Variational Quantum Monte Carlo for the Harmonic Oscillator

In this topic, we will study simple quantum mechanical systems of particles which have bound states. The simplest such system is the quantum harmonic oscillator in one dimension. To find the energy eigenstates, we solve the time-independent Schrödinger equation

$$H \psi(x) = \left[ -\frac{\hbar^2}{2m} \frac{d^2}{dx^2} + \frac{1}{2} m \omega^2 x^2 \right] \psi(x) = E \psi(x) ,$$

subject to boundary conditions

$$\lim_{x \to \pm \infty} \psi(x) = 0 .$$

The solution of this equation is the wave function of the particle. The interpretation of the wave function $\psi(x)$ is that $|\psi(x)|^2 \, dx$ is the probability of finding the particle between position $x$ and position $x + dx$.

Such a simple one-dimensional problem can easily be solved numerically using deterministic algorithms such as Runge-Kutta or Numerov. In fact, the harmonic oscillator problem can be solved exactly. Solutions which satisfy the boundary conditions exist only for discrete eigenvalues of the energy

$$E_n = \left( n + \frac{1}{2} \right) \hbar \omega \quad n = 0, 1, 2, 3, \ldots$$

and the normalized energy eigenfunctions are given by

$$\psi_n(x) = \left( \frac{m \omega}{\pi \hbar} \right)^{1/4} \frac{1}{\sqrt{2^n n!}} H_n \left( x \sqrt{\frac{m \omega}{\hbar}} \right) e^{-m \omega x^2 / (2 \hbar)} ,$$

where $H_n$ are Hermite polynomials

$$H_0(y) = 1 , \quad H_1(y) = 2y , \quad H_2(y) = 4y^2 - 2 , \quad \text{etc.}$$

Exact solutions have been found only for a very small number of problems which can essentially be reduced to one-dimensional ordinary differential equations. Another example is the hydrogen atom which consists of a proton and an electron interacting through a Coulomb force.

The variational theorem

The eigenfunctions of a quantum mechanical problem are complete. This means that any wave function $\Psi(x)$ can be expressed as a linear superposition

$$\Psi(x) = \sum_n c_n \psi_n(x) ,$$

where $c_n$ are complex numbers. According to the rules of quantum mechanics, the average energy of a particle with this wave function is given by

$$\langle E \rangle = \frac{\int \, dx \, |\Psi(x)|^2 \frac{1}{2} m \omega^2 x^2}{\int \, dx \, |\Psi(x)|^2} .$$
The variational theorem states that $\langle E \rangle \geq E_0$ for any $\Psi$, and $\langle E \rangle = E_0$ if and only if $\Psi(x) = c_0 \Psi_0(x)$. It is easy to see this if we use the fact that the eigenfunctions $\psi_n(x)$ can be chosen to be orthonormal

$$\int dx \psi_n^*(x) \psi_{n'}(x) = \delta_{nn'}$$

$$\langle E \rangle = \frac{\sum_{n,n'} c_n^* E_n c_{n'} \int dx \psi_n^*(x) \psi_{n'}(x)}{\sum_{n,n'} c_n^* c_{n'} \int dx \psi_n^*(x) \psi_{n'}(x)} = \frac{\sum_n |c_n|^2 E_n}{\sum_n |c_n|^2} = E_0 + \frac{\sum_n |c_n|^2 (E_n - E_0)}{\sum_n |c_n|^2}$$

we have used the eigenvalue equation $H \psi_{n'} = E_n \psi_{n'}$. Because $E_n - E_0 > 0$, the second term in the last expression is positive and larger than zero unless all $c_n = 0$ for all $n \neq 0$.

The variational method is based on this important theorem: to estimate the ground state energy and wave function, choose a trial wave function $\Psi_{T,\alpha}(x)$ which depends on a parameter $\alpha$. The expectation value $\langle E \rangle$ will depend on the parameter $\alpha$, which can be varied to minimize $\langle E \rangle$. This energy and the corresponding $\Psi_{T,\alpha}(x)$ then provide the best estimates for the ground state energy and wave function.

Mean-field variational methods

For most quantum systems which involve more than two particles, i.e., atomic nuclei and electrons, numerical methods must be used. Deterministic variational methods can be used to solve Schrödinger’s equation for many-particle systems. These methods typically replace the effects of the many particles by an average mean field: each particle is then acted on by this field, thus reducing the problem to an effective one-particle system. This problem must be solved self-consistently because the mean field is determined by the positions of the particles, and the motion of the particles is determined by the mean field!

The problem with these methods is that they do not take into account many-particle effects and correlations between particles in a simple way.

Quantum Monte Carlo methods use random numbers and random walks to try to improve on deterministic variational methods.

Variational Monte Carlo (VMC)

In the Variational Monte Carlo method, a trial wave function $\Psi_{T,\alpha}$, which depends on a set of variational parameters $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_S)$, is carefully chosen.

An efficient way must be found to evaluate the expected value of the energy

$$\langle E \rangle = \frac{\int dR \Psi_{T,\alpha}^* H \Psi_{T,\alpha}}{\int dR |\Psi_{T,\alpha}|^2}$$

where $R = (r_1, \ldots, r_N)$ are the positions of the particles in the system. The problem is that this multi-dimensional integral must be evaluated many many times as the program searches the $\alpha$ parameter space for the minimum $\langle E \rangle$.

Monte Carlo methods can be used to evaluate multi-dimensional integrals much more efficiently than deterministic methods. The key to using a Monte Carlo method is to define a positive definite weight function.
which is used to sample the important regions of the multi-dimensional space. In the VMC method, the weight function is taken to be

\[ \rho(R) = \frac{|\Psi_{T,\alpha}(R)|^2}{\int dR \, |\Psi_{T,\alpha}|^2}. \]

The expected value of the energy can then be written

\[ \langle E \rangle = \frac{\int dR \, |\Psi_{T,\alpha}|^2 \frac{H\Psi_{T,\alpha}}{\Psi_{T,\alpha}}}{\int dR \, |\Psi_{T,\alpha}|^2} = \int dR \, \rho(R) E_L(R), \]

where the local energy \( E_L(R) \) is defined by

\[ E_L(R) = \frac{H\Psi_{T,\alpha}(R)}{\Psi_{T,\alpha}(R)}. \]

The variational wave function \( \Psi_{T,\alpha}(R) \) is usually chosen to be real and non-zero (almost) everywhere in the region of integration. In evaluating the ground state of the system, it can generally be chosen to be real and positive definite.

The VMC strategy is to generate a random set of points \( \{R_i\}, i = 1, \ldots, M \) in configuration space that are distributed according to \( \rho(R) \). Then

\[ \langle E \rangle = \frac{1}{M} \sum_{i=1}^{M} E_L(R_i). \]

**VMC Program for the Harmonic Oscillator**

Consider a simple variational trial wave function for the Harmonic Oscillator:

\[ \Psi_{T,\alpha}(x) = e^{-\alpha x^2}. \]

Let’s choose units so that \( m = 1, \ h = 1, \) and \( \omega = 1. \) The Hamiltonian operator in these units is

\[ H = -\frac{d^2}{dx^2} + \frac{1}{2} x^2, \]

from which the local energy can be derived:

\[ E_L(x) = \alpha + x^2 \left( \frac{1}{2} - 2\alpha^2 \right). \]

Note that when \( \alpha = 1/2 \) we obtain the exact ground state energy and eigenfunction.

