Visualizing the Quantum World

Cris Cecka
Harvey Mudd College
Physics 170 - Computational Physics
28 April 2005

Abstract

In this paper, I investigate methods of visualizing solutions to the One-Dimensional Time-Dependent Schrödinger Equation. Last year, I researched and implemented a numerical method which integrates the Schrödinger equation directly in order to evolve a state in time. Although successful, it has severe speed limitations and suffers from many common problems of a numerical integrator including propagation of error over time resulting in divergences from conservation laws. In this paper, we'll derive the previous method, show some of the problems with it, and propose an improved algorithm to solves the time-independent Schrödinger equation for an arbitrary potential, then evolves these eigenstates in time.

Introduction

At the beginning of the twentieth century, experimental evidence suggested that atomic particles were also wave-like in nature. For example, electrons were found to give diffraction patterns when passed through a double slit in a similar way to light waves. Therefore, it was reasonable to assume that a wave equation could explain the behavior of atomic particles.

Schrödinger was the first to write down such a wave equation. Many physicists spent years interpreting the Schrödinger equation, finding new ways to visualize particles, and generally attempting to reformulate the classical image of physics. An extreme difficulty is that the Schrödinger equation includes a function called the Potential Energy Function $V(x)$, which changes based on the system you are interested in. Thus, the Schrödinger equation cannot have a stand alone solution and many systems with rather simple potentials also do not have closed form solutions. Numerical integration methods must then be used to find an approximation of the solution. Many computational physicists have tackled this problem to produce movies and examples of certain educationally interesting examples in order to provide a medium in which the effects of quantum mechanics can literally be seen. As of last year when this project started, I could not find a single real-time integrator though. Since then, they seem to have been popping up everywhere. One of my favorite is Paul Falstad. In early 2005, he released an applet on his website which accomplished what I had wanted to try since I finished my first version a year earlier. His applet peaked my interest since it ran incredibly fast and smoothly. All the methods I had previously seen for finding eigenstates of a potential were iterative and slow, I knew I must be missing something. Unfortunately, at the time I took up this project, he had not released his source. Thus, this project extends my previous work on creating a real-time tool to visualize some of the implications of quantum mechanics by attempting to reproduce some of Paul Falstad's results.

Derivation of Time-Dependent Numerical Method

Feynman, in many of his computational methods, assumed all constants to be one for simplicity of calculations. In the numerical approach here, we will do the same. We define $\hbar = 1, m_e = 1, q_e = -1$. The time dependent Schrödinger Equation in one dimension then becomes

$$\imath\frac{\partial\Psi(x,t)}{\partial t} = -\frac{1}{2}\frac{\partial^2\Psi(x,t)}{\partial x^2} + V(x)\Psi(x,t) \tag{1}$$

To integrate this numerically, we must of course work in a discrete world. Thus, $\Psi(x,t)$ must be reduced to a domain of equally spaced points: $x_j = x_0 + jdx$. At these points define $\Psi(x_j,t) = \Psi_j(t)$ and $V(x_j) = V_j$. The Schrödinger Equation then becomes

$$\imath\frac{\partial\Psi_j(t)}{\partial t} = -\frac{1}{2}\frac{\partial^2}{\partial x^2}\Psi_j(t) + V_j\Psi_j(t) \tag{2}$$

where we can use the standard approximation $\frac{\partial^2}{\partial x^2}\Psi_j(t) \simeq \frac{\Psi_{j+1}(t)-2\Psi_j(t)+\Psi_{j-1}(t)}{\partial x^2}$ to yield:

$$\imath\frac{\partial\Psi_j(t)}{\partial t} = -\frac{\Psi_{j+1}(t) - 2\Psi_j(t) + \Psi_{j-1}(t)}{2\partial x^2} + V_j\Psi_j(t) \tag{3}$$

