Diffusion Quantum Monte Carlo: A Java Based Simulation and Visualization

Ian Terrell

April 16, 2004

Abstract

After giving a brief background and theoretical discussion of the subject, this paper details the implementation of both a Diffusion Quantum Monte Carlo simulation of a one dimensional simple harmonic oscillator, and a graphical user interface to the simulation which can be used to explore the qualitative results of different parameters input to the system.

Contents

1 Background

1.1 Monte Carlo Simulation

Monte Carlo simulation is a class of numerical techniques that uses stochastic processes to solve mathematical problems that are either too difficult or impossible to solve either analytically or with other deterministic numerical techniques. Although the specifics of a Monte Carlo simulation vary greatly from application to application, all Monte Carlo simulations fit into a basic common structure.

A Monte Carlo simulation involves a large number of iterations of an algorithm that manipulates some set of data in a random fashion. From this data, an estimation $\hat{P}$ of a property P is made. Although the details of the governing statistical laws[1] are complicated, the basic idea is that in the limit as the number of iterations $n \to \infty$, the simulated result $\hat{P}$ converges to the actual empirical value of P. Given that it takes a large number of iterations involving a large quantity of random numbers to obtain an estimate of a property near its empirical value, Monte Carlo simulation is a task well suited for computers.

1.2 Quantum Monte Carlo Simulation

The fundamental relation in quantum mechanics is the Schrödinger Equation:

$$i\hbar\frac{\partial}{\partial t}\Psi = -\frac{\hbar^2}{2m}\nabla^2\Psi + V\Psi, \tag{1}$$

where Ψ is the wavefunction of the system and V is the potential of the system. This is a second order, complex, partial differential equation; it is impossible to solve for all but the simplest of systems. A few different Monte Carlo simulation techniques have been successfully applied to gain information from quantum systems by numerically manipulating mathematical features of the Schrödinger Equation. This project explores one of those methods, Diffusion Quantum Monte Carlo, which can be used to estimate the ground state and, in principle, excited state energies and wavefunctions of complex quantum systems[2].

1.3 Project Motivation and Description

Although the basic theory of Diffusion Monte Carlo is easy to learn, there are many nuances of the process that are less obvious. The primary purpose of this project is to construct a visual learning aid that will help anyone with a basic understanding of quantum mechanics to more quickly and thoroughly learn the Diffusion Monte Carlo process. To accomplish that goal, the project has two parts. First,

an example Diffusion Monte Carlo simulation system, based off of a tutorial article[3], is implemented and applied to calculate the ground state wavefunction and energy of a single particle in a one dimensional simple harmonic oscillator. Second, a graphical user interface to that simulation is implemented. This graphical user interface provides an easy method of inputting parameters to the simulation as well as provides a graphical representation of various measurements calculated by the simulation. Through this interface, a user can easily and quickly see the results of experimentation with different input parameters, gaining important qualitative knowledge and experience in the process.

2 Theory

The simulation implemented is based off of the algorithm described by Ioan Kosztin, Byron Faber, and Klaus Schulten; as such, the necessary theory behind Diffusion Monte Carlo simulation given here parallels the development of their paper. For a full discussion of the theory and their algorithm, the reader is strongly encouraged to view their original paper.[3]

2.1 Relevant Quantum Mechanics

If the potential V in the Hamiltonian ($H = -\frac{\hbar^2}{2m}\nabla^2 + V$) is time independent, then the time dependence of Eq. (1) can be separated. The solutions to the time independent Schrödinger equation are found by solving the equation

$$H\phi_n = E_n\phi_n \tag{2}$$

where E_n are the *eigenenergies* of the *eigenfunctions* ϕ_n, ordered as $E_0 < E_1 \leq E_2 \ldots$. The time dependent eigenfunctions are then found (by solving the separated ordinary differential equation for time) to be

$$\psi_n = \phi_n e^{-\frac{iE_n t}{\hbar}}. \tag{3}$$

Any wavefunction can be written as a linear combination of eigenfunctions in a particular basis as

$$\Psi = \sum_{n=0}^{\infty} c_n\psi_n = \sum_{n=0}^{\infty} c_n\phi_n e^{-\frac{iE_n t}{\hbar}}. \tag{4}$$

