Visualization of Diffusion Monte Carlo

Huy Nguyen*[1] and Shiwei Zhang†[2]

[1]Department of Physics, Reed College, Portland OR 97202
[2]Department of Physics, College of William & Mary, Williamsburg, VA 23185

August 8, 2012

Abstract

After giving background and theoretical discussion of the Diffusion Monte Carlo method, this paper details its implementation as described in Kosztin *et al.*, Am. J. Phys. **64**, 633 (1996) and a graphical user interface to illustrate the method.

Supported by the National Science Foundation under Grant No. PHY-1156997 and by Reed College under the Summer Experience Award.

1 Introduction

Diffusion Monte Carlo (DMC) is one of many quantum Monte Carlo (QMC) algorithms, a class of algorithms that use stochastic processes to sample the target distribution (ground state wave function for ground state methods or the density matrix for finite-temperature methods) in order to evaluate expectation values of observables using Monte Carlo integration. They are attractive because of polynomial scaling with system size, in contrast to deterministic methods (e.g. exact diagonalization) whose dimensionality grows exponentially with system size. However, the stochastic nature of these algorithms can be challenging to beginners. This project aims to create visualization software to help users familiarize with these methods, develop intuition about them and explore the effect of various parameters on the simulation.

2 Theory

2.1 Markov chain Monte Carlo (MCMC)

Monte Carlo allows one to evaluate integrals like $I = \int f(x)g(x)\,\mathrm{d}x$ where x can be a scalar or a vector in many-dimensional space and the integral is performed over some part of a predefined state space. If $f(x)$ can be treated like a probability distribution function (pdf), meaning $f(x)$ is everywhere non-negative and normalized to 1 then we can sample M points x_i from the distribution

*me@huy.dev
†shiwei@wm.edu (advisor)

f and compute $g(x_i)$ for each x_i. The sample mean of these outputs is the Monte Carlo estimate for the integral:

$$\int \mathrm{d}x\, f(x)g(x) \approx \frac{1}{M}\sum_{i=1}^{M} g(x_i) \tag{1}$$

This estimate is unbiased and consistent, unbiased meaning the expectation value of the estimate equals the true value of the integral and consistent meaning the estimate converges with probability one to the true value as the number of sampled points tends to infinity. Note that although the asymptotic behavior (at infinity) is guaranteed, there is no guarantee for the rate of convergence. The error of the estimate goes down with the number of (independently) sampled points as $\frac{1}{\sqrt{N}}$, independent of the number of dimensions of the integral. Hence we usually say that Monte Carlo integration avoids the so-called "curse of dimensionality" because traditional numerical quadrature requires exponentially more points in order to maintain accuracy.

Sometimes we do not know the explicit form of the distribution f but we know its integral form:

$$f(y) = \int \mathrm{d}x\, f(x)G(x \to y) \tag{2}$$

where $G(x \to y)$ is a transition probability (also known as a Green's function) that describe the probability of moving from the state x to the state y and G is normalized: $\int \mathrm{d}y\, G(x \to y) = 1$. Probability theory says that the distribution f is the stationary distribution (equilibrium state) of a Markov chain that has $G(x \to y)$ as the transition probability in the limit of infinitely many iterations.

2.2 Imaginary-time Schrodinger equation

This section and the next outline the theoretical basis for the simple version of Diffusion Monte Carlo. It will mostly be a summary since the theory has been discussed extensively in the literature. Here we roughly follow the discussion as laid out in Kosztin *et al.* [1]. The wave function $\Psi(x,t)$ of a single particle of mass m moving along in a potential $V(x)$ is described by the Schrodinger equation:

$$\mathrm{i}\hbar\frac{\partial\Psi}{\partial t} = \hat{H}\Psi \tag{3}$$

where the Hamiltonian $\hat{H} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x)$ is simply the sum of the kinetic and potential energy. Since the eigenbasis of the Hamiltonian is complete and orthonormal any solution can be expressed in the form

$$\Psi(x,t) = \sum_{n=0}^{\infty} c_n\phi_n(x)\mathrm{e}^{-\frac{\mathrm{i}}{\hbar}E_n t} \tag{4}$$

where the coefficients c_n. With the benefit of hindsight we introduce a normalization factor through a shift of energy scale by a reference energy E_R

$$\mathrm{i}\hbar\frac{\partial\Psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\Psi}{\partial x^2} + \left[V(x) - E_\mathrm{R}\right]\Psi \tag{5}$$

