Visualizing Quantum Mechanics with Diffusion Monte Carlo

Melissa A. Mamura
UMSA Seminar
The College of William and Mary
Williamsburg, VA 23185
mamamu@wm.com

ABSTRACT
Quantum Diffusion Monte Carlo is one of the most commonly used implementations in the study of many-particle systems in quantum physics. From Diffusion Monte Carlo, a fairly accurate approximation of the wave function can be obtained, given merely a potential energy surface. This paper reexamines the classic quantum mechanic problem of The Particle in a Box, with the use of graphical real-time simulation, as motivation for graphical simulations in the study and research of quantum mechanics.

General Terms
Monte Carlo, Random walk

Keywords
Diffusion Monte Carlo, Schrödinger equation, Green's function, Importance sampling

1. INTRODUCTION
Diffusion Monte Carlo (DMC) is a powerful analytical tool for solving the ground-state wave function of simple particle systems. The method is essentially a Markov process, or random walk, which makes use of the variational principle, which states that a wave function adjusts itself until the particle has the smallest possible total average energy. Random walks simulate the diffusion process, from which the ground state wave function can be obtained. From hereon, particles will be referred to as random walkers.

DMC is based on the fact that the Schrödinger equation can take on a form, which has an imaginary time variable, τ, incorporated into the wave function. The Schrödinger equation solves for the wave function and corresponding energy of a particle. The time-dependent Schrödinger equation is as follows.

$$|\psi(\tau_1 + \delta\tau)\rangle = e^{-\hat{H}\delta\tau}|\psi(\tau_1)\rangle \quad , \qquad (1)$$

where the state | ψ) evolves from imaginary time **τ** to a later time $\tau_1 + \delta\tau$

Green's function,

$$f(\mathbf{R}', \tau_0 + \tau) = \int G(\mathbf{R}', \mathbf{R}; \tau) f(\mathbf{R}, \tau_0) d\mathbf{R} \qquad (2)$$

propogates the wave function, by giving the probability that a walker will move to the next time step.

In DMC, a large number of walkers are initially placed in a configuration that resembles the system to be modeled. For example, for the Particle in a Box, walkers are placed on a line. Walkers then move across the space configuration with a random step size in imaginary time. Walkers multiple or disappear with a given probability. After a large time, τ,, the distribution of walkers converge into a steady state. This distribution of walkers is also known as the ground-state wave function.

The wave function is significant in that it describes the probability of a particle being at a particular point. The wave function is key to extracting or extrapolating properties about a particular system. Although the Schrodinger equation can be solved numerically, DMC offers a comparable solution, even in cases when the problem is intractable and cannot be solved by numerical methods.

Computer simulation has become an increasingly integral part of quantum mechanics research. However, relatively little visualization of such simulations has taken place. Visualization provides a greater understanding of particle behavior and functions as an educational tool for solving more complex models. This paper reexamines the classic Particle in a Box problem as motivation for the real-time visualization of the state of a given particle system.

2. Modeling the Particle in a Box
The Particle in a Box problem was implemented as a Java applet to allow an easy, aesthetic interface for the user. Java is most ideal for animation and graphics; however, its trade-off is its computational efficiency. Since the Particle in a Box is a relatively simple problem paradigm, a Java applet was chosen over a stand-alone for implementation.

The applet models a particle in a box in one dimension and displays four plots, which examine different parameter dependencies of the system. The user-specified parameters are as follows:

- Initial number of walkers, *N0*;
- Box-half width, *width*;
- Time step, *dt*;
- Number of bins for psi(x) plot, *numBins*;
- Number of steps for equilibration, *Nequil*;
- Number of steps per block, *blockSize*;
- Trial wave function, *twf*;
- Number of walkers to sample for the plot, *sampleSize*.

The DMC algorithm involved in the propagation of the ensemble is somewhat unconventional, but it involves fewer computations

than the conventional DMC algorithm for the Particle in a Box problem. This compensates for the additional computation and memory needed for the graphical implementation. The algorithm implemented is as follows.

Step 1 Initialize the ensemble of N0 walkers at random positions x_i

Step 2 Select a small time step dt to propagate the ensemble. Let t = 0 be the initial start time. Use a trial wave function that approximately resembles the shape of the actual wave function, to guess the ground state energy.

Step 3 Propagate each walker in the ensemble until walkers have a new position at time t+dt

- A. Randomly select a walker with positions x_I at time t.
- B. Let the new position be $x_I' = x_I + (\Delta x)_\sigma$, where $(\Delta x)_\sigma$ is a random number from a Gaussian distribution with width $\sigma = \sqrt{2D\Delta t}$. If x_i' is outside the boundary of the box, reject the position, and go to Step A.
- C. Calculate $P(x_i',x_I;\Delta t)$, which is the probability of moving from x_I to x_i'.
- D. Let Uniform(0,1) be a random number selected from a uniform distribution between 0 and 1. If Uniform(0,1) > P, accept the move; otherwise, reject it, and go to Step A.
- E. Repeat Steps A-D until N new walkers, with new positions, are in the ensemble. Note that this causes the number of particles to be held constant throughout t.

Step 4 If necessary, adjust the energy to keep the number of particles within the boundary conditions. Let the new energy value be $E_t' = E_t - a\ (N(t) - N0)/(N0(\Delta x))$, where N(t) is the number of walkers at time t, and N0 is the number of initial walkers.