Now, the Schrödinger equation is like a first order differential equation in time of a vector with a dimension j, where j is the number of spatial points. Integrals involving $\Psi(x,t)$ can be approximated using the discretized approximation of Ψ. For example,

$$\int \Psi^*\Psi dx = \int \|\Psi(x,t)\|^2 dx = \sum_j \|\Psi_j(t)\|^2 \tag{4}$$

$$\langle x\rangle(t) = \int \Psi^* x\Psi dx = \sum_j x_j\|\Psi_j(t)\|^2 \tag{5}$$

There is a difficulty in solving the this equation in that it is "stiff". Thus if you try the simple Euler approximation $\frac{\partial\Psi_j(t)}{\partial t} = \frac{\Psi_j(t+dt)-\Psi_j(t)}{\partial t}$ with the resulting equation

$$\Psi_j(t+dt) = \Psi_j(t) + \imath\partial t\left(-\frac{\Psi_{j+1}(t) - 2\Psi_j(t) + \Psi_{j-1}(t)}{2\partial x^2} + V_j\Psi_j(t)\right) \tag{6}$$

you will find the norm of the wave function diverges fairly quickly. This is characteristic of a "stiff" system, one in which the eigenvalues of the Jacobean matrix differ greatly in magnitude, causing instability. One of the primary postulates of Quantum Mechanics is the Principle of Conservation of Probability, which will clearly not occur if the norm of the wave function is diverging. Thus, a more accurate approach is needed. By extending the Euler approximation forward and backward in the standard way, we find that $\frac{\partial\Psi_j(t+\frac{dt}{2})}{\partial t} \simeq \frac{1}{2}\left(\frac{\Psi_j(t+dt)-\Psi_j(t+\frac{dt}{2})}{\partial t/2} + \frac{\Psi_j(t+\frac{dt}{2})-\Psi_j(t)}{\partial t/2}\right) = \frac{\Psi_j(t+dt)-\Psi_j(t)}{\partial t}$ and $\Psi_j(t+\frac{dt}{2}) \simeq \frac{\Psi_j(t)+\Psi_j(t+dt)}{2}$, which can be combined with Equation 3 to eventually yield

$$\Psi_j(t+dt) - \imath\frac{\partial t}{2}\Big(-\frac{\Psi_{j+1}(t+dt) - 2\Psi_j(t+dt) + \Psi_{j-1}(t+dt)}{2\partial x^2} + V_j\Psi_j(t+dt)\Big)$$

$$= \Psi_j(t + \frac{dt}{2}) =$$

$$\Psi_j(t) + \imath\frac{\partial t}{2}\Big(- \frac{\Psi_{j+1}(t) - 2\Psi_j(t) + \Psi_{j-1}(t)}{2\partial x^2} + V_j\Psi_j(t)\Big) \tag{7}$$

The right hand side of this equation is something that we can calculate, call it Φ_j.

$$\Psi_j(t+dt) - \imath\frac{\partial t}{2}\Big(- \frac{\Psi_{j+1}(t+dt) - 2\Psi_j(t+dt) + \Psi_{j-1}(t+dt)}{2\partial x^2} + V_j\Psi_j(t+dt)\Big) = \Phi_j(t) \tag{8}$$

The wave function at time $t + dt$ can then be obtained from solving

$$D_j\Psi_j(t+dt) + \mu\Psi_{j-1}(t+dt) + \mu\Psi_{j+1}(t+dt) = \Phi_j(t) \tag{9}$$

where $D_j = 1 + \Big(\imath\frac{\partial t}{2}\Big)\Big(V_j + \frac{1}{\partial x^2}\Big)$ and $\mu = -\imath\frac{\partial t}{4\partial x^2}$. This forms a tridiagonal matrix equation which can be most easily solved for $\Psi(t + dt)$ when it's written as

$$\mu\Psi_2 + D_1\Psi_1 = \Phi_1 \tag{10}$$
$$\mu\Psi_3 + D_2\Psi_2 + \mu\Psi_1 = \Phi_2 \tag{11}$$
$$\mu\Psi_4 + D_3\Psi_3 + \mu\Psi_2 = \Phi_3 \tag{12}$$
$$\vdots$$