To convert the oscillatory behavior of the wavefunction over time to exponential behavior, we perform a Wick rotation of time (also known as imaginary time) by substituting $\mathrm{i}t = \tau$.

$$\hbar\frac{\partial\Psi}{\partial\tau} = \frac{\hbar^2}{2m}\frac{\partial^2\Psi}{\partial x^2} - \left[V(x) - E_\mathrm{R}\right]\Psi \tag{6}$$

This new equation has solution

$$\Psi(x,\tau) = \sum_{n=0}^{\infty} c_n \phi_n(x) \mathrm{e}^{-\frac{E_n - E_\mathrm{R}}{\hbar}\tau} \tag{7}$$

If we choose E_R to be the same as the ground state energy, $\lim_{\tau\to\infty} \Psi(x,\tau) = c_0\phi_0(x)$, meaning the wave function converges to the ground state regardless of the initial wave function Ψ (of course assuming that Ψ is not orthogonal to ϕ_0).

2.3 Feynman's path integral

There are many ways to arrive at an algorithm for Diffusion Monte Carlo. One of them is by comparing Eq. (6) with the classical diffusion equation for a concentration ϕ, recognizing that the kinetic and potential terms play the same role as that of the diffusion (D) and local reaction term ($R(x)$), respectively.

$$\frac{\partial \phi}{\partial t} = D\frac{\partial^2 \phi}{\partial x^2} + R(x)\phi \tag{8}$$

However, to see a random walk more clearly, a better formulation would be in terms of the Feynman path integral. The solution of Eq. (6) can be expressed as

$$\Psi(x, t+\tau) = \int_{-\infty}^{\infty} \mathrm{d}x_0 \, K(x_0, 0 \to x, \tau) \tag{9}$$

where K is the usual quantum mechanical propagator after many instances of the completeness relation have been inserted:

$$K(x_0, 0 \to x, \tau) = \lim_{N\to\infty} \int \mathrm{d}x_1 \cdots \int \mathrm{d}x_{N-1} \left(\frac{m}{2\pi\hbar\Delta\tau}\right) \exp\left\{-\frac{\Delta\tau}{\hbar}\sum_{j=1}^{N}\left[\frac{m}{2\Delta\tau^2}(x_j - x_{j-1})^2 + V(x) - E_\mathrm{R}\right]\right\} \tag{10}$$

Here $\Delta\tau = \frac{\tau}{N}$ is a small time step. We need to take the limit $N \to \infty$ so as to minimize the time-step error in the Trotter-Suzuki expansion:

$$\mathrm{e}^{-\Delta\tau\hat{H}} = \mathrm{e}^{-\Delta\tau\hat{K}}\mathrm{e}^{-\Delta\tau(\hat{V} - E_\mathrm{R})} + O\left(\Delta\tau^2\right) \tag{11}$$

We used this formula to approximate the time evolution operator $e^{-\Delta\tau\hat{H}}$ which we don't know in terms of the kinetic and potential propagators whose forms we do know. To make the following discussion tidier, Eq. (9) can be written more succinctly as

$$I = \Psi(x,\tau) = \lim_{N\to\infty} \int_{-\infty}^{\infty} \left(\prod_{j=0}^{N-1} \mathrm{d}x_j\right) \left(\prod_{n=1}^{N} W(x_n) P(x_n, x_{n-1})\right) \Psi(x_0, 0) \tag{12}$$

where we have defined

$$P(x_n, x_{n-1}) = \left(\frac{m}{2\pi\hbar\Delta\tau}\right)^{1/2} \exp\left[-\frac{m(x_n - x_{n-1})^2}{2\hbar\Delta\tau}\right] \tag{13}$$

and

$$W(x_n) = \exp\left[-\frac{V(x_n) - E_\mathrm{R}}{\hbar}\Delta\tau\right] \tag{14}$$