Since the walker population is stabilized (held constant), scaling techniques are not necessary, and so our model is somewhat simplified, involving fewer computations. The model of the Particle in a Box problem presented here is fairly simple, since the purpose is to provide a foundational basis for future work in visualizing quantum mechanics. To first test the computational threshold of the applet, no weights or branching technique was used in the implementation. Branching, while a statistical efficiency improving technique, is computationally expensive in terms of a graphical implementation. However, the Java applet does make use of Importance Sampling, which is a technique that exploits the fact that the shape of the wave function is known. Each walker also takes a variable step size, instead of some fixed size, which is a more realistic representation of a lightweight particle system.

3. Visualizing the Particle in a Box

The Particle in a Box displays four graphs, along with a few basic statistical real-time data. The parameter values of the applet are instantiated every time the `reset` button is selected. The `step` button represents the next computational step. Each time the `step` button is pressed the number of *Skip Steps* specified are skipped, and its corresponding data is not added into the statistical output. The number of equilibration steps specified by the user also determines the number of Monte Carlo steps to be skipped before data is to be added to the statistical output, displayed in the upper right box.

The source code makes use of many modulus operations to avoid creating many instances of large arrays. Since Java does not allow for pointers, in a way that languages such as C or C++ does, code optimization was a major consideration for computationally expensive Monte Carlo simulations. Reusability of the source was also another consideration, and so the design of the applet is somewhat generic, displaying the current number of walkers, although it is held constant.

The first graph displays the position of *sampleSize* number of walkers' positions as a function of time. Each of the different walkers has a different color to distinguish walkers from each other. The potential energy is not plotted, but it is intrinsically defined to be 0 within or at the walls of the box and some infinity beyond the walls. Notice how Green's function causes the walker position to 'bounce' off the walls of the box in Figure 1. The second graph is of the current energy as a function of time. The third graph, below the `Current Stats` display, shows the total number of walkers at time t. The fourth graph is merely the normalized probability distribution function of the first graph. Most of the walk positions are located near the center of the infinitely deep box, and the height of the wave function, *psi(x),* peaks at the center of the box as well. The fourth graph also displays the actual wave function for comparison purposes.

4. Results

The results of the applet implementation verify well-known statistical dependencies of the DMC method, in particular, within the context of the Particle in a Box problem.

For Tables 1-4, the following parameters are kept constant: *dt* = 0.01, *width* = 1.0, and *twf* = polynomial. The runtime for each of these tables was approximately *t* = 50.0, or 50/*dt* Monte Carlo steps.

4.1 Number of Walkers vs. Energy

The first set of examined dependencies is the statistical dependency of the Energy, *E*, on the number of walkers, *N*. As the number of walkers increases, the experimental energy value approaches its theoretical value. The standard deviation and mean error correspondingly decrease with the increase in the number of walkers. Figures 2 and 3 show the change in Energy with different N0 values. Table 1 summarizes the statistical data from the four runs of the applet.