These equations can be solved in the following manner. Multiply Eq. 10 by $-\frac{\mu}{D_1}$ and add to Eq. 11 to get

$$\mu\Psi_2 + D_1\Psi_1 = \Phi_1 \tag{13}$$
$$\mu\Psi_3 + d_2\Psi_2 = f_2 \tag{14}$$
$$\mu\Psi_4 + D_3\Psi_3 + \mu\Psi_2 = \Phi_3 \tag{15}$$
$$\vdots$$

where $f_2 = \Phi_2 - \mu\frac{\Phi_1}{D_1}$ and $d_2 = D_2 - \frac{\mu^2}{D_1}$. Now multiply Eq. 14 by $-\frac{\mu}{d_2}$ and add to Eq. 15 to get

$$\mu\Psi_2 + D_1\Psi_1 = \Phi_1 \tag{16}$$
$$\mu\Psi_3 + d_2\Psi_2 = f_2 \tag{17}$$
$$\mu\Psi_4 + d_3\Psi_3 = f_3 \tag{18}$$
$$\vdots$$

where $f_3 = \Phi_3 - \mu \frac{f_2}{d_2}$ and $d_3 = D_3 - \frac{\mu^2}{d_2}$. This can be repeated for all points. The algorithm can be written as
(1) $d_1 = D_1$ and $f_1 = \Phi_1$
(2) For j greater than or equal to 2 and less then or equal the number of spatial points, calculate

$$f_j = \Phi_j - \mu \frac{f_{j-1}}{d_{j-1}} \text{ and } d_j = D_j - \frac{\mu^2}{d_{j-1}}.$$

(3) Finally, calculate

$$\Psi_n = f_n / d_n \tag{19}$$

$$\Psi_{n-1} = \frac{f_{n-1} - \mu \Psi_n}{d_{n-1}} \tag{20}$$

$$\Psi_{n-2} = \frac{f_{n-2} - \mu \Psi_{n-1}}{d_{n-2}} \tag{21}$$

$$\vdots$$

Notice that if you evaluate the equations in the order given then the right hand side in each equation is already calculated by the time it is needed. This sequence of steps gives $\Psi_j(t + dt)$. An important property of this numerical propagator is that the norm of the wave function does not change significantly with time, as we will show. An important point to remember with this algorithm is that it used $\Psi_0 = 0$ and $\Psi_{n+1} = 0$. Thus, we have implicitly built in the stipulation that we are propagating Schrödinger's equation with the potential $V(x)$ with infinite walls at the points x_0 and x_{n+1}. Also, an interesting note is that we can also let $V_j \to V_j(t)$ without any problem in the derivation. Thus, this algorithm can easily handle time-dependent potentials, although this was never implemented or tested. The following code implements the algorithm:

```
/*
 * An relatively unoptimized version of the integrator derived above.
 */
void incrementInTime(Complex[] psi, double tStep)
{
    Complex[] d = new Complex[psi.length];
    Complex[] phi = new Complex[psi.length];
    double pref = tStep/(4.0*dx*dx);
    Complex mu = new Complex(0,-pref);
    for( int j = 1; j < psi.length - 2; ++j ) {
        double imag = .5*tStep*vPot[j];
        phi[j] = psi[j].times(1.0, -imag).add(psi[j+1].plus(psi[j-1]).mult(0,pref));
        d[j] = new Complex(1.0, imag);
    }

    for( int j = 2; j < psi.length; ++j ) {
        phi[j].sub(phi[j-1].times(offd).div(d[j-1]));
        d[j].sub(mu.squared().div(d[j-1]));
    }

    for( int j = psi.length - 2; j >= 1; --j ) {
        psi[j] = phi[j].minus(psi[j+1].times(mu)).div(d[j]);
    }
}
```

Finally, we note that this runs in $O(J)$ time. However, we have to sweep through the array 3 times performing many calculations each time. This algorithm is much faster when `Complex` objects are not used, reducing temporary object allocation overhead.