An attentive reader would have noticed that $P(x_n, x_{n-1})$ is simply a normal probability distribution (and hence non-negative and normalizable) centered at x_{n-1} with standard deviation $\sigma = \sqrt{\frac{\hbar\Delta\tau}{m}}$. On the other hand, $W(x_n)$ is not normalized and cannot be interpreted as a probability distribution. If we identify $P(x_n, x_{n-1})$ and $W(x_n)$ in Eq. (12) with $f(x)$ and $g(x)$ in Eq. (1) respectively, we can approximate the path integral by Monte Carlo

$$I = \frac{1}{M}\sum_{i=1}^{M} f(x^{(i)}) \tag{15}$$

where $f(x^{(i)}) = \prod_{i=1}^{N} W(x_i)$ and $x^{(i)}$ is a vector (that is theoretically infinitely long but in practice has a finite but large number of components) sampled from the distribution $\prod_{i=1}^{N} P(x_n, x_{n-1})$. To sample the vectors $x^{(i)}$ we start out with some x_0 as the first component of the vector and then successively generate a sequence of vector components $x_1, x_2, \ldots, x_{N-1}$ such that any one component x_n is related to previous one x_{n-1} by a normal distribution centered at x_n with standard deviation $\sigma = \sqrt{\frac{\hbar\Delta\tau}{m}}$. This is in effect a random walk (Markov chain). It is easy to verify that the resultant vector $(x_0, x_1, \ldots, x_{N-1})$ sampled this way is in fact distributed according to $\prod_{i=1}^{N} P(x_n, x_{n-1})$.

The calculation of function f merits further discussion. It is simply a product of the weights $W(x_n)$ accumulated at points visited by the random walk. If we take Eq. (15) literally, we would calculate f by simply multiplying at each step the new weight found in that step with the products of all existing weights. However, if the weights for a walker are on average greater than one then the value of f for that walker will keep increasing and becomes very large by the end of the walk. Conversely, if the weights are on average less than one then f will be very small. The downside of this approach is that although it is the points with large values of f that contribute the most to the Monte Carlo estimate (and points with small f contributing very little), we are spending the same amount of computational effort to propagate the walkers with small f as those with large f. This is inefficient. Furthermore, the Monte Carlo estimate will be dominated by a few walkers with very large weights, implying a tremendous loss of statistical information.

We instead use a branching process. At every step after computing the weight $W(x_n)$ we replace each walker by

$$m = \lfloor W(x_n) + 3 \rfloor \tag{16}$$

walker at the same position. If $m = 0$ we simply kill the walker. This process on average replaces one walker by $W(x_n)$ walkers (we do need the integer m as the number of walkers to be added because in the implementation it is not possible to add fractional walkers). To avoid the possible explosion of population where the potential diverges (becomes infinitely large), we only allow m to be as large as 3 at the expense of incurring a population control bias. Furthermore, this interpretation of the weights allows a new method of propagating the walkers. Instead of propagating one walker at a time, we propagate the whole ensemble of walkers but only keeping information about the current population. This approach has a further advantage: since we have ready access to the all the walkers in the current time step we can update the reference energy at every step to follow the evolution of the system (which also keep the fluctuations in the walker population to a minimum).

2.4 Importance sampling

Importance sampling is a very general and useful technique in Monte Carlo integration. Its goal is to replace a given sampling distribution with a different distribution that is more efficient and gives

a lower variance to the estimate without changing the expectation value. For example, instead of the interpretation

$$I = \int \mathrm{d}x\, f(x)g(x) \approx \frac{1}{M}\sum_{i=1}^{M} f(x_i), \quad x_i \sim f(x) \tag{17}$$

we can multiply and divide the integrand by a function $h(x)$ to obtain

$$I = \int \mathrm{d}x \left[\frac{f(x)g(x)}{h(x)}\right] h(x) \approx \frac{1}{M}\sum_{i=1}^{M} \frac{f(x_i)g(x_i)}{h(x_i)}, \quad x_i \sim h(x) \tag{18}$$

The advantage of this method is that if we can make $h(x)$ have similar form to $f(x)g(x)$ then $h(x) \approx f(x)g(x)$ and the ratio $\frac{f(x_i)g(x_i)}{h(x_i)}$ is approximately constant. This will decrease the variance due to less fluctuation in the function being evaluated.