Figure 1: Particle in a Box Applet

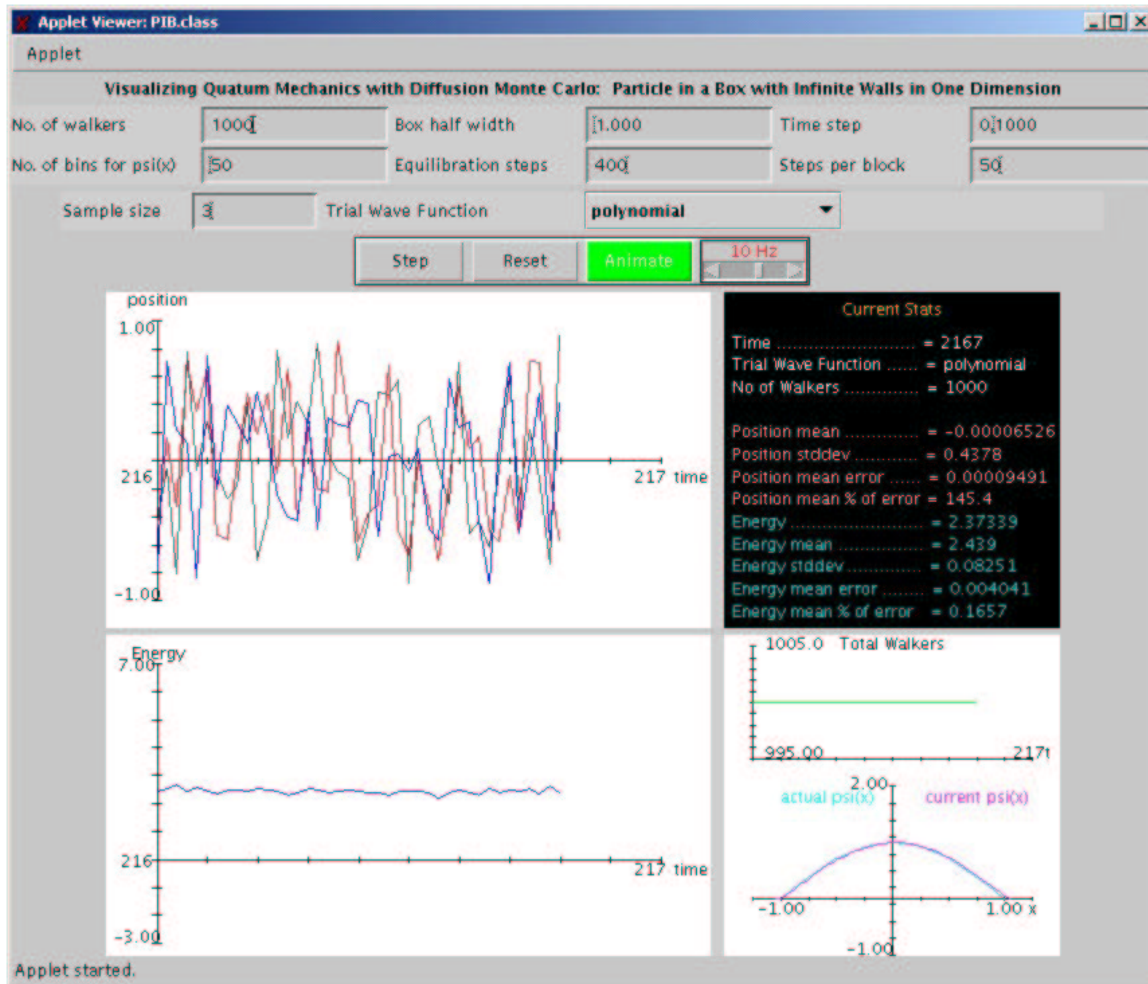


Figure 2: Number of Walkers = 100

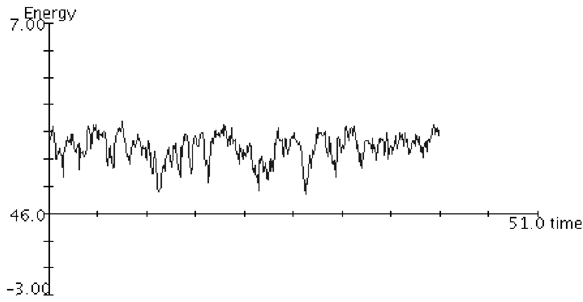


Figure 3: Number of Walkers = 5000

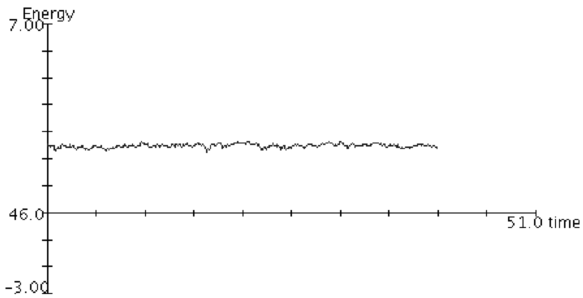


Table 1: Dependency of Energy on Number of Walkers

No of Walkers	Energy			
	E0	**Mean**	**Stddev**	**Mean Error**
100	3.23268	2.402	0.4975	0.04701
500	2.43871	2.470	0.1961	0.75040
1000	2.49717	2.448	0.01369	0.01369
5000	2.35033	2.468	0.06368	0.00617

4.2 Steps per Block vs. Energy

The next statistical dependency trial runs were conducted on the effect of number of steps per block on *E*. In this case, the constant parameters were N = 1000 and *Nequil* = 400. Figures 4 and 5 show the difference from *SkipSteps* = 0 to *SkipsSteps* = 400. From the summary table, we can see that the mean error decreases as *SkipSteps* increases. The shape of the plots are unaffected, because *SkipSteps* only affects the data being computed for statistical purposes. Computing the E after every time step *dt*, can slow down the run of the applet considerably, and so it is advisable to select an optimal number of steps to skip in each block (or for each press of the `step` button on the applet). As the number of steps per block increases, the mean error also increases.

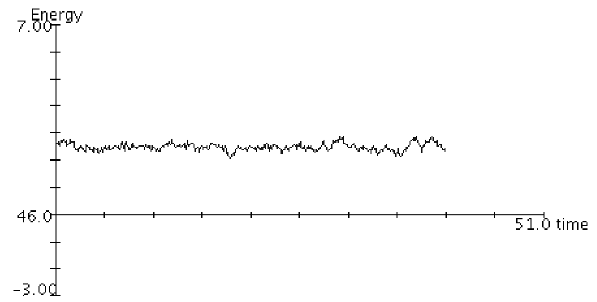


Figure 5: Number of Steps per Block = 400

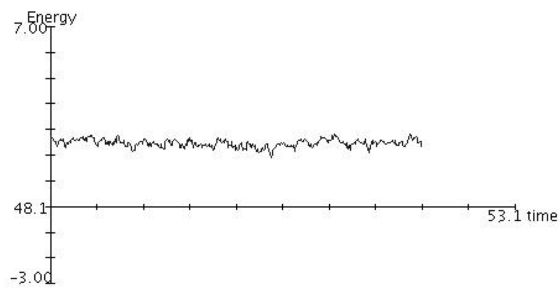


Table 2: Dependency of Energy on Block Size

No of Steps per Block	Energy			
	E0	**Mean**	**Stddev**	**Mean Error**
0	2.53556	2.463	0.1390	0.00205
40	2.36266	2.439	0.1452	0.01372
100	2.46171	2.499	0.1207	0.01724
400	2.52298	2.433	0.1617	0.04875

4.3 Equilibration Steps vs. Energy

The number of equilibration steps is specified in order to discard erroneous statistical data from the start of the diffusion process. In the next four trial runs, the constant parameters were: N = 1000 and *SkipSteps* = 40. As the number of equilibration steps increases, *E0* should approach its theoretical value, since the starting point at which statistical calculations begin should be near or at the steady state. However, since Importance Sampling is used, *E0* is already near its theoretical value, and so the four trials are insufficient to verify this trend.