Calculating and Evolving the Eigenfunctions

As we've learned throughout quantum mechanics, every Hamiltonian $\hat{H} = \frac{\hat{p}_x^2}{2m} + V(\hat{x}) = \frac{1}{2m}\frac{\partial^2}{\partial x^2} + V(x)$ defines an orthogonal complete set of eigenstates $\varphi_n(x)$ with eigenvalues E_n such that

$$\hat{H}\varphi_n(x) = E_n\varphi_n(x) \tag{22}$$

Since $\varphi_n(x)$ form a complete set, we can express our state as $\Psi(x) = \sum_n a_n\varphi_n(x)$. We also know that the state $\varphi_n(x)$ evolves in time through application of the time evolution operator $\hat{U} = e^{-i\hat{H}t}$, so that $\varphi_n(x,t) = e^{-i\hat{H}t}\varphi_n(x) = e^{-iE_nt}\varphi_n(x)$. Our full state can then be expressed as

$$\Psi(x,t) = \sum_n a_n e^{-iE_nt}\varphi_n(x) \tag{23}$$

However, in order to determine E_n and $\varphi_n(x)$ for all n, we must solve the eigenvalue equation from 22, reproduced in descritized form here:

$$\frac{\partial^2 \varphi_{nj}}{\partial x^2} = 2m[V_j - E_n]\varphi_{nj} \tag{24}$$

using the same second second derivative approximation as before, we have

$$\varphi_{nj} - 2\varphi_{nj} + \varphi_{nj} = 2m\partial x^2 [V_j - E_n]\varphi_{nj} \tag{25}$$

which forms a matrix eigenvalue equation:

$$\frac{-1}{2m\partial x^2}\begin{pmatrix} -2 & 1 & 0 & 0 & 0 & \\ 1 & -2 & 1 & 0 & 0 & \\ 0 & 1 & -2 & 1 & 0 & \cdots \\ 0 & 0 & 1 & -2 & 1 & \\ & & \vdots & & & \ddots \end{pmatrix}\begin{pmatrix} \varphi_{n1} \\ \varphi_{n2} \\ \varphi_{n3} \\ \vdots \\ \varphi_{nj} \end{pmatrix} - \begin{pmatrix} V_1 \\ V_2 \\ V_3 \\ \vdots \\ V_j \end{pmatrix}\begin{pmatrix} \varphi_{n1} \\ \varphi_{n2} \\ \varphi_{n3} \\ \vdots \\ \varphi_{nj} \end{pmatrix} = E_n \begin{pmatrix} \varphi_{n1} \\ \varphi_{n2} \\ \varphi_{n3} \\ \vdots \\ \varphi_{nj} \end{pmatrix}$$

Thus, we need to find the eigenvalues of

$$\frac{-1}{2m\partial x^2}\begin{pmatrix} -2-V_1 & 1 & 0 & 0 & 0 & \\ 1 & -2-V_2 & 1 & 0 & 0 & \\ 0 & 1 & -2-V_3 & 1 & 0 & \cdots \\ 0 & 0 & 1 & -2-V_4 & 1 & \\ & & \vdots & & & \ddots \end{pmatrix} \tag{26}$$