3 Diffusion Monte Carlo algorithm

1. Before the simulation begins, N_i walkers are distributed according to the initial wave function Ψ and the initial value of the reference energy E_R is set to be the average potential energy of all walkers.

2. For each step of the simulation

   - The simulation time is advanced by the time step $\Delta\tau$.
   - The position of each walker is changed by a random number whose mean is 0 and standard deviation is $\frac{\hbar\Delta\tau}{m}$. If importance sampling is used, in addition to the random diffusion each walker also experiences a "drift force" or "quantum force" which in the case of the harmonic potential with a Gaussian trial wave function $\mathrm{e}^{-\alpha x^2}$ is given by $\frac{1}{2}\Delta\tau(-4\alpha x)$.
   - The reference energy is updated as

     $$E_{\mathrm{R}} = \langle V\rangle + \alpha\left(1 - \frac{N}{N_i}\right) \tag{19}$$

     where $\langle V\rangle$ is the average potential of all walkers and N is the current number of walkers. If importance sampling is used, the local energy is also calculated

     $$E_{\mathrm{L}}(x) = \frac{\hat{H}\Psi}{\Psi} = \alpha + x^2\left(\frac{1}{2} - 2\alpha^2\right) \tag{20}$$

   - For each walker a weight

     $$W(x) = \mathrm{e}^{-\frac{(V(x)-E_{\mathrm{R}})\Delta\tau}{\hbar}} \tag{21}$$

     and a number m

     $$m = \lfloor W(x) + \eta \rfloor \tag{22}$$

     are calculated where η is a random uniform number on the interval [0,1]. Then $m-1$ new walkers are created. If $m = 0$ the walker is removed from the ensemble. To prevent population explosion when the potential differs greatly from the reference energy, m is restricted to be at most 3. If importance sampling is used, the weight has a difference expression $W(x) = \mathrm{e}^{-(E_L - E_{\mathrm{R}})\Delta\tau}$.

3. The estimate for the ground state energy is found by averaging the successive reference energies while the ground state wave function is found from the normalized spatial distribution of the walkers. If importance sampling is used then the energy estimate is instead found by averaging the successive local energies.

4 Diffusion Monte Carlo package

This JAVA application [1] builds on the work done by Ian Terrell, an undergraduate at the College of William and Mary for his senior thesis [2]. The application consists of the Graphical User Interface, or GUI (`GUI.java`); the classes that actually carry out the simulation in the background and feed data into the GUI (all classes in the package `DMC`); and supporting classes (e.g. plotting histograms, generating random numbers).

4.1 The simulation

1. The `Walker` class contains the properties of a walker.

   - The field `x` contains the current position and `weight` the current weight of the walker. `drift` is the magnitude of the drift force that the walker experiences if an importance sampling scheme is used. These fields are essential to the simulation.
   - To facilitate the visualization, a number of extra fields are needed. `count` is a static variable that increments every time a walker is created so as to give each new walker a unique identification which is stored in the field `id`. This `id` is used in the GUI to match walkers across different time steps in order to trace out their trajectories over time. `WalkerColor` is the color of the walker that will show up in the visualization.
   - A walker's color is picked from JAVA's built-in color constants (red, blue, green etc) which are stored in the array `colors`. Function `assignColor()` cycles through the available colors and assign them to walkers to minimize identical colors among current walkers.
   - The constructor `Walker(x)` makes a new walker with position `x` with automatically assigned `id` and `color`.

2. The `DMC` class is the base class to the Diffusion Monte Carlo simulation and will never be called directly in a simulation. Rather, other classes will inherit it and implement their own version of certain functions, e.g. the potential. However, these functions still need to be present in the base class in order to provide a common interface.

   - The `DMC()` constructor takes in essential parameters (e.g. the imaginary time step `dTau`, the `seed` for the random number generator etc) and creates an instance of the simulation. It creates an array of walkers with the given type of distribution (specified by the argument `initMode` which can be δ function, uniform or Gaussian distributions) with associated attributes (e.g. Gaussian distribution needs the mean and the standard deviation) which are given by the two parameters `param1` and `param2`. It also calculates the energy of the initial walker population.