Figure 6: Nequil = 0

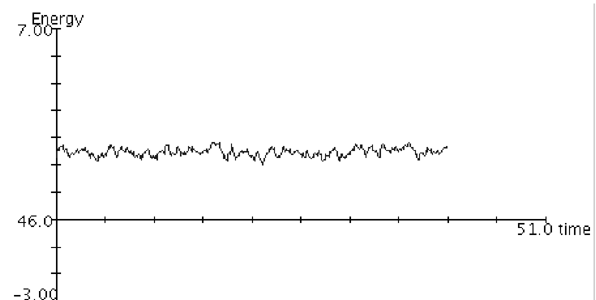


Figure 7: Nequil = 400

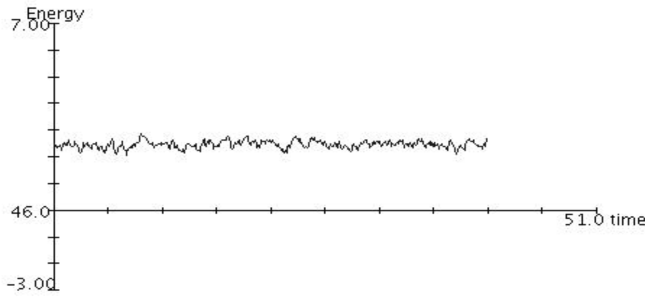


Figure 9: dt = 0.100

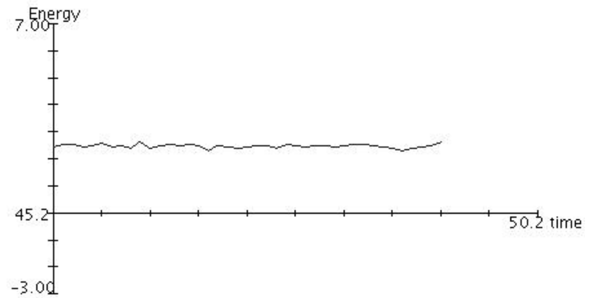


Table 3: Dependency of Energy on Equilibration Steps

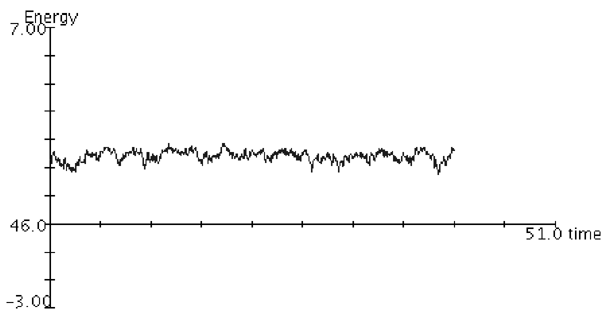
No of Equil Steps	Energy			
	E0	Mean	Stddev	Mean Error
0	2.62826	2.452	0.1413	0.01285
40	2.48841	2.467	0.1313	0.01194
100	2.29336	2.464	0.1326	0.01216
400	2.54996	2.464	0.1474	0.01392

Table 4: Dependency of Energy on Time Step Size

Time Step Size	Energy			
	E0	Mean	Stddev	Mean Error
0.005	2.64053	2.447	0.1917	0.01253
0.010	2.39067	2.454	0.1375	0.01299
0.050	2.48988	2.478	0.08084	0.02160
0.100	2.63077	2.528	0.08394	0.05935

4.4 Time Step vs. Energy

With a small *dt*, greater accuracy can be achieved in the estimate of the ground state energy. The parameters held constant for the four trial runs are: *SkipSteps* = 40, *Nequil* =400, and *N* = 1000. In Table 4, contrary to popular believe, the standard deviation of the *E0* actually increases as *dt* decreases. When the Particle in a Box applet is run to display a sample size of three or more walkers, it appears that a *dt* with a very small size causes an increase in variance between walker positions, as reflected with the fluctuating *psi(x)* function. Since the step size of the walkers relies on *dt*, the selection of a 'good' time step value is not necessarily a trivial task. The experimental results displayed in Figures 7 and 8, provide an empirical validation for this non-intuitive walker behavior.



5. Conclusions

The Schrödinger equation can actually be solved numerically through a series of partial differential equation integrations, and so this paper may seem trivial. However, the numerical solution provides verification and validation for the graphical implementation. The exact solution for the Particle in a Box problem, with inifinite walls and *width* = 1, is 2.4674. The actual shape of the wave function resembles a cosine function, and the use of the Particle in a Box applet verifies the rapid convergence of the experimental *psi(x)*, in comparison to the polynomial trial wave function.

In evaluation of the Particle in a Box applet, it functions well and performs at a sufficient speed to run simulations, given a modest populations size. However, when *N* exceeds a value greater than 300,000, the speed of the applet begins to decrease drastically. Furthermore, time axis on three of the graphs is not determined relative to *dt*, and so the walker paths are difficult to distinguish as *dt* becomes smaller. While the Java applet performed well for the Particle in a Box problem, with our specified conditions, a Java applet implementation would not be ideal for modeling more sophisticated many-particle systems. The use of Java wrapper classes with C code may be a better method of implementation of future visual simulation of quantum mechanics. In addition methods for optimization of the DMC's algorithm may help encourage the use of visualization for quantum mechanic simulations.

6. References

[1] Wirawan, Purhanto. Diffusion Quantum Monte Carlo by Example. The College of William and Mary.

[This page left intentionally blank.]