The following program `vmc.cpp` implements the VMC method outlined above.

```
// Variational Monte Carlo for the harmonic oscillator

#include <cmath>
#include <cstdlib>
#include <iostream>
```
#include <fstream>
using namespace std;

#include "../tools/random.hpp"

The program uses $N$ Metropolis random walkers

The weight function
$$\rho(x) \sim e^{-2\alpha x^2},$$
can easily be generated using a single Metropolis random walker, as was done in Topic 3 to evaluate a Gaussian integral. However, in more complex problems, it is conventional to use a large number of independent random walkers that are started at random points in the configuration space. This is because the weight function can be very complicated in a multi-dimensional space: a single walker might have trouble locating all of the peaks in the distribution; using a large number of randomly located walkers improves the probability that the distribution will be correctly generated.

Variables to measure observables

Variables are introduced to accumulate $E_L$ values and compute the Monte Carlo average and error estimate. The probability distribution is accumulated in a histogram with bins of size $dx$ in the range $-10 \leq x \leq 10$.

```cpp
double eSum; // accumulator to find energy
double eSqdSum; // accumulator to find fluctuations in E
double xMin = -10; // minimum x for histogramming psi^2(x)
double xMax = +10; // maximum x for histogramming psi^2(x)
double dx = 0.1; // psi^2(x) histogram step size
double *psiSqd; // psi^2(x) histogram
int nPsiSqd; // size of array

void zeroAccumulators() {
    eSum = eSqdSum = 0;
    for (int i = 0; i < nPsiSqd; i++)
        psiSqd[i] = 0;
}
```
Initialization

The following function allocates memory to hold the positions of the walkers and distributes them uniformly at random in the range $-0.5 \leq x \leq 0.5$. The step size $\delta$ for the Metropolis walk is set to 1.

```cpp
void initialize() {
    x = new double [N];
    for (int i = 0; i < N; i++)
        x[i] = uniform_dist() - 0.5;
    delta = 1;

    nPsiSqd = int((xMax - xMin) / dx);
    psiSqd = new double [nPsiSqd];

    zeroAccumulators();
}
```

Probability function and local energy

The following function evaluates the ratio

$$w = \frac{\rho(x_{\text{trial}})}{\rho(x)}$$

which is used in the Metropolis algorithm: if $w \geq 1$ the step is accepted unconditionally; and if $w < 1$ the step is accepted only if $w$ is larger than a uniform random deviate between 0 and 1.

```cpp
double alpha; // trial function is exp(-alpha*x^2)

double p(double xTrial, double x) {
    // compute the ratio of rho(xTrial) / rho(x)
    return exp(- 2 * alpha * (xTrial*xTrial - x*x));
}

double eLocal(double x) {
    // compute the local energy
    return alpha + x * x * (0.5 - 2 * alpha * alpha);
}
```
One Metropolis step

One Metropolis step is implemented as follows:

- Choose one of the $N$ walkers at random
- The walker takes a trial step to a new position that is Gaussian distributed with width $\delta$ around the old position. The function `gasdev` defined in `rng.h` returns a Gaussian deviate with unit width: multiplying this by a step size $\delta$ yields a Gaussian deviate with $\sigma = \delta$. This choice of trial step is suggested by the programs on the author’s web site.

```cpp
int nAccept; // accumulator for number of accepted steps
void MetropolisStep() {
   // choose a walker at random
   int n = int(uniform_dist() * N);

   // make a trial move
   double xTrial = x[n] + delta * normal_dist();

   // Metropolis test
   if (p(xTrial, x[n]) > uniform_dist()) {
      x[n] = xTrial;
      ++nAccept;
   }

   // accumulate energy and wave function
   double e = eLocal(x[n]);
   eSum += e;
   eSqdSum += e * e;
   int i = int((x[n] - xMin) / dx);
   if (i >= 0 && i < nPsiSqd)
      psiSqd[i] += 1;
}
```

As usual, when we have multiple walkers, one Monte Carlo Step is conventionally defined as $N$ Metropolis steps:

```cpp
void oneMonteCarloStep() {
   // ...
}
```
// perform N Metropolis steps
for (int i = 0; i < N; i++) {
    MetropolisStep();
}

Steering the computation with the main function

int main() {

    cout << " Variational Monte Carlo for Harmonic Oscillator\n";
    cout << " Enter number of walkers: ";
    cin >> N;
    cout << " Enter parameter alpha: ";
    cin >> alpha;
    cout << " Enter number of Monte Carlo steps: ";
    int MCSteps;
    cin >> MCSteps;

    initialize();

    As in all Monte Carlo calculations, some number of steps are taken and discarded to allow the walkers to come to “equilibrium.” The thermalization phase is also used to adjust the step size so that the acceptance ratio is approximately 50%. If $\delta$ is too small, then too many steps will be accepted; and conversely, if $\delta$ is too large, then too many steps will be rejected. Multiplying $\delta$ by one half of the acceptance ratio will increase $\delta$ if the ratio is larger than 0.5, and decrease $\delta$ if the ratio is smaller than 0.5.

    // perform 20% of MCSteps as thermalization steps
    // and adjust step size so acceptance ratio ~ 50%
    int thermSteps = int(0.2 * MCSteps);
    int adjustInterval = int(0.1 * thermSteps) + 1;
    int nAccept = 0;
    cout << " Performing " << thermSteps << " thermalization steps ..." << flush;
    for (int i = 0; i < thermSteps; i++) {
        oneMonteCarloStep();
        if ((i+1) % adjustInterval == 0) {
            delta *= nAccept / (0.5 * N * adjustInterval);
            nAccept = 0;
        }
    }

    // perform 20% of MCSteps as thermalization steps
    // and adjust step size so acceptance ratio ~ 50%
    int thermSteps = int(0.2 * MCSteps);
    int adjustInterval = int(0.1 * thermSteps) + 1;
    int nAccept = 0;
    cout << " Performing " << thermSteps << " thermalization steps ..." << flush;
    for (int i = 0; i < thermSteps; i++) {
        oneMonteCarloStep();
        if ((i+1) % adjustInterval == 0) {
            delta *= nAccept / (0.5 * N * adjustInterval);
            nAccept = 0;
        }
    }

}
Once the system has thermalized, the accumulators for observables are initialized and the production steps are taken.

```cpp
// production steps
zeroAccumulators();
nAccept = 0;
cout << " Performing " << MCSteps << " production steps ..." << flush;
for (int i = 0; i < MCSteps; i++)
    oneMonteCarloStep();
```