This is a real, symmetric, tridiagonal matrix! A popular procedure called the TQLI (Tridiagonal QL Implicit) algorithm can find the eigenvalues of this matrix exceptionally well in $O(J^2)$ time. This algorithm can also find the eigenvectors (our eigenstates $\varphi_n(x)$), but it requires $O(J^3)$ time and is not integral to the eigenvalue calculation. Thus, when the potential is being modified directly by the user, we can calculate <u>only</u> the eigenvalues until the user is done. This will make the program adapt the energy levels in the potential window very quickly, but not allow us to evolve a state in time until the user is finished. I found the effect of being able to see the energy levels adjust in real-time to my modification much more satisfying than having a state evolve as I was doing it. The TQLI call looks like:

```
/*
 * Calculates the energy eigenvalues and optionally the eigenstates of
 * a potential defined in pot. If tempModes is null, eigenstates are not
 * calculated. If it is the identity matrix, eigenstates are stored into it.
 */
double[] calculateEigs(double[] pot, double[][] tempModes)
{
    double diag[] = new double[pot.length];
    double subDiag[] = new double[pot.length];
    double fac = -1/(2 * massBar.getValue() * dx * dx);
    for( int i = 0; i < pot.length; ++i ) {
        subDiag[i] = fac;
        diag[i] = -2 * fac - pot[i];
    }

    tqli(diag, subDiag, pot.length, tempModes);
    double[] E = diag;

    if( tempModes == null )
        sort(E);                // Sorts the energies
    else
        sort(E, tempModes);     // Sorts the energies,
                                // preserving the mode <=> energy indices
    return E;
}
```

See *Numerical Recipes in C++* for a discussion and presentation of the TQLI algorithm. Very few modifications were made to it and a discussion of the matrix decomposition does not seem appropriate here.

We see the importance now of requiring that the potential is infinite on each of the boundaries. If this were not the case, the matrix size would be unbounded and, just like the iterative method, our eigenvalues would case the state to diverge eventually. In the case with infinite potential walls, we understand that the state goes to zero at the boundaries and can stop there. Although there are still unavoidable inaccuracies, we at least have a bounded, sensible, and controllable eigenstate.

Thus, we can now determine the eigenstates of any potential relatively quickly and easily and construct a wave function by either combining the eigenstates or decomposing an existing state (such as a user-defined Gaussian):

```
/*
 * Construct a normalized Gaussian wavefunction with initial
 * <x> = xInit
 * <p> = pInit
 * psi needs to be initialized to the length (resolution) of the system.
 */
void constructGauss(Complex[] psi, double xInit, double waveWidth, double pInit)
{
    double sum = 0;
    double x = -xInit;
    double s;
    for( int i = 1; i < psi.length-1; ++i ) {
        x += dx;
        s = x/waveWidth;

        psi[i] = Complex.exp(-s*s, pInit*x);
        sum += psi[i].absSq();
    }
    psi[0] = psi[length] = Complex.zero;

    double norm = 1.0/Math.sqrt(sum * dx);

    for( int i = 1; i < psi.length-1; ++i ) {
        psi[i].mult(norm);
}

/* A state is a Complex-valued function which we wish to express as a
 * series of orthogonal eigenfunctions. modes[n] is the real-valued
 * nth eigenfunction. This returns a Complex-valued array of projections
 * onto the eigenstates which we can use to draw phasors and evolve the
 * original state in time easily.
 */
Complex[] decomposeState(Complex[] psi, double[][] modes)
{
    Complex[] proj = new Complex[modes.length];
    for( int n = 0; n < modes.length; ++n ) {
        Complex sum = new Complex();
        double[] eigSt = modes[n];
        for( int i = 1; i < state.length; ++i )
            sum.add(psi[i].times(eigSt[i]));

        if( sum.isSmall(epsilon) )
            sum.setToZero();

        proj[n] = sum.times(dx);
    }
    return proj;
}
```

```
/*
 * This function takes a time step, the list of a_n projections, the
 * eigenfunctions of the potential, and the energy eigenvalues E to
 * calculate a new state psi. Note this function also updates the
 * proj array by a phase determined by the time step.
 */

Complex[] evolveStateInTime(double tStep, Complex[] proj, double[][] modes, double[] E)
{
    t += tStep;
    Complex nMult;
    Complex[] nMode;
    Complex[] psi = Complex.zeroArray(psi.length);

    for( int n = 0; n < modes.length; ++n ) {
        if( proj[n].isZero() )
            continue;
        nMult = proj[n].mult(Complex.exp(0, -E[n] * t));
        nMode = modes[n];
        for( int i = 1; i < psi.length - 1; ++i ) {
            psi[i].add(nMult.times(nMode[i]));
        }
    }
    return psi;
}
```

Thus, after precalculating the energy eigenvalues and corresponding eigenfunctions, we can calculate the state at any point in time in O(NJ) time (a high bound on many states since we can skip zeroed projections).