7. Appendix

7.1 Appendix 1: Particle in a Box, PIB.java

Note that PIB.java is only the main source file, and supporting java files are not included in this listing.

```
// ************************************************************
// Title      : PIB.java, Particle in a Box
// Author     : Melissa Mamura, The College of William and Mary
// Date       : April 23, 2002
// Purpose    : To display the behavior of particles with the
// One-Dimentional Schrödinger equation. This is the an overly
// simplified model, which makes use of an unconventional algorithm
// for DMC. The number of walkers is constant throughout the simulation.
// Therefore, there are no scaling mechanisms implemented. There are
// no weights or branching. The step size is variable, and dt is
// fixed.
//
// Acknowledgement* : The code for this applet was partly adapted from
// C++ code written by Wirawan Purwanto, a Physics PhD candidate at
// The College of William and Mary. This applet also makes use
// of the comphys library from http://www.physics.buffalo.edu/ComPhys/
// ************************************************************

import comphys.graphics.*;
import comphys.Easy;
import java.awt.*;
import java.awt.event.*;
import java.lang.*;
import java.util.*;
import dmc.*;
import javax.swing.*;


public class PIB extends Animation {

    double[] x;                 // walker positions at time t
    double[] psi;               // wave function produced by a diffused trial wave funtion

    double[][] sample;          // sample of walkers
    int sampleSize = 1;         // # of walkers to sample = 1; otherwise, graph is messy
    int[] sindex;               // array of walker indexes chosen at random

    double[] Energy;            // energy at time t
    int N0 = 5000;              // desired number of walkers
    double width = 1;           // half width of initial region
    int N;                      // number of walkers
    double[] TotalWalkers;      // record of total number of walkers for plotting
    double dt = 0.01;           // time step
    double t = 0.0;             // time
    //    double tmin = 0;              // min time value for plot
    //double tmax = 5;              // max time value for plot

    double xmin = 0;            // minimum x for plotting
    double xmax = 5;            // maximum x for plotting

    int maxT = (int) Math.round((xmax - xmin) * (1/ dt));

    int numBins = 50;           // number of bins for psi(x)
    String twfName = "cosine";  // trial wave function
    int mcs;                    // Monte Carlo steps
    int nequil = 400;           // equilibriation steps
    double binwidth;            // binwidth for psi array
    int maxN = N0 * 10;         // max number of walkers
    Color[] Colors = {Color.red, Color.darkGray, Color.blue, Color.green, Color.magenta,
                              Color.cyan, Color.pink, Color.orange};
    WaveFunction twf = new WaveFunction();   // trial wave function
    WaveFunction awf = new WaveFunction();   // actual wave function
    BasicStats WalkerStats = new BasicStats();
    BasicStats EnergyStats = new BasicStats();
```

```
void initial ()
// ***************************************************************
// initialize variables here
// only create new arrays of the new size is larger than the
// initialized size. here we rely on Java's garbage collector
// ***************************************************************
{
        maxN = N0 * 10;
        if(x == null || maxN > 50000) {
          x = new double[maxN+1];          // create array of walker positions
        }

        maxT = (int) Math.round((xmax - xmin) * (1/ dt));

        if(Energy == null || maxT > 500) {
          Energy = new double[maxT*2];       // create array of energy values
        }

        if(sindex == null || sampleSize > 1) {
          sindex = new int[sampleSize];      // create array to store sample walkers' indexes
        }


        sample = new double[sampleSize][maxT*2];  // create array to hold sample walker positions


        for(int i = 0; i < sampleSize; i++) {    // select walkers to sample at random
         sindex[i] = (int) (N0 * Math.random());
        }

        for(int i = 0; i < sampleSize; i++)      // initialize sample walker positions
         for(int j = 0; j < (maxT*2); j++)
           sample[i][j] = 0;

        if(TotalWalkers == null || maxT > 500) {  // create array to hold walker positions
          TotalWalkers = new double[maxT*2];    // for plotting
        }

        for(int i = 0; i < (maxT*2); i++) {      // initialize total walkers array
          TotalWalkers[i] = 0;
        }

        for(int i = 0; i < (maxT*2); i++) {      // initialize energy array
          Energy[i] = 0;
        }

        if(psi == null || numBins > 50) {        // create array to hold psi values
          psi = new double[numBins+2];
        }

        for(int i = 0; i < numBins+2; i++) {     // initialize psi array
          psi[i] = 0;
        }

  mcs = 0;          // initialize time to 0
        t = 0.0;

        xmin = 0;          // minimum x for plotting
        xmax = 5;          // maximum x for plotting

        WalkerStats.Clear();          // clear stats
        EnergyStats.Clear();
  N = N0;                  // initial # of walkers = desired # of walkers
        RandomizeWalkers();           // choose initial positions of walkers at random
        binwidth = 2 * width / numBins;  // binwidth for psi array
        TotalWalkers[mcs%(maxT*2)] = N;  // record N for plotting
  data();                  // accumulate data
}
```

```
private int limit(int x, int xmin, int xmax)
// ***************************************************************
// used for finding the appropriate bin index
// ***************************************************************
{
  if(x > xmax)
          return(xmax);
  else
          if(x < xmin)
             return(xmin);
  return(x);
} // limit()


double V (double x)
// ***************************************************************
// The corresponding potential function
// ***************************************************************
{
        if(-width <= x && x <= width)
           return 0;
        return(Double.NaN);
} // V()


public void RandomizeWalkers()
// ***************************************************************
// randomizeWalkers: intitializes walkers with random positions
// ***************************************************************
{
        for(int i = 0; i < N; i++)
           x[i] = (2 * Math.random() - 1) * width;
}

private static Random rng = new Random();  // creates a random number generator

public void MoveWalkers()
// ****************************************************************