Finally the average value of the energy and the Monte Carlo error estimate are printed, and the probability distribution $\sim \psi_0^2(x)$ is written in the form of a histogram to a file.

```cpp
// compute and print energy
double eAve = eSum / double(N) / MCSteps;
```

The following plots show results for the average energy $\langle E \rangle$ and its variance $\langle E^2 \rangle - \langle E \rangle^2$ as functions of the variational parameter $\alpha$. Runs were performed with $N = 300$ walkers and $MCSteps = 10,000$. 

```cpp
double eVar = eSqdSum / double(N) / MCSteps - eAve * eAve;
double error = sqrt(eVar) / sqrt(double(N) * MCSteps);
cout << " \langle Energy \rangle = " << eAve << " +/- " << error
     << " \langle Variance \rangle = " << eVar << endl;
```

```cpp
// write wave function squared in file
ofstream file("psiSqd.data");
double psiNorm = 0;
for (int i = 0; i < nPsiSqd; i++)
    psiNorm += psiSqd[i] * dx;
for (int i = 0; i < nPsiSqd; i++) {
    double x = xMin + i * dx;
    file << x << 't' << psiSqd[i] / psiNorm << '\n';
}
file.close();
cout << " Probability density written in file psiSqd.data" << endl;
```
As might be expected, the average energy is minimum $\langle E \rangle = 1/2$, and the variance is zero, at $\alpha = 1/2$ which corresponds to the exact solution for the harmonic oscillator ground state.
Variational Monte Carlo for Hydrogen and Helium

The Hydrogen Atom

The Hydrogen atom is a system with two particles, electron and proton. The configuration space in which the system moves is therefore six dimensional. By moving to the center-of-mass system, the problem becomes effectively 3 dimensional, with Hamiltonian

\[ H = -\frac{\hbar^2}{2m} \nabla^2 - \frac{e^2}{r}, \]

where \( r = r_e - r_p \) is the relative coordinate of the electron with respect to the proton, \( e \) is the magnitude of the electron’s charge, and \( m = m_e m_p / (m_e + m_p) \) is the reduced mass.

Reduction to a one-dimensional problem

By using conservation of angular motion and the fact that the ground state is spherically symmetric, i.e., it has zero orbital angular momentum, the problem can be reduced to one dimension with Hamiltonian operator

\[ H = -\frac{\hbar^2}{2m} \left[ \frac{d^2}{dr^2} + \frac{2}{r} \frac{d}{dr} \right] - \frac{e^2}{r}, \]

which depends on the radial coordinate \( r \).

Exact solution for the ground state

The exact ground state energy and wavefunction are given by

\[ E_0 = -\frac{e^2}{2a_0}, \quad \psi_0(r) \sim e^{-r/a_0}. \]

where the Bohr radius

\[ a_0 = \frac{\hbar^2}{me^2}. \]

It is convenient to use atomic units in which \( \hbar = m = e = 1 \) so

\[ H = -\frac{1}{2} \left[ \frac{d^2}{dr^2} + \frac{2}{r} \frac{d}{dr} \right] - \frac{1}{r}, \quad E_0 = -\frac{1}{2}, \quad \psi_0(r) \sim e^{-r}. \]

Variational trial wave function and local energy

A simple trial wave function for the Hydrogen atom ground state is

\[ \psi_{T,\alpha}(r) = e^{-\alpha r}. \]

The local energy for this choice can easily be computed:

\[ E_L(r) = \frac{1}{\psi_{T,\alpha}} H \psi_{T,\alpha}(r) = -\frac{1}{2} \left[ \alpha^2 - \frac{2\alpha}{r} \right] - \frac{1}{r}. \]

Note two important points about this local energy:
• It is minimum and also independent of \( r \) at \( \alpha = 1 \), which gives the exact ground state energy and eigenfunction.

• For \( \alpha \neq 1 \) it is singular at \( r = 0 \) where the potential diverges. For more complex problems, like the Helium atom to be considered next, these singularities can cause problems with the numerical calculation. To deal with these singularities, cusp conditions are used to restrict the variational parameters.

Two additional problems need to be addressed. Since \( r \geq 0 \), a one-dimensional Metropolis walker should not be allowed to cross the origin to \( r < 0 \). Also, the probability that the one-dimensional walker is found between \( r \) and \( r + dr \) must be proportional to \( 4\pi r^2 \), which is the surface area of a sphere of radius \( r \). We will use a walker in 3 dimensional space. Given a walker position \( r \) and a maximum step size \( \delta \), the next trial step is chosen uniformly at random within a cube of side \( 2\delta \) centered on the point \( r \) and aligned with the coordinate axes. This solves both of the problems above at the expense of making three calls to the random number generator for each trial move.

It is straightforward to adapt the harmonic oscillator program developed in the first lecture to find the ground state of the Hydrogen atom.

**Variational Monte Carlo for the Helium Atom**

The Helium atom is a 3-particle problem: two electrons orbit around a nucleus, which consists of two protons with charge \( e \) each and two neutral neutrons. The nucleus, which is \( \sim 8,000 \) times more massive than an electron, can be assumed to be at rest at the origin of the coordinate system. The electrons have positions \( r_1 \) and \( r_2 \). This is simpler than making a transformation to the center-of-mass system of the three particles, and it is sufficiently accurate.

If we use atomic units with \( \hbar = m_e = e = 1 \), the Hamiltonian for the motion of the two electrons can be written

\[
H = -\frac{1}{2} \nabla_1^2 - \frac{1}{2} \nabla_2^2 - \frac{2}{r_1} - \frac{2}{r_2} + \frac{1}{r_{12}},
\]

where \( r_{12} = |r_{12}| = |r_1 - r_2| \). The terms \(-2/r_i\) represent the negative (attractive) potential energy between each electron with charge \(-1\) and the Helium nucleus with charge \(+2\), and the term \(+1/r_{12}\) represents the positive (repulsive) potential energy between the two electrons.

A simple choice of variational trial wave function

If the repulsive term \( 1/r_{12} \) were not present, then the Hamiltonian would be that of two independent Hydrogen-like atoms. It can be shown that the energy and ground state wave function of a Hydrogen-like atom whose nucleus has charge \( Z \) are given by

\[
E_0 = -\frac{Z^2}{2}, \quad \psi_0 \sim e^{-Zr}.
\]

The wave function of the combined atom with two non-interacting electrons would be the product of two such wave functions:

\[
\psi(r_1, r_2) \sim e^{-2r_1}e^{-2r_2}.
\]
This suggests a trial wave function of the form

$$\Psi_{T,\alpha} = e^{-\alpha r_1} e^{-\alpha r_2},$$

similar to what was done for the Hydrogen atom. If the electron-electron interaction is neglected, then the average energy with this wave function can be calculated

$$\langle -\frac{1}{2} \nabla_1^2 - \frac{1}{2} \nabla_2^2 - \frac{2}{r_1} - \frac{2}{r_2} \rangle = 2 \times \frac{\alpha^2}{2} - 4 \times \alpha,$$

which has a minimum at $$\alpha = 4$$, which gives $$\langle E \rangle = -4$$. The experimentally measured ground state energy is $$E_0 = -2.904$$. The difference is due to the repulsive electron-electron interaction, which raises the energy. It is straightforward to show that

$$\langle \frac{1}{r_{12}} \rangle = \frac{5 \alpha}{8^3 \alpha}.$$

The average energy including the electron-electron interaction is

$$\langle -\frac{1}{2} \nabla_1^2 - \frac{1}{2} \nabla_2^2 - \frac{2}{r_1} - \frac{2}{r_2} + \frac{1}{r_{12}} \rangle = \alpha^2 - 4\alpha + \frac{5}{8} \alpha = \alpha^2 - \frac{27}{8} \alpha.$$

This expression has a minimum at $$\alpha = 27/16$$, which gives a variational estimate $$\langle E \rangle = -2.8477$$. This shows that the repulsion between the electrons is important. The simple product wave function gives a remarkably good variational estimate just 2% higher than the experimental value.