[1] Available at `https://www.huy.dev/DMC/`

- Function `V(x)` returns the potential energy at position `x`. It is meant to be overridden by specific potential functions in child classes, say, the harmonic potential.
- Function `Weight(x)` returns the weight at the current position as $e^{-[V(x)-E_R]\Delta\tau}$. Classes that implements the simple version of DMC will not override this function. However, those that use importance sampling will need to redefine this in order to use their own version of the weight e.g. $e^{-[E_L-E_R]\Delta\tau}$.
- Function `branch()` carries out the birth/death process. For each walker it takes the weight in order to compute the number $\mathtt{m} = \lfloor \mathtt{weight} + \eta \rfloor$ where η is a uniform random number between 0 and 1 and then creates (`m` -1) new copies of the walker. If $\mathtt{m} = 0$, the walker is killed. It is easy to verify that this process on average produces `weight` walkers from one original walker.
- Function `acceptOrReject(oldPosition, new Position)` decides whether a proposed move should be accepted (true) or rejected (false). It is only used in classes that need importance sampling and hence always trivially returns true in the base class. Along the same line of usage, function `drift(x)` returns the drift force and `localEnergy(x)` returns the local energy but they are not needed in classes that do not use importance sampling.
- Function `walk()` performs the diffusion. It moves each walker to a new position and updates the reference energy.
- Function `Iterate()` is like a "wrapper" function to drive the simulation. It invokes `walk()` and `branch()` to carry out one step of the random walk. If the GUI requires visualization of individual walkers, at the beginning of every step it will "back up" previous ensembles of walkers in order to trace out the trajectories of walkers over time. It stores these walkers in a queue so that only a maximum of six ensembles are remembered at any one time, the reason being the GUI can only show six ensembles at once. However, as of the time this code was written I have not found a way to implement an array of arrays which prevented me from using JAVA's built-in queue, hence the "primitive" implementation of a queue seen in the code. I plan to work on this in a future version. If a Metropolis accept/reject step is used this function will also adjust the time step as discussed later.

3. Class `DMC_SHO` is a child class of `DMC` and implements the DMC simulation for the harmonic oscillator by simply overriding the potential function `V(x)` with $V = \frac{1}{2}x^2$.

4. Class `DMC_SHO_Gaussian` is a child class of `DMC` and implements the importance-sampled version of DMC for the harmonic oscillator by overriding the base class' functions `V`, `drift`, `localEnergy` and `weight` with the appropriate expressions.

   - Function `acceptOrReject` performs a Metropolis accept/reject step given the proposed move to `newPosition` from `oldPosition`. It returns true to indicate acceptance and false to indicate rejection.
   - As a result of the possible rejection of a move, the effective time step will be lower and hence must be adjusted accordingly in function `walk()`. We chose to modify it by multiplying the current time step by the current ratio of acceptance (i.e. how many moves are accepted out of the number of proposed moves). However, this is problematic

because as the random walk approaches equilibrium, the walkers will tend to stay in regions where the wave function has large amplitude, making proposed moves less likely to be rejected and suggesting that the time step should be kept constant. However, at this time the acceptance ratio is always less than one (due to the inevitable rejections at the beginning of the walk), implying the time step will steadily decrease as the walk proceeds because we keep multiplying the current time step by the acceptance ratio. Although we intentionally choose a Metropolis scheme that makes the acceptance probability close to one, we could do better by keeping track of the most recent accept/reject outcomes and calculate the acceptance ratio base one those 10 instead of the entire history of the walk. Alternatively, we can do away with this adjustment in the measurement phase and leave it up to the user to determine how long the thermalization (and hence the adjustment of the time step) process should take.