// MoveWalkers: moves walkers according to Green's function random distribution
// G_d normalized => D = 1
// ****************************************************************
{
        double W[] = new double[N];        // temp buffer for new walker positions
        double x0, x1;                     // old and new positions
        double sigma = Math.sqrt(2 * dt);  // variance of GF's Gaussian bell
        double it = 1.0 / dt;              // imaginary time value
        double P;                          // P(x'', x')
        int i, j;                          // indexes

        for(i = 0; i < N;i++) {
           do {
                   do {
                      j = (int) (N * Math.random()); // select a walker randomly
                      x0 = x[j];
                      x1 = x0 + sigma * rng.nextGaussian();
                   } while (Math.abs(x1) > width);    // find a new position in the box
                   P = (x0 > 0 ? Math.exp(-it * (x1 - width) * (x0 - width)):
                       Math.exp(-it * (x1 + width) * (x0 + width)));
           } while(Math.random() <= P);
           W[i] = x1;                              // store into temp buffer
        }
        // replace old positions with new ones
        for(i = 0; i < N; i++) {
          x[i] = W[i];
        }

   mcs++;              // increment number of monte carlo steps
        t += dt;            // increment current time value

        for(i = 0; i < sampleSize; i++) {
          sample[i][mcs%(maxT*2)] = x[sindex[i]];
```

```
        }

        TotalWalkers[mcs%(maxT*2)] = N;

} // MoveWalkers()


void GetEnergy(String wavefunction)
// ***************************************************************
// GetEnergy: gets the energy value at time t
// ***************************************************************
{
        double numInt = 0;    // numerator
        double denInt = 0;    // denominator
        double sum = 0;       // sum

        for(int i = 0; i < N; i++) {
           twf.GetTrialWaveFunction(wavefunction, x[i], width);
           numInt += twf.Hphi();
           denInt += twf.phi();
        }

        Energy[mcs%(maxT*2)] = numInt / denInt;   // store Energy at time t for plotting

} // GetEnergy()


void zeroData ()
// ***************************************************************
// reset data
// ***************************************************************
{
        for(int i = 0; i < numBins+2; i++)
           psi[i] = 0;
        WalkerStats.Clear();

} // zeroData()

void data ()
// ***************************************************************
// accumulate data
// ***************************************************************
{
   // bin walkers
   for (int i = 0; i < N; i++) {
           int bin = (int) ((x[i] + width) / binwidth + 1e-13) + 1;
           int k = limit(bin, 0, numBins + 1);
           psi[bin]++;

      if(mcs - nequil > 0)
                 WalkerStats.AddData(x[i]);        // store walker position data

   } // for()
} // data()


class Graph extends Plot
// ***************************************************************
// Graph: graph of walker positions as a function of time
// ***************************************************************
{
   Graph ()
   // ***************************************************************
   // Graph: constructor class
   // ***************************************************************
        {
      setSize(450, 250);
   }

   public void paint ()
   // ***************************************************************
```

```
    // paint: paint the graph
    // ***************************************************************
            {
               clear();
               setColor("black");

               if(mcs*dt >= (xmax-1)) {        // set axes
                          xmin = t - (xmax-xmin-1);
                          xmax = xmin+5;              // range (xmax-xmin) is 5
               }
               drawAxes(xmin, xmax, -width, width);

               double xOld = xmin;
               int i, j;

               int points = (int) (mcs - xmin/dt + 0.5);  // number of line segments for plot
                                                     // 0.5 added to compensate for truncation

               for(j = 0, i = (int) (xmin/dt) % (maxT*2); j < points; j++, i = (i+1) % (maxT*2)) {
                          double x = xOld + dt;

                          for(int m = 0; m < sampleSize; m++) {
                            setColor(Colors[m%Colors.length]);
                            plotLine(xOld, sample[m][i], x, sample[m][(i+1)%(maxT*2)]);
                          }
                          xOld = x;
               }

               double y = -0.08 * (width * 2);
        setColor("black");
        plotStringCenter("time", xmax+0.3, y);  // changed
        plotStringCenter(Easy.format(xmin, 3), xmin - 0.2, y);  // added 0,5
        plotStringCenter(Easy.format(xmax, 3), xmax - 0.1, y);

               plotStringCenter(Easy.format(-width, 3), xmin - 0.2, -width);
               plotStringCenter(Easy.format(width, 3), xmin - 0.2, width - 0.1 * width);
               plotStringCenter("position", xmin, width + 0.1 * width);
            } // paint()
} // Graph()


class Output extends Plot
// ***************************************************************
// Output: Prints the real-time data
// ***************************************************************
{
    Output ()
    // ***************************************************************
    // Output: constructor class
    // ***************************************************************
            {
        setSize(250, 250);
    }

    public void paint ()
    // ***************************************************************
    // display the output stats
    // ***************************************************************
            {
        clear();
        setWindow(0, 200, 0, 3);//changed from 100 to 200
        setColor("black");
        boxArea(0, 200, 0, 3);// changed to 200
               setColor("orange");
               plotStringCenter("Current Stats", 100, 2.8);
        setColor("white");
               plotString("Time ......................... = " + Easy.format(t, 4), 5, 2.5);
        plotString("Trial Wave Function ...... = " + twfName, 5, 2.3);
        plotString("No of Walkers ............. = " + N, 5, 2.1);

               setColor("pink");
```

```
            plotString("Position mean ............. = " + Easy.format(WalkerStats.mean(), 4), 5, 1.7);
            plotString("Position stddev ............ = " +
                                Easy.format(WalkerStats.stddev(), 4), 5, 1.5);
            plotString("Position mean error ...... = " +
