rd edit.*, Camb. Univ. Press 2007.

HINT: The original standard C did not have a complex data type. To use the original C for this homework exercise, you would have to write out the complex arithmetic the long way using two variables or arrays (one for the real component and one for the imaginary component). The Standard Template Library (STL) has been added to the C++ language and this contains a complex class template. The complex class template can be used like a complex data type with an underlying data type of float, double or long double. The STL is not documented very well (if at all) in most of the standard books on C++. The attached sample program (below) illustrates how to use the complex template set to a type double (replace the expression in <> with "float" or "long double" to use the other available types). The STL can make your program somewhat more elegant and easier to write but as of now the code it produces may run significantly slower (about a factor of two on some compilers) than writing it out in standard C with two real arrays (varies with compiler). It is fine for short program like this one but be careful using it on computationally intensive programs. You may find the STL useful for this homework exercise.

The new version of C defined in the 1999 standard (called C99) includes a native complex data type (which is about the only thing in C that is not in C++) which should compile into fast code, but not all current compilers implement this very well (there may be some links on course web site for some information on the C99 library standards and the complex data type). The status of C99 in gcc is listed at
`http://http://gcc.gnu.org/c99status.html` (which indicates the the native complex type is working). For now its probably best to stick with the STL version of complex (which works, but is a little slow). MS visual studio does not yet support the native complex data type.

Example of Using the Complex Data Type in the STL

```
//          *** complex_stl.cpp ***
//
//  quick demonstration of complex template using
//  the C++ Standard Template Library
//
//  reference:  C. Hughs and T. Hughes, "Mastering the
//          Standard C++ Classes", Wiley 1999, p. 444-448
//      Rob McGregor, Pratical C++, Que 1999, chap. 31
//
//  this has been tested with g++ 8.1.0 64 bit
//
//  21-oct-2019 E. Kirkland

#include <cstdlib>
#include <iostream>     // C++ I/O operations

using namespace std;

// define the following symbol to enable bounds checking
//#define ARRAYT_BOUNDS_CHECK
#include "arrayt.hpp"

// include next two lines for complex data class
// note: complex<> supports float, double, and long double
#include <complex>      // new C++ standard library

// shorthand data types to save some typing
typedef complex<double> CMPLX;  // method 1
```

```
typedef arrayt<CMPLX> arrayc;

//-----------------------------------------------------
//   dummy function to multiply all elements
//   of complex array tmp() by s=(0,1)=sqrt(-1)
//
void scale( arrayc& tmp, int n )
{
    int i;
    const CMPLX s( 0.0, 1.0);   // complex constant
    for( i=0; i<n; i++) tmp(i) = tmp(i) * s;
    return;
}
//-----------------------------------------------------
int main()
{
    int i;
    const int N=10;
    CMPLX x, y;     // complex variables
    arrayc a(N);    // an array of complex valued numbers

    //------- first test simple arithmetic --------------------
    // set x real part to 1.0 and x imag part to 2.0
    x = CMPLX( 1.0, 2.0 );
    y = CMPLX( 3.0, 4.0 );

    x = x + y;      // complex arithmetic (+ - * and / allowed)
    cout << "x = " << x << endl;    // C++ output

    x =  1.0;       // initialize to a real value
    y = CMPLX( 1.0, 1.0 );
    x = x / y;
    cout << "norm(x) = " << norm(x) << endl; // norm() = square mag.

    // -------- next test complex arrays ----------------------
    // initialize complex values in array
    for( i=0; i<N; i++ ) a(i) = CMPLX(1.0,((double)i));

    scale( a, N );          // test subroutine call with complex array

    for(i=0; i<N; i++)      // print values of new array
        cout << i<< " real= " <<  a(i).real() << ", imag= " << a(i).imag()
                     <<", complex= " << a(i) << endl;

    return( EXIT_SUCCESS );
}  // end main()
```

Example of Using the Native Complex Data Type

```
/*  ---- complexc99.c

  quick test of using complex numbers in C99
  only works in plain C NOT C++ sadly

*/
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main()
{
    double complex a = 32.123 + 24.456*I;
    double complex b = 23.789 + 42.987*I;
    double complex c = 3.0 + 2.0*I;

    double complex sum = a + b;
    double complex pwr = cpow( a, c );

    printf("a is %f + %fi\n", creal(a), cimag(a) );
    printf("b is %f + %fi\n", creal(b), cimag(b) );

    printf("a+b is %f + %fi\n", creal(sum), cimag(sum) );
    printf("a-b is %f + %fi\n", creal(a-b), cimag(a-b) );
    printf("a*b is %f + %fi\n", creal(a*b), cimag(a*b) );
    printf("a/b is %f + %fi\n", creal(a/b), cimag(a/b) );
    printf("a^b is %f + %fi\n", creal(pwr), cimag(pwr) );

    return( 0 );

}    /* end main() */
```