The Diffusion Monte Carlo process critically depends on two transformations being made to Eq. (4). The first is a simple shift of the

energy scale by a constant E_R, the *reference energy*. This shift transforms $E_n \to E_n - E_R$. The second transformation is less trivial, and involves the introduction of *imaginary time*, $\tau = it$.

These two changes transforms Eq. (4) to

$$\Psi = \sum_{n=0}^{\infty} c_n \phi_n e^{-\frac{(E_n - E_R)\tau}{\hbar}}. \tag{5}$$

If $E_R = E_0$ we see that

$$\Psi = c_0\phi_0 + \sum_{n=1}^{\infty} c_n \phi_n e^{-\frac{(E_n - E_0)\tau}{\hbar}}, \tag{6}$$

from which we notice that, assuming τ is real,

$$\lim_{\tau \to \infty} \Psi = c_0\phi_0. \tag{7}$$

2.1.1 Simple Harmonic Oscillator

The only potential looked at in the project is the potential of the simple harmonic oscillator, which is given as

$$V(x) = \frac{1}{2} m\omega^2 x^2. \tag{8}$$

This is a well understood, solved problem, with the ground state energy

$$E_0 = \frac{1}{2}\hbar\omega \tag{9}$$

and the ground state wavefunction

$$\phi_0(x) = \left(\frac{m\omega}{\hbar\pi}\right)^{\frac{1}{4}} e^{-\frac{m\omega x^2}{2\hbar}}. \tag{10}$$

2.2 Diffusion Monte Carlo Algorithm

2.2.1 Overview

The Diffusion Monte Carlo algorithm can be summarized as follows: an unnormalized wavefunction Ψ is represented numerically as a distribution of *walkers*, which can be though of as points in space having a definite position. These walkers are propagated through imaginary time, dying off or being created at each iteration depending on their position, their potential energy, and the average potential of all of the walkers, which is interpreted as the energy of the system. The initially incorrect reference energy is constantly adjusted to keep an approximately constant number of walkers. After a large number of

iterations, the average of many successive measurements of E_R will approach the actual E_0 of the system, and the distribution of the superposition of the walkers of many successive iterations will converge to the ground state wavefunction ϕ_0 by virtue of Eq. (7).

2.2.2 Algorithm

The algorithm to manipulate the walkers and obtain the ground state energy and wavefunction is as follows:

1. At the start of the simulation,
   (a) N_i initial walkers are distributed in space to make up some initial wavefunction Ψ.
   (b) the reference energy is set to be the average potential of the walkers.
2. At each iteration of the simulation,
   (a) the time τ is advanced by a small $\Delta\tau$.
   (b) the walkers' positions are changed by a random variate from a Gaussian distribution with mean 0 and standard deviation

$$\sigma = \sqrt{\frac{\hbar\Delta\tau}{m}}. \tag{11}$$

   (c) the updated reference energy is calculated from the newly arranged walkers by the function

$$E_R = \langle V \rangle + \alpha\left(1 - \frac{N}{N_i}\right) \tag{12}$$

   where $\langle V \rangle$ is the average potential of all of the walkers, and N is the current number of walkers.
   (d) for each walker a number m is calculated by

$$m = \lfloor W(x) + U \rfloor \tag{13}$$

   where U is a random number uniformly distributed from 0 to 1, and $W(x)$ is the *weight* of the walker with position x, defined as

$$W(x) = e^{-\frac{(V(x)-E_R)\Delta\tau}{\hbar}}. \tag{14}$$

   If $m = 0$, the walker is destroyed. If $m = 1$, nothing is done to the walker. If $m > 1$, $m - 1$ copies of the walker are made. To prevent uncontrolled growth, a maximum of two copies of any given walker are made at each iteration.