An interesting result that we can take advantage of is that since we can calculate the state at any time, we can not only let `tStep` be related to a user-defined speed control device, but ALSO be related to the last time we updated the graph. Thus, presuming we can draw the graphs faster than 15-20 frames/sec (not difficult), the output will be smooth and fewer 'jumps' will occur when the system misses a beat in our process. The incremental algorithm cannot do this nearly as elegantly or stably.

Converting to Momentum Space

In quantum, we saw that a wave function can be converted from position space to momentum space through:

$$\langle x|p\rangle = \frac{1}{\sqrt{2\pi}}\, e^{ipx} \tag{27}$$

$$\Psi(p) = \langle p|\Psi\rangle = \int dx \langle p|x\rangle\langle x|\Psi\rangle = \int dx \frac{1}{\sqrt{2\pi}}\, e^{ipx}\Psi(x) \tag{28}$$

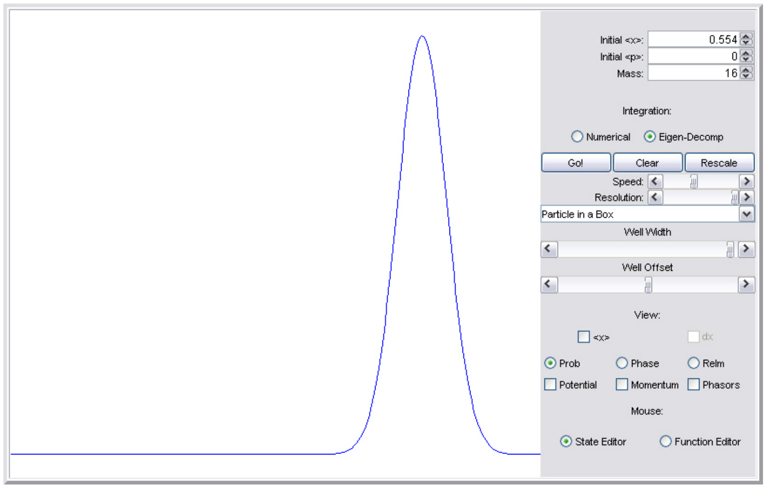
This is the well known Fourier Transform relationship between position and momentum space. Since it is so popular, it has many simple solutions that are readily implemented. I used the double precision `dfour1` algorithm found in *Numerical Recipes*, section 12.2. It takes a single array of length $2N = 2^x$ representing [real, imag] complex pairs and replaces this array with its discrete Fourier transform. In order to call this routine, we have to make sure the array is the smallest array we can make that is larger than our resolution but still a power of 2 in length. Additionally, I made some simple modifications to the FFT algorithm so that it will work with our `Complex` class (i.e. made a mapping between $2N = 2^x$ and $N = 2^x$, combining adjacent cells). Thus, we'll simply copy the data into an array for the `dfour1` function:

```
/*
 * This takes our state psi and projects it into momentum space by performing a FFT.
 * pSize is a power of 2 that is just larger than psi.length.
 * First, the psi must be fftshifted so that the center corresponds to the f0 position.
 * This rearranging preserves periodicity and pads the center with zeroes (note that psi
 * goes to zero anyway)
 */
Complex[] getPFunc(Complex[] psi, int pSize)
{
    Complex[] p = Complex.zeroArray(pSize);
    int halfCount = sampleCount/2, index, n = psi.length - 1;

                       // fftshift so that x = 0 (psi[halfCount]) is in first position
    for( int i = 0; i < psi.length; ++i ) {    // and also pads the center with zeroes
        index = i <= halfCount ? halfCount - i : p.length + halfCount - i;
        p[index] = psi[n - i];
    }

    dfour1(p, pSize, 1);     // Modified to accomodate our Complex class
    // Rather than waisting time fftshifting and normalizing,
    // I let the momentum grapher do it implicitly
}
```

Although the adaptation of `dfour1` to use our `Complex` class causes it to have a small amount of undue overhead, I find that it's beneficial later on in graphing. It prevents us from having to make multiple conversions between `double` arrays and `Complex` arrays amid calculation and graphing as well as preserving homogeneity and reuseability of annoying graphing code.