---

Padé-Jastrow wave function

We will use the trial wave function

$$\Psi(r_1, r_2) = e^{-2r_1} e^{-2r_2} e^{\frac{r_{12}}{2(1+\alpha r_{12})}},$$

with $$\alpha$$ as a variational parameter. The local energy with this wave function can be calculated

$$E_L(r_1, r_2) = -4 + \frac{\alpha}{(1+\alpha r_{12})} + \frac{\alpha}{(1+\alpha r_{12})^2} + \frac{\alpha}{(1+\alpha r_{12})^3}$$

$$-\frac{1}{4(1+\alpha r_{12})^4} + \frac{r_{12} \cdot (r_1 - r_2)}{(1+\alpha r_{12})^2}.$$

VMC program for the Helium Atom

The following program vmc-he.cpp implements this trial function choice.

```
// Variational Monte Carlo for the Helium Atom

#include <cmath>
#include <cstdlib>
#include <iostream>

using namespace std;
```

vmc-he.cpp
#include "../tools/random.hpp"

using namespace std;

const int NDIM = 3; // dimensionality of space
const int NELE = 2; // number of electrons
int N; // number of walkers
double (*r)[NELE][NDIM]; // walker coordinates in 6-D configuration space

double alpha; // Pade-Jastrow variational parameter
double delta; // trial step size

void initialize() {
    r = new double [N][NELE][NDIM];
    for (int n = 0; n < N; n++)
        for (int e = 0; e < NELE; e++)
            for (int d = 0; d < NDIM; d++)
                r[n][e][d] = uniform_dist() - 0.5;
    delta = 1;
}

double eSum;
double eSqdSum;

void zeroAccumulators() {
    eSum = eSqdSum = 0;
}

double Psi(double *rElectron1, double *rElectron2) {
    // value of trial wave function for walker n
    double r1 = 0, r2 = 0, r12 = 0;
    for (int d = 0; d < 3; d++) {
        r1 += rElectron1[d] * rElectron1[d];
        r2 += rElectron2[d] * rElectron2[d];
        r12 += (rElectron1[d] - rElectron2[d])
            * (rElectron1[d] - rElectron2[d]);
    }
    r1 = sqrt(r1);
    r2 = sqrt(r2);
    r12 = sqrt(r12);
    double Psi = - 2*r1 - 2*r2 + r12 / (2 * (1 + alpha*r12));
    return exp(Psi);
}
double eLocal(double *rElectron1, double *rElectron2) {
    // value of trial wave function for walker n
    double r1 = 0, r2 = 0, r12 = 0;
    for (int d = 0; d < 3; d++) {
        r1 += rElectron1[d] * rElectron1[d];
        r2 += rElectron2[d] * rElectron2[d];
        r12 += (rElectron1[d] - rElectron2[d]) * (rElectron1[d] - rElectron2[d]);
    }
    r1 = sqrt(r1);
    r2 = sqrt(r2);
    r12 = sqrt(r12);
    double dotProd = 0;
    for (int d = 0; d < 3; d++) {
        dotProd += (rElectron1[d] - rElectron2[d]) / r12 *
            (rElectron1[d] / r1 - rElectron2[d] / r2);
    }
    double denom = 1 / (1 + alpha * r12);
    double denom2 = denom * denom;
    double denom3 = denom2 * denom;
    double denom4 = denom2 * denom2;
    double e = - 4 + alpha * (denom + denom2 + denom3)
        - denom4 / 4 + dotProd * denom2;
    return e;
}

int nAccept;

void MetropolisStep(int walker) {
    // make a trial move of each electron
    double rElectron1[3], rElectron2[3], rTrial1[3], rTrial2[3];
    for (int d = 0; d < 3; d++) {
        rElectron1[d] = r[walker][0][d];
        rTrial1[d] = rElectron1[d] + delta * (2 * uniform_dist() - 1);
        rElectron2[d] = r[walker][1][d];
        rTrial2[d] = rElectron2[d] + delta * (2 * uniform_dist() - 1);
    }
    // Metropolis test
double w = \( \frac{\Psi(r_{\text{Trial1}}, r_{\text{Trial2}})}{\Psi(r_{\text{Electron1}}, r_{\text{Electron2}})} \);
if (uniform_dist() < w * w) {
    for (int d = 0; d < 3; d++) {
        r[walker][0][d] = rElectron1[d] = rTrial1[d];
        r[walker][1][d] = rElectron2[d] = rTrial2[d];
    }
    ++nAccept;
}

// accumulate local energy
double e = eLocal(rElectron1, rElectron2);
eSum += e;
eSqdSum += e * e;
}

void oneMonteCarloStep() {
    // do Metropolis step for each walker
    for (int n = 0; n < N; n++)
        MetropolisStep(n);
}

int main() {
    cout << " Variational Monte Carlo for Helium Atom\n"
    << " ---------------------------\n";
    cout << " Enter number of walkers: ";
    cin >> N;
    cout << " Enter parameter Pade-Jastrow parameter alpha: ";
    cin >> alpha;
    cout << " Enter number of Monte Carlo steps: ";
    int MCSteps;
    cin >> MCSteps;
    initialize();

    // perform 20% of MCSteps as thermalization steps
    // and adjust step size so acceptance ratio ~50%
    int thermSteps = int(0.2 * MCSteps);
    int adjustInterval = int(0.1 * thermSteps) + 1;
    nAccept = 0;
    cout << " Performing " << thermSteps << " thermalization steps ..." << flush;
    for (int i = 0; i < thermSteps; i++) {
        // do Metropolis step for each walker
        for (int n = 0; n < N; n++)
            MetropolisStep(n);
    }

    // accumulate local energy
    double e = eLocal(rElectron1, rElectron2);
eSum += e;
eSqdSum += e * e;
    cout << " Variational Monte Carlo for Helium Atom\n"
    << " ---------------------------\n";
    cout << " Enter number of walkers: ";
    cin >> N;
    cout << " Enter parameter Pade-Jastrow parameter alpha: ";
    cin >> alpha;
    cout << " Enter number of Monte Carlo steps: ";
    int MCSteps;
    cin >> MCSteps;
    initialize();