                                Easy.format(WalkerStats.meanError(), 4), 5, 1.3);
            plotString("Position mean % of error = " +
                          Easy.format(Math.abs(WalkerStats.meanError() / WalkerStats.mean()) * 100, 4),
                                5, 1.1);

            setColor("cyan");
        plotString("Energy ........................ = " +
                          Easy.format(Energy[mcs%(maxT*2)], 6), 5, 0.9);
            plotString("Energy mean ............... = " + Easy.format(EnergyStats.mean(), 4), 5, 0.7);
            plotString("Energy stddev ............. = " + Easy.format(EnergyStats.stddev(), 4),
                                5, 0.5);
            plotString("Energy mean error ........ = " + Easy.format(EnergyStats.meanError(), 4),
                                5, 0.3);
            plotString("Energy mean % of error   = " +
                          Easy.format(Math.abs(EnergyStats.meanError() / EnergyStats.mean()) * 100, 4), 5,
                                0.1);

    } // paint()
} // Output()


class NGraph extends Plot
// *****************************************************************
// NGraph: graph of total number of walkers as a function of time
// *****************************************************************
{
    NGraph ()
    // *****************************************************************
    // NGraph: class constructor
    // *****************************************************************
          {
        setSize(250, 100);
    } // NGraph()


    public void paint ()
    // *****************************************************************
    // paint: paints the graph
    // *****************************************************************
          {
            clear();
            setColor("black");

            if(mcs*dt >= (xmax-1)) {       // set x axis
                    xmin = t - (xmax-xmin-1);
                    xmax = xmin+5;            // range (xmax-xmin) is 5
            }

        double ymin = N-5;             // y axis range set relative to current
        double ymax = N+5;             // number of walkers
            drawAxes(xmin, xmax, ymin, ymax);

        setColor("green");

            double xOld = xmin;
            int i, j;

            int points = (int) (mcs - xmin/dt + 0.5);  // number of line segments for plot
                                             // 0.5 added to compensate for truncation

            for(j = 0, i = (int) (xmin/dt) % (maxT*2); j < points; j++, i = (i+1) % (maxT*2)) {
                    double x = xOld + dt;
                    plotLine(xOld, TotalWalkers[i], x, TotalWalkers[(i+1)%(maxT*2)]);
                    xOld = x;
            }
```

```
            double formaty = (-0.1 * (ymax - ymin))/3;

        // format the axes
        setColor("black");
        plotStringCenter(Easy.format(xmax, 3) + "t", xmax, ymin);

            plotStringCenter(Easy.format(ymin, 5), xmin + (xmax-xmin) * 0.15, ymin);
            plotStringCenter(Easy.format(ymax, 5), xmin + (xmax-xmin) * 0.15, ymax - 0.3);

            plotStringCenter("Total Walkers", xmin + (xmax-xmin) * 0.5, ymax - 0.3);

            plotStringCenter(Easy.format(Energy[mcs%(maxT*2)], 3), t, Energy[mcs%(maxT*2)] + 0.4);
    } // paint()

} // NGraph()



class EGraph extends Plot
// **************************************************************
// EGraph: graph of Energy as a function of time
// **************************************************************
{
    EGraph ()
    // **************************************************************
    // EGraph: class constructor
    // **************************************************************
            {
        setSize(450, 250);
    } // Egraph()


    public void paint ()
    // **************************************************************
    // paint: paints the energy graph
    // **************************************************************
            {
            clear();
            setColor("black");

            if(mcs*dt >= (xmax-1)) {        // set x axis
                    xmin = t - (xmax-xmin-1);
                    xmax = xmin+5;              // range (xmax-xmin) is 5
            }

        double ymin = -3;               // y axis range selected based on actual GSWF
        double ymax = 7;
            drawAxes(xmin, xmax, ymin, ymax);

        setColor("blue");

            double xOld = xmin;
            int i, j;

            int points = (int) (mcs - xmin/dt + 0.5);  // number of line segments for plot
                                              // 0.5 added to compensate for truncation

            for(j = 0, i = (int) (xmin/dt) % (maxT*2); j < points; j++, i = (i+1) % (maxT*2)) {
                    double x = xOld + dt;
                    setColor("blue");
      plotLine(xOld, Energy[i], x, Energy[(i+1)%(maxT*2)]);
                    xOld = x;
            }

        // format the axes
        setColor("black");
        plotStringCenter("time", xmax+0.3, -0.6);
        plotStringCenter(Easy.format(xmin, 3), xmin - 0.2, -0.3);
        plotStringCenter(Easy.format(xmax, 3), xmax - 0.1, -0.6);
```

```
            plotStringCenter(Easy.format(ymin, 3), xmin - 0.2, ymin);
            plotStringCenter(Easy.format(ymax, 3), xmin - 0.2, ymax - 0.3);
            plotStringCenter("Energy", xmin, ymax + 0.15);

    } // paint()

 } // Egraph()


class PsiGraph extends Plot
 // ****************************************************************
 // PsiGraph: Prints the wave function at time t
 // ****************************************************************
 {
    PsiGraph ()
    // ****************************************************************
    // class constructor
    // ****************************************************************
        {
      setSize(250, 150);
    }

    public void paint () {
      clear();

            double xmax = width + 0.25;
            double xmin = -width - 0.25;
            double ymax = (int) Math.sqrt(width) + 1;
            double ymin = -1;

            setColor("black");
            drawAxes(xmin, xmax, ymin, ymax);

            double xOld = -width;

            setColor("cyan");
            for(int i = 0; i < (width * 2)/ dt; i++) {
                    double x = xOld + dt;
        plotLine(xOld, awf.ExactGSWF(xOld, width), x, awf.ExactGSWF(x, width)); // added
                    xOld = x;
            }

            int sum = 0;
            int findmax = 0;
            for(int i = 0; i < numBins+2; i++) {
                    sum += psi[i];
                    if(psi[i] > findmax) {
                        findmax = (int) psi[i];
                    }
            }