Comparing Conservation of Probability

One interesting comparison between these two approaches is how well they uphold the Law of Conservation of Probability. To test this, we obtain a printout of the norm of the wave function at every iteration for some benchmark test under each algorithm. The benchmark initial conditions are shown in Figure 1. I hardcoded a Gaussian benchmark in the infinite well potential. Take note of the highest resolution setting.



Initial <x>: 0.554
Initial <p>: 0
Mass: 16
Numerical [✓ Eigen-Decomp]
Particle in a Box
[✓ Prob] Phase ReIm
[✓ State Editor] Function Editor

Figure 1: A plot of the wavenorm over time using each of the integration methods.

I also verified that the `tStep`'s were calibrated correctly so that over the course of these iterations, approximately the same final state was reached. We can now run this benchmark using each of the methods presented here. Figure 2 shows the results of these two methods.

As we expect, the eigenfunction decomposition algorithm has no error-build up since it simply does not even use the previous $\Psi(x,t)$ in its calculation of $\Psi(x,t+dt)$. Thus, any error that is present is roundoff error (which actually did create an error of approximately 25×10^{-15} in each data point) and is not compiled over time. In contrast of course, the direct numerical method does depend on its previous $\Psi(x,t)$ so error is accumulated, though slowly indeed. In the 35000 time iterations it was run, the norm decreased by less 0.06%. This corresponds to approximately 2 minutes of actual runtime. Last year, I estimated that the system would have to be run under this numerical integration scheme for approximately 2-3 hours to have the

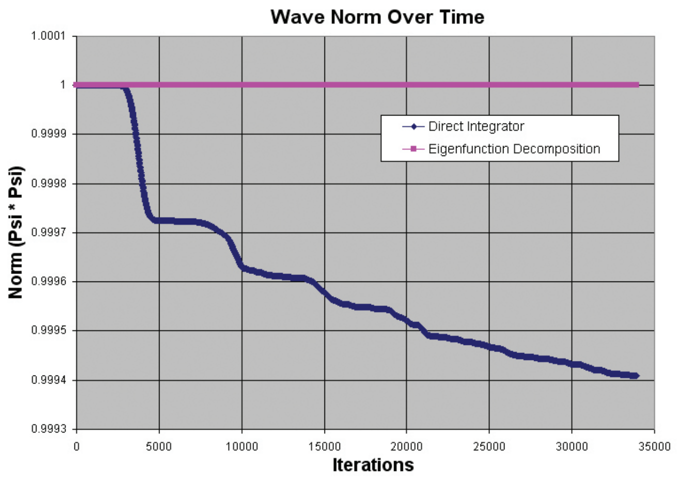


Figure 2: A plot of the wavenorm over time using each of the integration methods.

norm deviate a ”non-negligible” amount of 1%. At this point, the system would be of no qualitative or quantitative value anyway.

Future Plans and Development

Last year I also made measurements of the transmission coefficient of a Gaussian packet through a barrier with varying initial momentum. I’d like to perform the same test with the new eigenfunction method but, unfortunately, my units no longer correlate as well as they used to. It would just take another day or two to work this out... if I hadn’t screwed up the linker between the GUI and the actual physics methods described above. It would take the same day or two to get everything rewired correctly so that it’s actually usable. I seem to be out of time... Also, this project has spent more time on the Computer Science front in designing and implementing the applet than the physics algorithms themselves and may have finally managed to break my moral regarding patient applet design. Right now it’s not in a usable state, but I’ll likely be fixing it over the summer.