    // perform 20% of MCSteps as thermalization steps
    // and adjust step size so acceptance ratio ~50%
    int thermSteps = int(0.2 * MCSteps);
    int adjustInterval = int(0.1 * thermSteps) + 1;
    nAccept = 0;
    cout << " Performing " << thermSteps << " thermalization steps ..." << flush;
    for (int i = 0; i < thermSteps; i++) {
        // do Metropolis step for each walker
        for (int n = 0; n < N; n++)
            MetropolisStep(n);
    }

    // accumulate local energy
    double e = eLocal(rElectron1, rElectron2);
eSum += e;
eSqdSum += e * e;
}

void oneMonteCarloStep() {
    // do Metropolis step for each walker
    for (int n = 0; n < N; n++)
        MetropolisStep(n);
}

int main() {
    cout << " Variational Monte Carlo for Helium Atom\n"
    << " ---------------------------\n";
    cout << " Enter number of walkers: ";
    cin >> N;
    cout << " Enter parameter Pade-Jastrow parameter alpha: ";
    cin >> alpha;
    cout << " Enter number of Monte Carlo steps: ";
    int MCSteps;
    cin >> MCSteps;
    initialize();

    // perform 20% of MCSteps as thermalization steps
    // and adjust step size so acceptance ratio ~50%
    int thermSteps = int(0.2 * MCSteps);
    int adjustInterval = int(0.1 * thermSteps) + 1;
    nAccept = 0;
    cout << " Performing " << thermSteps << " thermalization steps ..." << flush;
    for (int i = 0; i < thermSteps; i++) {
        // do Metropolis step for each walker
        for (int n = 0; n < N; n++)
            MetropolisStep(n);
    }

    // accumulate local energy
    double e = eLocal(rElectron1, rElectron2);
eSum += e;
eSqdSum += e * e;
    cout << " Variational Monte Carlo for Helium Atom\n"
    << " ---------------------------\n";
    cout << " Enter number of walkers: ";
    cin >> N;
    cout << " Enter parameter Pade-Jastrow parameter alpha: ";
    cin >> alpha;
    cout << " Enter number of Monte Carlo steps: ";
    int MCSteps;
    cin >> MCSteps;
    initialize();

    // perform 20% of MCSteps as thermalization steps
    // and adjust step size so acceptance ratio ~50%
    int thermSteps = int(0.2 * MCSteps);
    int adjustInterval = int(0.1 * thermSteps) + 1;
    nAccept = 0;
    cout << " Performing " << thermSteps << " thermalization steps ..." << flush;
    for (int i = 0; i < thermSteps; i++) {
        // do Metropolis step for each walker
        for (int n = 0; n < N; n++)
            MetropolisStep(n);
    }

    // accumulate local energy
    double e = eLocal(rElectron1, rElectron2);
eSum += e;
eSqdSum += e * e;
}

void oneMonteCarloStep() {
    // do Metropolis step for each walker
    for (int n = 0; n < N; n++)
        MetropolisStep(n);
}

int main() {
    cout << " Variational Monte Carlo for Helium Atom\n"
    << " ---------------------------\n";
    cout << " Enter number of walkers: ";
    cin >> N;
    cout << " Enter parameter Pade-Jastrow parameter alpha: ";
    cin >> alpha;
    cout << " Enter number of Monte Carlo steps: ";
    int MCSteps;
    cin >> MCSteps;
    initialize();

    // perform 20% of MCSteps as thermalization steps
    // and adjust step size so acceptance ratio ~50%
    int thermSteps = int(0.2 * MCSteps);
    int adjustInterval = int(0.1 * thermSteps) + 1;
    nAccept = 0;
    cout << " Performing " << thermSteps << " thermalization steps ..." << flush;
    for (int i = 0; i < thermSteps; i++) {
        // do Metropolis step for each walker
        for (int n = 0; n < N; n++)
            MetropolisStep(n);
    }

    // accumulate local energy
    double e = eLocal(rElectron1, rElectron2);
eSum += e;
eSqdSum += e * e;
    cout << " Variational Monte Carlo for Helium Atom\n"
    << " ---------------------------\n";
    cout << " Enter number of walkers: ";
    cin >> N;
    cout << " Enter parameter Pade-Jastrow parameter alpha: ";
    cin >> alpha;
    cout << " Enter number of Monte Carlo steps: ";
    int MCSteps;
    cin >> MCSteps;
    initialize();

    // perform 20% of MCSteps as thermalization steps
    // and adjust step size so acceptance ratio ~50%
    int thermSteps = int(0.2 * MCSteps);
    int adjustInterval = int(0.1 * thermSteps) + 1;
    nAccept = 0;
    cout << " Performing " << thermSteps << " thermalization steps ..." << flush;
    for (int i = 0; i < thermSteps; i++) {
        // do Metropolis step for each walker
        for (int n = 0; n < N; n++)
            MetropolisStep(n);
    }

    // accumulate local energy
    double e = eLocal(rElectron1, rElectron2);
eSum += e;
eSqdSum += e * e;
    cout << " Variational Monte Carlo for Helium Atom\n"
    << " ---------------------------\n";
    cout << " Enter number of walkers: ";
    cin >> N;
    cout << " Enter parameter Pade-Jastrow parameter alpha: ";
    cin >> alpha;
    cout << " Enter number of Monte Carlo steps: ";
    int MCSteps;
    cin >> MCSteps;
    initialize();

    // perform 20% of MCSteps as thermalization steps
    // and adjust step size so acceptance ratio ~50%
    int thermSteps = int(0.2 * MCSteps);
    int adjustInterval = int(0.1 * thermSteps) + 1;
    nAccept = 0;
    cout << " Performing " << thermSteps << " thermalization steps ..." << flush;
    for (int i = 0; i < thermSteps; i++) {
        // do Metropolis step for each walker
        for (int n = 0; n < N; n++)
            MetropolisStep(n);
    }

    // accumulate local energy
    double e = eLocal(rElectron1, rElectron2);
eSum += e;
eSqdSum += e * e;
}
oneMonteCarloStep();
if ((i+1) % adjustInterval == 0) {
    delta *= nAccept / (0.5 * N * adjustInterval);
    nAccept = 0;
}
}
cout << "\n Adjusted step size delta = " << delta << endl;

// production steps
zeroAccumulators();
nAccept = 0;
cout << " Performing " << MCSteps << " production steps ..." << flush;
for (int i = 0; i < MCSteps; i++)
    oneMonteCarloStep();

// compute and print energy
double eAve = eSum / double(N) / MCSteps;
double eVar = eSqdSum / double(N) / MCSteps - eAve * eAve;
double error = sqrt(eVar) / sqrt(double(N) * MCSteps);
cout << "\n <Energy> = " << eAve << " +/- " << error
<< " Variance = " << eVar << endl;

Appendix: Derivation of Local Energy for Helium

The Padé-Jastrow wave function has the form
\[ \Psi(r_1, r_2) = e^{-2r_1} e^{-2r_2} e^{\frac{r_{12}}{2(1 + \alpha r_{12})}} \],
where \( \alpha \) is a variational parameter.