            xOld = -width - binwidth;   // show value of x < MinRange
            double yOld = 0;
      setColor("magenta");
            for(int i = 0; i < numBins+2; i++) {
                    double x = xOld + binwidth;
                    double y = psi[i]/findmax;
                    plotLine(xOld, yOld, x, y);
                    xOld = x;
                    yOld = y;
            }

            double formaty = (-0.3 * (ymax - ymin))/3;
      plotString("current psi(x)", width * 0.3, ymax + formaty);

            setColor("cyan");
      plotString("actual psi(x)", -width, ymax + formaty);

            // format axes
      setColor("black");
      plotStringCenter(Easy.format(width, 3), width, formaty);
```

```
        plotStringCenter(Easy.format(-width, 3), -width, formaty);
                plotStringCenter("x", xmax * width, formaty);
                plotStringCenter(Easy.format(ymax, 3), -0.2 * width, ymax - 0.05);
                plotStringCenter(Easy.format(ymin, 3), -0.2 * width, ymin);
            }
}


// ***************** Start of main applet functions*********************


EGraph energygraph;
PsiGraph psig;
NGraph ngraph;

Graph graph;
Output output;
Reader N0Reader, widthReader, dtReader, binReader, nequilReader, skipReader, sampleReader;
Label twfLabel;
JComboBox twfReader;
int skip;

public void init ()
// ****************************************************************
// initialize the applet
// ****************************************************************
{
   initial();

          add(new JLabel("Visualizing Quatum Mechanics with Diffusion Monte Carlo: "));
          add(new JLabel("Particle in a Box with Infinite Walls in One Dimension"));

          JPanel StatPanel = new JPanel();
          StatPanel.setLayout(new GridLayout(0, 3, 2, 1));

          StatPanel.add(N0Reader = new Reader("No. of walkers ", N0));
          StatPanel.add(widthReader = new Reader("Box half width ", width, 4));
          StatPanel.add(dtReader = new Reader("Time step ", dt, 4));
          StatPanel.add(binReader = new Reader("No. of bins for psi(x) ", numBins));
          StatPanel.add(nequilReader = new Reader("Equilibration steps", nequil));
   StatPanel.add(skipReader = new Reader("Steps per block", skip));


          JPanel Stat2Panel = new JPanel();
          Stat2Panel.setLayout(new GridLayout(0, 4, 3, 0));

          Stat2Panel.add(sampleReader = new Reader("Sample size  ", sampleSize));
          String [] items = {"cosine", "polynomial"};
          twfReader = new JComboBox(items);
   twfReader.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         JComboBox cb = (JComboBox)e.getSource();
         twfName = (String)cb.getSelectedItem();
      }
   });
          twfLabel = new Label("Trial Wave Function");
          Stat2Panel.add(twfLabel);
          Stat2Panel.add(twfReader);

          add(StatPanel);
          add(Stat2Panel);
          addControlPanel();

          JPanel Stat3Panel = new JPanel();
          GridBagLayout gridbag = new GridBagLayout();
          GridBagConstraints c = new GridBagConstraints();
          Stat3Panel.setLayout(gridbag);
          c.fill = GridBagConstraints.HORIZONTAL;

          graph = new Graph();
          c.gridx = 0;
```

```
            c.gridy = 0;
            gridbag.setConstraints(graph, c);
            Stat3Panel.add(graph);

            output = new Output();
            c.gridx = 1;
            c.gridy = 0;
            c.insets = new Insets(0, 10, 0, 0);
            gridbag.setConstraints(output, c);
            Stat3Panel.add(output);


            JPanel Stat4Panel = new JPanel();
            gridbag = new GridBagLayout();
            c = new GridBagConstraints();
            Stat4Panel.setLayout(gridbag);
            c.fill = GridBagConstraints.HORIZONTAL;


            energygraph = new EGraph();
            c.weightx = 1;
            c.weighty = 1;
            c.gridheight = 2;
            c.gridx = 0;
            c.gridy = 0;
            gridbag.setConstraints(energygraph, c);
            Stat4Panel.add(energygraph);


            ngraph = new NGraph();
            c.gridheight = 1;
            c.gridx = 1;
            c.gridy = 0;
            c.insets = new Insets(0, 10, 0, 0);
            gridbag.setConstraints(ngraph, c);
            Stat4Panel.add(ngraph);


            psig = new PsiGraph();
            c.gridx = 1;
            c.gridy = 1;
            c.insets = new Insets(0, 10, 0, 0);
            gridbag.setConstraints(psig, c);
            Stat4Panel.add(psig);

            add(Stat3Panel);
            add(Stat4Panel);
}

public void step ()
// ***************************************************************
// take one step, which is determined by block size
// ***************************************************************
{
  for (int s = 0; s <= skip; s++) {
        MoveWalkers();
        GetEnergy(twfName);
    if (mcs == nequil)
          zeroData();
    else
          data();
  }

      if(mcs - nequil > 0)
        EnergyStats.AddData(Energy[mcs%(maxT*2)]); // store energy data

  graph.repaint();
  output.repaint();
      ngraph.repaint();
  energygraph.repaint();
      psig.repaint();
```

```
    } // step()


    public void reset ()
    // ****************************************************************
    // reset: resets the variables and clears the graphs
    // ****************************************************************
    {
        N0 = N0Reader.readInt();
        width = widthReader.readDouble();
        dt = dtReader.readDouble();
        nequil = nequilReader.readInt();
        skip = skipReader.readInt();
        if (skip < 0)
            skip = 0;

                numBins = binReader.readInt();
                sampleSize = sampleReader.readInt();


        initial();
        graph.repaint();
        output.repaint();
                ngraph.repaint();
                energygraph.repaint();
                psig.repaint();

    } // reset()


    PIB outer;                  // make it unix compatible

    public PIB ()
    // ****************************************************************
    // PIB: class constructor
    // ****************************************************************
    {
        super();                // create comphys.graphics.Animation
        outer = this;           // allows access to x[i] in inner classes
    } // PIB()


    public static void main (String[] args)
    // ****************************************************************
    // main function of the program
    // ****************************************************************
    {
        PIB pib = new PIB();
        pib.frame("Particle in a Box", 780, 680);
    } // main()

} // PIB
```