4.2 The visualization

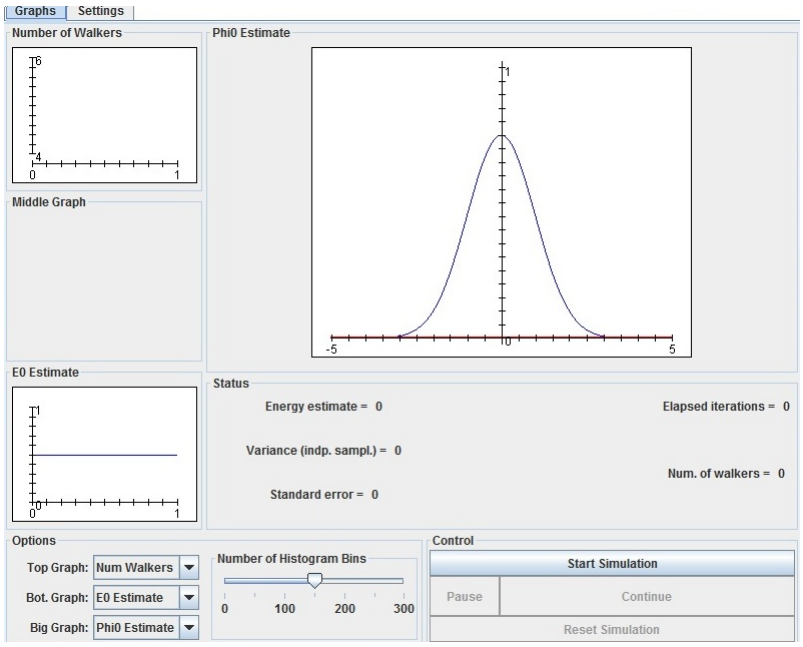

Figure 1: The `Graph` tab of the interface shows graphs of various parameters of a simulation.

The graphical interface as shown in Figs. 1 and 2 is written in JAVA using the Swing and Abstract Window Toolkit packages that come with all standard JAVA distributions. It allows the user to choose most of the parameters of a simulation and shows graphically many measurement outputs. The GUI allows the user to

- Start, pause, continue reset the simulation and assign defaults value to the parameters.
- Set the input parameters: the number of walkers, the time step, the feedback parameter, the seed for the random number generator and the number of steps in the thermalization phase.

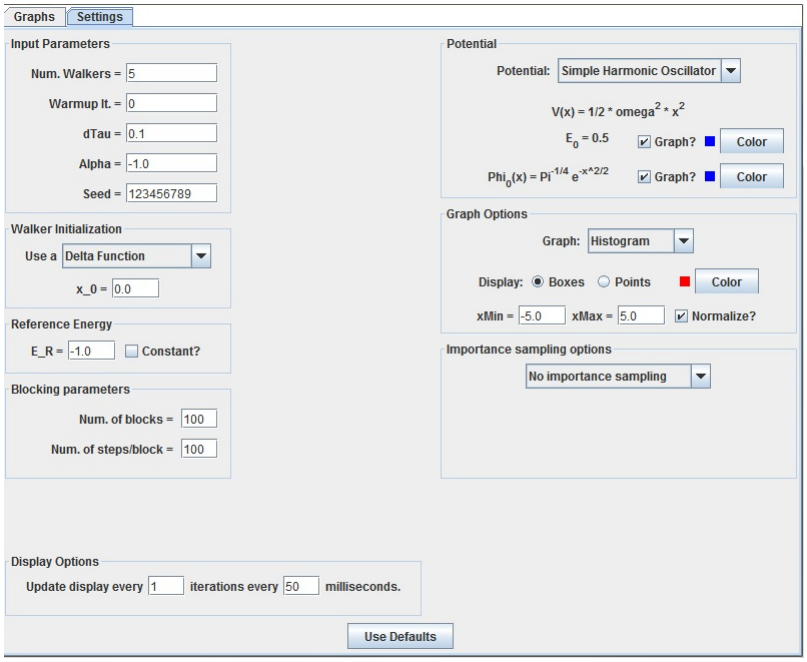

Figure 2: The `Settings` tab of the interface allows users to control all parameters of a simulation.

- Set the initial distribution of walkers: the uniform distribution with the upper and lower limits, the Gaussian distribution with the mean and standard deviation and the Dirac delta function with the position of the peak.
- Set reference energy and choose whether or not to keep it constant throughout the walk. Keeping it constant allows the user to see the role the reference energy plays in keeping the walker population under control.
- Choose a potential. Right now there are two options: the simple harmonic potential and the particle in a box. The user can also choose whether or not to see the true ground state wave function and ground state energy plotted and choose the color for those curves.
- Choose whether or not to use importance sampling and if so, choose the appropriate variational parameter for the trial wave function. For example, for the simple harmonic potential the trial function is $e^{-\alpha x^2}$ where α is the
- Set the options for all the graphs. The options are explained for each graph below.