The local energy is given by the equation
\[ E_L = \frac{1}{\Psi} \left[ -\frac{1}{2} \nabla_1^2 - \frac{1}{2} \nabla_2^2 - \frac{2}{r_1} - \frac{2}{r_2} + \frac{1}{r_{12}} \right] \Psi. \]

Thijssen First Edition (1999) gives the local energy in Eq. (12.10) on page 318:
\[ E_L = \frac{\alpha_1}{r_1} + \frac{\alpha_1}{r_2} - \frac{\alpha_1 \hat{r}_{12}}{2(1 + \alpha_2 r_{12})} (\hat{r}_1 - \hat{r}_2) - \frac{1}{(1 + \alpha_2 r_{12})^3} - \frac{1}{4(1 + \alpha_2 r_{12})^4} - \frac{2}{r_1} - \frac{2}{r_2} + \frac{1}{r_{12}}. \]

Thijssen Second Edition (2007) gives the local energy in Eq. (12.12) on page 377:
\[ E_L = -4 + (\hat{r}_1 - \hat{r}_2) \cdot (r_1 - r_2) \frac{1}{r_{12}(1 + \alpha r_{12})^2} - \frac{1}{4(1 + \alpha r_{12})^4} + \frac{1}{r_{12}}. \]

These expression agree for \( \alpha_1 = 2 \) and \( \alpha_2 = \alpha \). Let’s write
\[ \Psi = f(r_1)f(r_2)g(r_{12}) \]
where \( f(r) = e^{-2r} \) and \( g(r) = e^{r/(2(1+\alpha r))} \). Then,
\[
\nabla_1^2 f(r_1)g(r_{12}) = (\nabla_1^2 f(r_1))g(r_{12}) + f(r_1)(\nabla_1^2 g(r_{12})) + 2(\nabla_1 f(r_1)) \cdot (\nabla_1 g(r_{12})).
\]

The Laplacian of a scalar function can be computed as follows:
\[
\nabla_1^2 f(r_1) = \nabla \cdot (\nabla f(r_1)) = \hat{r}_1 \cdot f'(r_1) \hat{r}_1 + f'(r_1) \hat{r}_1 = f''(r_1) + \frac{2}{r_1} f'(r_1),
\]
where \( \hat{r}_1 = \hat{r}_1/r_1 \) is a unit vector. Similarly
\[
\nabla_1^2 g(r_{12}) = g''(r_{12}) + \frac{2}{r_{12}} g'(r_{12}),
\]
\[
(\nabla_1 f(r_1)) \cdot (\nabla_1 g(r_{12})) = \hat{r}_1 \cdot \hat{r}_{12} f'(r_1) g'(r_{12}),
\]
and
\[
(\nabla_2^2 f(r_2)) \cdot (\nabla_2^2 g(r_{12})) = \hat{r}_2 \cdot \hat{r}_{21} f'(r_2) g'(r_{21}).
\]

Note that \( r_{12} = r_{21} \), but \( \hat{r}_{12} = -\hat{r}_{21} \).

The derivatives of the scalar functions are easily computed:
\[
\frac{f'(r)}{f(r)} = -2, \quad \frac{f''(r)}{f(r)} = 4,
\]
\[
\frac{g'(r)}{g(r)} = \frac{1}{2(1+\alpha r)^2}, \quad \frac{g''(r)}{g(r)} = \frac{1}{4(1+\alpha r)^4} - \frac{\alpha}{(1+\alpha r)^3}.
\]

Collecting all terms, the local energy is given by
\[
E_L = -\frac{f''(r_1)}{2f(r_1)} - \frac{f''(r_2)}{2f(r_2)} - \frac{g''(r_{12})}{r_1 f(r_1)} - \frac{g''(r_{12})}{r_1 f(r_2)} - \frac{2g'(r_{12})}{r_1 f(r_1) g(r_{12})} - \frac{2g'(r_{12})}{r_1 f(r_2) g(r_{12})} - \frac{2}{r_2} - \frac{2}{r_1} + \frac{1}{r_{12}}
\]
\[
= -4 - \frac{1}{4(1+\alpha r_{12})^4} + \frac{\alpha}{(1+\alpha r_{12})^3} + \frac{2}{r_1} - \frac{2}{r_2} - \frac{1}{r_{12}(1+\alpha r_{12})^2} + \frac{\hat{r}_1 \cdot \hat{r}_2 \cdot \hat{r}_{12}}{(1+\alpha r_{12})^2} - \frac{2}{r_1} + \frac{1}{r_{12}}.
\]

Using
\[
\frac{1}{r_{12}} - \frac{1}{r_{12}(1+\alpha r_{12})^2} = \frac{\alpha}{(1+\alpha r_{12})} + \frac{\alpha}{(1+\alpha r_{12})^2}
\]
gives the final form for the local energy
\[
E_L(r_1, r_2) = -4 + \frac{\alpha}{(1+\alpha r_{12})} + \frac{\alpha}{(1+\alpha r_{12})^2} + \frac{\alpha}{(1+\alpha r_{12})^3} - \frac{1}{4(1+\alpha r_{12})^4} + \frac{\hat{r}_{12} \cdot (\hat{r}_1 - \hat{r}_2)}{(1+\alpha r_{12})^2}.
\]

This result does not agree with Thijssen!
VMC for Helium: Slater-Jastrow Functions and Cusp Conditions

The choice of variational wave function is the most important step in doing a variational Monte Carlo calculation.

With a well-chosen trial wave function, writing the computer code is very straightforward. Obtaining accurate results requires large amounts of computer time because Monte Carlo errors scale like $\frac{1}{\sqrt{N}}$, where $N$ is the number of steps.

With a poorly chosen wave function, the random walk may difficulty finding the global minimum. The local energy can also have singularities which might result in large fluctuations in the average energy and hence large errors.

Slater-Jastrow functions

A very popular way of choosing a trial wave function is to start with single-particle wavefunctions. For example, the ground state wave function of a Helium nucleus with one electron is

$$\psi_0(r) \sim e^{-2r}.$$  

The Slater-Jastrow function, introduced by R. Jastrow Phys. Rev. 98, 1479-1484 (1955), is a product of single-particle wave functions multiplied by an exponential of many-particle correlation factors:

$$\Psi(x_1, \ldots, x_N) = \Psi_{AS}(x_1, \ldots, x_N) \exp \left[ \frac{1}{2} \sum_{i,j} \phi(r_{ij}) \right].$$

Here $x$ denotes the position $r$ and spin coordinates of an electron. $\Psi_{AS}$ is an antisymmetric product, or Slater Determinant of single-particle functions. The exponential is a two-body Jastrow wave function: note that it is symmetric under exchange of any two particles. Thus this trial wave function satisfies Pauli’s exclusion principle.