3. To estimate the ground state energy, the reference energies of successive iterations are averaged. To estimate the ground state wavefunction, the spacial distribution (histogram) of the walkers of successive iterations is recorded and normalized.

The specifics of the algorithm, especially Eq. (11-14), are derived from the Feynman path integral solution of the Schrödinger equation. Their derivations can be found in the article by Kosztin, et al[3].

2.2.3 Dimensionless Units

To implement the algorithm in a standard programming language, the relevant physical quantities must be converted to dimensionless units. To do this, each physical quantity is expressed as a product of its magnitude and a unit (L, T and $\mathcal{E}$ for length, time, and energy). Then, each of the units are set to values that are both convenient with respect to the potential, and also satisfy the relations

$$\frac{\hbar T}{2mL^2} = \frac{1}{2} \tag{15}$$

and

$$\frac{T\mathcal{E}}{\hbar} = 1. \tag{16}$$

Using $T = 1/\omega$ for the simple harmonic oscillator potential, we are able to rewrite the necessary equations in convenient dimensionless units.[3]

The potential (Eq. (8)) becomes

$$V(x) = \frac{1}{2}x^2; \tag{17}$$

the ground state energy of the system (Eq. (9)) becomes simply

$$E_0 = \frac{1}{2}; \tag{18}$$

the ground state wavefunction (Eq. (10)) becomes

$$\phi_0(x) = \pi^{-\frac{1}{4}} e^{-\frac{x^2}{2}}; \tag{19}$$

the standard deviation of the Gaussian random variate used to displace the walkers (Eq. (11)) becomes

$$\sigma = \sqrt{\Delta\tau}; \tag{20}$$

and finally, the weight function of the walkers (Eq. (14)) becomes

$$W(x) = e^{-(V(x)-E_R)}. \tag{21}$$

3 Project

3.1 The Simulation

Steps 1 and 2 of the Diffusion Monte Carlo algorithm given in Section 2.2.2 were implemented with modularized classes in Java[4]. Since Step 3 of the algorithm is just data collection, it is left up to the user of the software to implement. The implemented package `dmc` includes the following classes:

- The `Walker` class contains the position of each walker.
- The `DMC` class contains all of the necessary functions to implement the Diffusion Monte Carlo algorithm given.
  - `DMC(...)` - This constructor implements Step 1 of the algorithm. It initializes the simulation, by setting the values of all of the variables necessary in the simulation to the values passed. It initializes the walkers' positions either with a delta function $\delta(x - x_0)$, or randomly according to either a Uniform(a,b) distribution or a Gaussian(μ,σ) distribution. It also provides a mean to override the default behavior of having the reference energy be the average potential at the locations of all of the walkers, and provides a mean to use the suggested[3] dimensionless feedback parameter $\alpha = 1/\Delta\tau$.
  - `V(double x)` - This function returns the potential at the location x. It is meant to be overwritten by derived classes.
  - `W(Walker w)` - This function computes the weight of the walker w as given by Eq. (21).
  - `Iterate()` - This function does one complete iteration of the simulation, including updating the imaginary time variable τ (Step 2.a), calling `walk()` and `branch()`.
  - `walk()` - This function moves all of the walkers by a random number from a Gaussian(0,σ) distribution, where σ is given by Eq. (20) (Step 2.b). To keep computation time down, as each walker is moved its potential is recorded, and before returning this function updates the reference energy of the system (Step 2.c).
  - `branch()` - This function branches the walkers as described in Step 2.d of the algorithm, calculating m from Eq. (13) for each walker, and creating or destroying the walkers as needed.
- The `DMC_SHO` class extends the `DMC` class and overrides the function `V(double x)` to return the potential for the simple harmonic oscillator as given in Eq. (17).