Some improvements that could be made include unrolling the `Complex` class within the code. This would make it a lot messier, but also faster since there would be no class construction overhead. It seems that Paul Falstad does this in his applet and uses various hacks to get the data he wants. When it is running, it seems to do fine, but this is one optimization that could be done to squeeze out a couple more resolution points.

Another thing to work on is making it much more conducive to data acquisition. This includes writing some general data-getting methods (such as a better, possible interfaced, norm integrator; calibrating units; etc) so that I could perform something like the barrier penetration comparison. I'm very curious to see what effect the finite, and restricted, resolution from the decomposition of the packet into eigenstates has on the test. Specifically, if it's better or worse than the direct numerical integrator, which suffers from finite resolution in space and time. It seems that Falstad's can't do this either because he can't adjust the the X Gaussian and P Gaussian independently, whereas in mine you can. You'd have to draw one yourself 3/4 offset in the box, not leaving you much room to measure the results. In mine, I have a barrier potential preset that sets up the Gaussians (adjustable) for you.

Implementing this in Matlab would give me many options for data analysis. The built in complex values would be perfect.

Conclusion

My goal was to create a tool that could be used to test and develop intuition in quantum mechanics. We have derived a direct numerical method for solving the One-Dimensional Time-Dependent Schrödinger Equation and have shown that it functions faithfully in real time. Furthermore, we've presented an alternative method involving the decomposition of a given state into its eigenstates and evolving those in time. We have shown that both methods preserve the norm of a wave function to an acceptable degree, upholding the Law of Conservation of Probability. In the future, I would like to show that these methods also uphold theoretical predictions in quantum tunnelling and examine any differences between the two methods.

The strength of this tool is present in its clarity, speed, and overall accuracy of its results. Being able to watch the real time evolution of a wave function over some potential gives an insight into the inner workings of quantum mechanics that straight theory simply fails to.

I'll give you a note once I rewrite the linker code and get it back online to play with. The picture of the initial conditions for the Conservation of Probability test shows the final GUI that I'm trying to rewire one last time.

References

[1] A. Askar and A.S. Cakmak, Explicit Integration Method for the Time-Dependent Schrödinger Equation for Collision Problems, J. Chem. Phys. (1978).

I used this source in conjunction with Visscher to formulate my version of the numerical integrator shown above. Although the math presented in this article was of a level high enough to be out of my scope, I was still able to use concepts from special cases in writing the applet and integrator.

[2] P.B. Visscher, A Fast Explicit Algorithm for the Time-Dependent Schrodinger Equation, Computers In Physics, 596598 (Nov/Dec 1991).

I used this source in conjunction with Askar and Cakmak to formulate my version of the numerical integrator. This article gives an algorithm to calculate the time-dependent Scrödinger equation, which I implemented with some modifications and assumptions so as to make it run faster and be able to function in real time.

[3] Robert Eisberg and Robert Resnick, Quantum Physics (John Wiley & Sons, Inc., New York, 1974)

Class Text.

[4] Paul Falstad. 1D Quantum Mechanics Applet.

Although I didn't get to use any of the code (yet), I used his applet as a model for what I wanted to accomplish and be able to produce. I didn't get everything implemented that he can do, but I got most. With his guide, I was able to expand my applet and computational physics repertory a great deal through this project.

[5] William H. Press et al., Numerical Recipes in C++: The Art of Scientific Computing, Cambridge U. Press

The computational physicists' bible. Discusses many algorithms including FFTs and TQLI, which were both used in this project.

[6] L. G. de Pillis, *private communcation*, 2004.

In Scientific Computing Math 164, we learned about stiff systems and solvers. I was able to recognize the system was stiff and work around it from de Pillis.

[7] Cecka, Cris. A Real-Time Numerical Integrator for the One-Dimensional Time-Dependent Schrödinger Equation. Final Paper and Project for Physics 52. April 2004.