Why did we not use a Slater determinant in writing

$$\Psi(r_1, r_2) = e^{-2r_1}e^{-2r_2}e^{\frac{r_{ij}}{2(1+\alpha r_{ij})}}$$

for the He trial function? The reason we can use a symmetric product is that the required antisymmetry can be provided by the spins of the electrons. The Slater determinant in this instance is

$$\frac{1}{\sqrt{2}} \begin{vmatrix} \phi_0(r_1)|1\uparrow\rangle & \phi_0(r_2)|2\downarrow\rangle \\ \phi_0(r_1)|1\downarrow\rangle & \phi_0(r_2)|2\uparrow\rangle \end{vmatrix} = \phi_0(r_1)\phi_0(r_2) \frac{1}{\sqrt{2}} \left[ |1\uparrow\rangle|2\downarrow\rangle - |1\downarrow\rangle|2\uparrow\rangle \right],$$

i.e., the spin wave function is an antisymmetric singlet state, thus ensuring the overall wave function obeys the Pauli principle. Since we have neglected spin-dependent terms in the Hamiltonian, the spin wave function does not affect the local energy.

In the Helium VMC, $\phi(r_{ij})$ is taken to be a Padé approximant

$$\phi(r_{ij}) = \frac{r_{ij}}{1 + \alpha r_{ij}}.$$
A Padé approximant is simply a rational function (i.e., a ratio of two polynomials) which is used to approximate a function that can be expanded in a power series. Rational functions are very useful in representing a power series in certain limits: for example,

\[ \phi(r_{ij} \to 0) = 0, \quad \text{and} \quad \phi(r_{ij} \to \infty) = 1/\alpha, \]

for the He Padé exponent. Thus, if \( \alpha \ll 1 \), then the trial wave function is enhanced when the two electrons are far apart \( (r_{ij} \to \infty) \) relative to when they are close together \( (r_{ij} \to 0) \):

\[
\Psi(r_1, r_2) = \begin{cases} 
    e^{-2r_1}e^{-2r_2} & \text{when } r_{12} = 0 \\
    e^{-2r_1}e^{-2r_2} \times e^{1/\alpha} & \text{when } r_{ij} \gg 1 
\end{cases}
\]

Searching for a minimum with \( \alpha \ll 1 \) will tend to minimize the repulsive Coulomb interaction energy between the two electrons.

**Coulomb Singularities and Kato Cusp Conditions**

The potential energy contribution

\[ -\frac{2}{r_1} - \frac{2}{r_2} + \frac{1}{r_{12}} \]

to the total energy is singular when any of \( r_1, r_2, \) or \( r_{12} \) become very small, i.e., when an electron approaches the nucleus or when the two electrons approach one another.

Consider a Hydrogen-like atom in its ground state. The equation for the wave function is

\[
-\frac{1}{2} \left[ \frac{d^2}{dr^2} + \frac{2}{r} \frac{d}{dr} \right] \psi(r) - \frac{Z}{r} \psi(r) = E \psi(r).
\]

If the energy of the atom is to be finite, the divergent negative potential energy at \( r = 0 \) must be cancelled by a divergence in the kinetic energy, i.e.,

\[-\frac{1}{r} \left[ \frac{d}{dr} + Z \right] \psi(r) = \text{finite}.\]

Consider a trial wave function of the form

\[ \Psi(r) = e^{-\alpha r}. \]

This function has a cusp (Latin for a point, or the tip of a spear) at \( r = 0 \). A cusp is a point on a curve at which the tangent changes sign. Consider \( \Psi(r) \) as a function of \( x \) for \( y = z = 0 \). Its slope at \( r = 0 \) is discontinuous:

\[
\frac{d}{dx}e^{-\alpha x} = \begin{cases} 
    -\alpha e^{-\alpha |x|} & \text{for } x > 0 \\
    +\alpha e^{-\alpha |x|} & \text{for } x < 0 
\end{cases}
\]

The discontinuity from negative to positive \( x \) is \( 2\alpha \).
Substituting the trial function in singular part of the wave equation
\[-\frac{1}{r} \left( \frac{d}{dr} + Z \right) e^{-ar} = -\frac{1}{r} [-\alpha + Z] e^{-ar} = \text{finite} ,\]
give the cusp condition
\[\alpha = Z .\]

To further illustrate the cusp conditions, let’s consider the following generalized wave function for the Helium atom problem:
\[\Psi(r_1, r_2) = e^{-Zr_1} e^{-Zr_2} e^{i\beta r_{12}} (1 + \alpha r_{12}) ,\]
where \(Z\) and \(\beta\) are taken to be two additional variational parameters. With a little effort, it is straightforward to obtain an expression for the local energy:
\[
E_L(r_1, r_2) = -Z^2 + \frac{(Z - 2)}{r_1} + \frac{(Z - 2)}{r_2} + \frac{1}{r_{12}} \left[ 1 - \frac{2\beta}{(1 + \alpha r_{12})^2} \right] \\
+ \frac{2\alpha\beta}{(1 + \alpha r_{12})^3} - \frac{\beta^2}{(1 + \alpha r_{12})^4} + \frac{Z\beta}{(1 + \alpha r_{12})^2} \hat{r}_{12} \cdot (\hat{r}_1 - \hat{r}_2) .
\]
This expression has obvious Coulomb singularities. However, if we impose the cusp conditions
\[Z = 2 , \quad \beta = \frac{1}{2} ,\]
then all the singular terms cancel and we obtain the local energy used in the Helium VMC program:
\[
E_L(r_1, r_2) = -4 + \frac{\alpha}{(1 + \alpha r_{12})} + \frac{\alpha}{(1 + \alpha r_{12})^2} + \frac{\alpha}{(1 + \alpha r_{12})^3} - \frac{1}{4(1 + \alpha r_{12})^4} + \frac{\hat{r}_{12} \cdot (\hat{r}_1 - \hat{r}_2)}{(1 + \alpha r_{12})^2} .
\]

Basically, the cusp conditions ensure that the trial wave function satisfies the singular terms in Schrödinger’s equation exactly: for the true wave function, a singular negative potential energy contribution is cancelled by a large positive kinetic energy.