3.2 The Visualization

A complete graphical user interface to the simulation was written in Java using its *Advanced Windowing Toolkit* and *Swing* advanced programming interfaces. It provides graphical means to input parameters to the simulation and display various measurements output by the simulation.

The main class `GUI` contains almost all of the standard user interface setup code, which creates all of the labels, text boxes, etc, and handles events such as the user pressing a button. In addition, the class

- provides mechanisms to start, pause, continue, and reset the simulation.
- provides methods to input the initial number of walkers (N_i from Step 1.a of the algorithm), the timestep ($\Delta\tau$ from Step 2.a), the feedback parameter (α in Eq. (12), the initial reference energy, and the seed for the random number generator for the simulation.
- provides a mechanism to select the desired method of initializing the walkers, including parameter input.
- provides graphs to display the current number of walkers, the current reference energy, a histogram of the walkers' position, the current estimate of the ground state energy, and the current estimate of the ground state wavefunction.
- provides various graphing options, including the ability to change the colors of the graphs, display data as either discreet points or connected lines, choose whether or not to graph the theoretical values, how many points to display at once, etc.
- provides special features such as the ability to keep the reference energy constant, stop collecting E_0 estimation data after a certain number of iterations, and only start collecting the ϕ_0 estimation data after a certain warmup period.
- provides the ability to manipulate how fast the simulation runs.

3.2.1 Graphs

In order to display the data graphically, several graphing classes had to be written. These were all derived from a freely distributed `Graph` class[5].

- `Histogram` - This class implements a histogram graph to display the distribution of walkers at any given iteration. The data can be displayed as either bars or points.

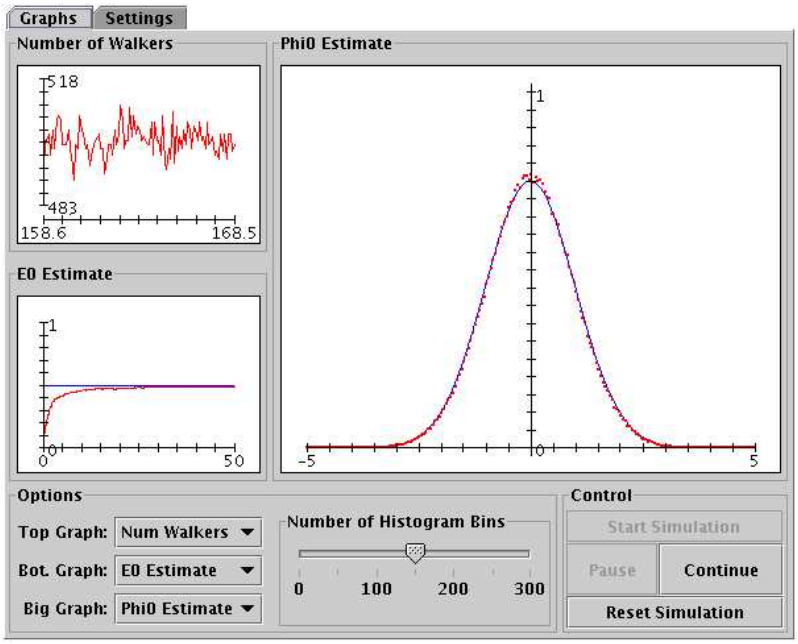

Figure 1: A screenshot of the visualization graphing the current number of walkers, and the estimates of E_0 and ϕ_0 after they had mostly converged to their theoretical values.