There are six different available graphs but only three of which are available for viewing at any one time on the three graph panels. The user can freely choose which graph to put on which panel but it is advisable to only view the "individual walkers" panel on the big graph because of the small details. The graphs are:

- Histogram: shows the unnormalized histogram of the distribution of walkers. The user can choose the upper and lower x-limits.
- Number of walkers: shows the variation of the number of walkers as a function of time.

- Reference energy: shows the variation of the reference energy as a function of time. The user can choose whether or not to join the points of the plot.
- E_0 estimate: shows the estimate for the energy of the ground state over time.
- Φ_0 estimate: shows the current estimate for the ground state wave function. The blue curve shows the true ground state wave function Φ_0. If importance sampling is used, the yellow curve shows the wave function that the distribution will converge to ($\Phi_0\Phi_\mathrm{T}$ where Φ_T is the trial wavefunction).
- Individual walkers: shows the movement of individual walkers. It is recommended that the number of walkers be kept low, say below 10, for ease of viewing. The time axis runs in the vertical direction from up to down. Every horizontal line denotes one time step. The x-axis goes from left to right with the vertical line in the middle denoting the origin. The walkers are denoted by solid circles whose sizes within one time step shows their relative weights. The same walkers between the time steps are connected by lines and over time these lines trace out their trajectories. If there is no line connecting a walker to the next time step that walker has died. If there are two different lines emerging from the same walker going into the next time step, that walker has multiplied into new walkers. If importance sampling is used, a horizontal line emerging from each walker will show the drift force that walker experiences. To make these lines more visible it is advisable to increase the time step $\Delta\tau$ because the force is proportional to the time step.

4.3 Auxiliary classes

The application also includes a number of helper components. The respective authors are noted in the documentation of the source code.

- The class `VariateGenerator` generates random numbers from a normal, uniform or delta distribution.
- The package `graphs` includes classes that draw various graphs such as histograms and time graphs. They extends the functionality of the package `edu.uah.math`.
- The class `ObjectCloner` performs "deep copy" of JAVA objects. By default JAVA performs "shallow" copying of objects, meaning it creates new references to the original objects instead of creating new objects that contain the same fields as the original objects. This is problematic when the new copies are needed but the original objects have gone "out of scope" and are destroyed (e.g. when we need to store the previuos array of walkers in order to plot their trajectories in the GUI).

5 Future work

- Clean up the GUI. The current GUI code sets up most of the interface programatically. Although this shows the user exactly the order and dependence among the various components of the GUI, most of the graphical objects created (mostly the panels) are never referenced again and thus do not need to be given meaningful names. Thus the GUI should be redone using a graphical interface designer (such as the plug-in Windowbuilder for the Eclipse

development environment or the GUI designing tool in the Netbeans environment) which automatically generates code to create these components. The event-handling mechanism of the GUI can also be improved. Right now all events generated by the vast majority of components are handled by a single listener, which checks for the source of the event against all the possible sources in the GUI. This is inefficient and makes maintenance difficult. Each component (or very similar components) should have their own event listener.

- Use better software engineering principles to improve the communication between the classes in the application and making future modifications easier. Right now to add any new function the programer must modify a substantial portion of the source code.
- Add the capability to simulate three-dimensional potentials. This will require a new JAVA graphing class and add a lot more options e.g. viewing angles. In this case it might be better to use dedicated mathematical software (MATHEMATICA or MATLAB) to provide the front-end graphics with JAVA providing the back-end simulation.
- Add the capability to simulate many-electron systems. This will require the ability to let user change the trial wave functions e.g. the Jastrow correlation factor in Jastrow-Slater trial wave functions.
- Add a module that shows users how to reduce the variance by decorrelating the data.

References

[1] I. Kosztin, B. Faber, and K. Schulten, Am. J. Phys. **64**, 633 (1996).

[2] I. Terrell, "Diffusion monte carlo: A java based simulation and visualization," `https://www.huy.dev/Terrel-thesis.pdf` (2004).