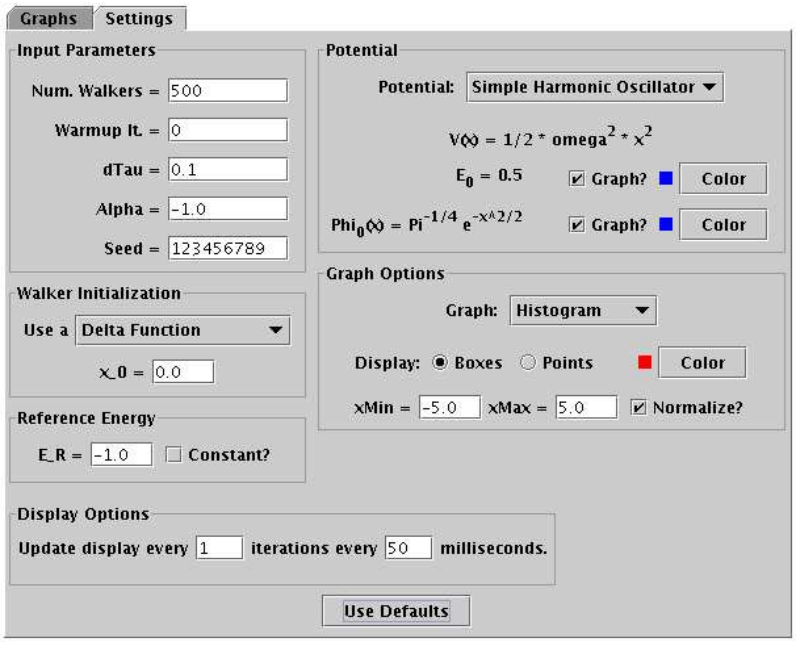

Figure 2: A screenshot of the settings panel of the visualization.

- `DataGraph` - This class implements a graph to display scatterplot data as either discrete points or connected lines. It uses the `GraphData` class to store data.
- `Phi0Histogram` - This class implements a specialized histogram to display the estimate of ϕ_0.
- `Function` - This class is meant to be used as a superclass, and provides a method to draw a function on a graph. It is used to draw the theoretical values on the graphs.

3.3 Project Webpage

A webpage[6] was created to act as a portal for information about the project. For developers it provides links to download the complete source code and view the JavaDoc for the source code. For users, it displays the visualization, as well as provides a tutorial describing how to use the visualization, and contains a list of frequently asked questions to troubleshoot problems.

3.4 Further Work

3.4.1 User Interface Design

The underlying design of the graphical user interface could use some modularization before the parts can be used easily in other graphical simulation programs. Specifically, it would be useful to

- modularize all graphical aspects of each type of graph into another class, and get the graph specific members out of the high up GUI.
- modularize all aspects of each potential into another class, etc.
- Remove all unnecessary data members, perhaps through using action commands in the event handling instead of the event source.
- Modularize the event handling into multiple classes that each handle events from a certain type of object.

3.4.2 Simulation

The simulation package written is very general, and will work with any well behaved one dimensional potential. Once the graphical design is more modularized, it would be easy to add potentials to the visualization.

The simulation could also be easily extended to work in multiple dimensions with multiple particles, or to utilize additional Diffusion Monte Carlo algorithms, such as importance sampling.

References

[1] Namely, the *Central Limit Theorem*. For an analytically thorough discussion of the theorem as it pertains to simulation, see *Simulation Modeling and Analysis*, Third Edition, by Averill M. Law and W. David Kelton, McGraw-Hill Higher Education, 2000, page 254.

[2] Peter J. Reynolds, Jan Tobochnik and Harvey Gould, "Diffusion Quantum Monte Carlo," *Computers in Physics*, Nov/Dec 1990, Volume 4, Issue 6, pp. 662-668.

[3] Ioan Kosztin, Byron Faber and Klaus Schulten, "Introduction to the Diffusion Monte Carlo Method," *American Journal of Physics*, May 1996, Volume 64, Issue 5, pp. 633-644.

[4] The source code is thoroughly documented with JavaDoc style comments, and although the source code is fairly concise, the comments swell the files to over 400 lines for the simulation and about 3000 lines for the graphical user interface. For this reason, the source is not included here, but is freely available through the project webpage[6].

[5] The `Graph` class used is freely available from the University of Alabama in Huntsville at `http://www.math.uah.edu/psol/objects/edu/uah/math/devices/Graph.html`

[6] `http://merlin.physics.wm.edu/~ian/dmcview/`