**VMC for Helium with parameters \(Z\) and \(\beta\)**

The program `vmc-he2` uses the same code as `vmc-he` but with the parameters \(Z\) and \(\beta\) included in the trial wave function and the local energy.

```cpp
// Variational Monte Carlo for the Helium Atom with Z and beta
#include <cmath>
#include <cstdlib>
#include <fstream>
```


```cpp
#include <iostream>
using namespace std;

#include "../tools/random.hpp"

const int NDIM = 3;  // dimensionality of space
const int NELE = 2;  // number of electrons
int N;  // number of walkers
double (*r)[NELE][NDIM];  // walker coordinates in 6-D configuration space

double alpha;  // Pade-Jastrow variational parameter
double delta;  // trial step size
double Z = 2;  // effective nuclear charge parameter
double beta = 0.5;  // effective electron-electron coupling parameter

void initialize() {
    r = new double [N][NELE][NDIM];
    for (int n = 0; n < N; n++)
        for (int e = 0; e < NELE; e++)
            for (int d = 0; d < NDIM; d++)
                r[n][e][d] = uniform_dist() - 0.5;
    delta = 1;
}

double eSum;
double eSqdSum;
void zeroAccumulators() {
eSum = eSqdSum = 0;
}

double Psi(double *rElectron1, double *rElectron2) {
    // value of trial wave function for walker n
    double r1 = 0, r2 = 0, r12 = 0;
    for (int d = 0; d < 3; d++) {
        r1 += rElectron1[d] * rElectron1[d];
        r2 += rElectron2[d] * rElectron2[d];
        r12 += (rElectron1[d] - rElectron2[d]) * (rElectron1[d] - rElectron2[d]);
    }
    r1 = sqrt(r1);
    r2 = sqrt(r2);
    r12 = sqrt(r12);
    return 1;
}
```
double Psi = - Z*r1 - Z*r2 + beta * r12 / (1 + alpha*r12);
return exp(Psi);
}

double eLocal(double *rElectron1, double *rElectron2) {
    // value of trial wave function for walker n
    double r1 = 0, r2 = 0, r12 = 0;
    for (int d = 0; d < 3; d++) {
        r1 += rElectron1[d] * rElectron1[d];
        r2 += rElectron2[d] * rElectron2[d];
        r12 += (rElectron1[d] - rElectron2[d]) *
               (rElectron1[d] - rElectron2[d]);
    }
    r1 = sqrt(r1);
    r2 = sqrt(r2);
    r12 = sqrt(r12);
    double dotProd = 0;
    for (int d = 0; d < 3; d++) {
        dotProd += (rElectron1[d] - rElectron2[d]) / r12 *
                  (rElectron1[d] / r1 - rElectron2[d] / r2);
        }
    }
    double denom = 1 / (1 + alpha * r12);
    double denom2 = denom * denom;
    double denom3 = denom2 * denom;
    double denom4 = denom2 * denom2;
    double e = - Z * Z + (Z - 2) *(1 / r1 + 1 / r2)
               + 1 / r12 * (1 - 2 * beta * denom2) + 2 * alpha * beta * denom3
               - beta * beta * denom4 + Z * beta * dotProd * denom2;
    return e;
}

int nAccept;

void MetropolisStep(int walker) {
    // make a trial move of each electron
    double rElectron1[3], rElectron2[3], rTrial1[3], rTrial2[3];
    for (int d = 0; d < 3; d++) {
        rElectron1[d] = r[walker][0][d];
        rTrial1[d] = rElectron1[d] + delta * (2 * uniform_dist() - 1);
        rElectron2[d] = r[walker][1][d];
        rTrial2[d] = rElectron2[d] + delta * (2 * uniform_dist() - 1);
        }
    }
    double<double Psi = - Z*r1 - Z*r2 + beta * r12 / (1 + alpha*r12);
return exp(Psi);
}

double eLocal(double *rElectron1, double *rElectron2) {
    // value of trial wave function for walker n
    double r1 = 0, r2 = 0, r12 = 0;
    for (int d = 0; d < 3; d++) {
        r1 += rElectron1[d] * rElectron1[d];
        r2 += rElectron2[d] * rElectron2[d];
        r12 += (rElectron1[d] - rElectron2[d]) *
               (rElectron1[d] - rElectron2[d]);
    }
    r1 = sqrt(r1);
    r2 = sqrt(r2);
    r12 = sqrt(r12);
    double dotProd = 0;
    for (int d = 0; d < 3; d++) {
        dotProd += (rElectron1[d] - rElectron2[d]) / r12 *
                  (rElectron1[d] / r1 - rElectron2[d] / r2);
    }
    double denom = 1 / (1 + alpha * r12);
    double denom2 = denom * denom;
    double denom3 = denom2 * denom;
    double denom4 = denom2 * denom2;
    double e = - Z * Z + (Z - 2) *(1 / r1 + 1 / r2)
               + 1 / r12 * (1 - 2 * beta * denom2) + 2 * alpha * beta * denom3
               - beta * beta * denom4 + Z * beta * dotProd * denom2;
    return e;
}

int nAccept;

void MetropolisStep(int walker) {
    // make a trial move of each electron
    double rElectron1[3], rElectron2[3], rTrial1[3], rTrial2[3];
    for (int d = 0; d < 3; d++) {
        rElectron1[d] = r[walker][0][d];
        rTrial1[d] = rElectron1[d] + delta * (2 * uniform_dist() - 1);
        rElectron2[d] = r[walker][1][d];
        rTrial2[d] = rElectron2[d] + delta * (2 * uniform_dist() - 1);
New function to do a Monte Carlo run

Let's add a new function which does a complete Monte Carlo run and computes the average energy and its variance.

```cpp
double eAve; // average energy
double eVar; // variance in the energy
int MCSSteps = 10000; // number of Monte Carlo steps per walker

void runMonteCarlo() {
    // perform 20% of MCSSteps as thermalization steps
    // and adjust step size so acceptance ratio ~50%
    int thermSteps = int(0.2 * MCSSteps);
    int adjustInterval = int(0.1 * thermSteps) + 1;
    nAccept = 0;
    for (int i = 0; i < thermSteps; i++) {
        MetropolisStep(n);
    }
    double e = eLocal(rElectron1, rElectron2);
    eSum += e;
    eSqdSum += e * e;

    // perform remaining steps
    for (int i = 0; i < MCSSteps - thermSteps; i++) {
        MetropolisStep(n);
    }

    eAve = eSum / nAccept;
    eVar = eSqdSum / nAccept - eAve * eAve;
}
```
oneMonteCarloStep();
    if ((i+1) % adjustInterval == 0) {
        delta *= nAccept / (0.5 * N * adjustInterval);
        nAccept = 0;
    }
}

// production steps
zeroAccumulators();
nAccept = 0;
for (int i = 0; i < MCSteps; i++)
    oneMonteCarloStep();
eAve = eSum / double(N) / MCSteps;
eVar = eSqdSum / double(N) / MCSteps - eAve * eAve;
}

Modified main function to steer the calculation

In the main function we will generate 3 sets of data by varying each of the three parameters \(Z, \beta, \alpha\) holding the other two constant.

```cpp
int main() {
    cout << " Variational Monte Carlo for Helium Atom\n"
         << " -----------------------------------\n";
    N = 300;
    cout << " Number of walkers = " << N << endl;
    cout << " Number of MCSteps = " << MCSteps << endl;
    initialize();

    // Vary Z holding beta and alpha fixed
    ofstream file("Z.data");
    beta = 0.5;
    alpha = 0.1;
    Z = 0.5;
    cout << " Varying Z with beta = 0.5 and alpha = 0.1" << endl;
    while (Z < 3.6) {
        runMonteCarlo();
        file << Z << 't' << eAve << 't' << eVar << '
';
        cout << " Z = " << Z << "tE = " << eAve
             << "tVariance = " << eVar << endl;
        Z += 0.25;
    }
```

PHY 411-506 Computational Physics 2 14 Monday, April 7
```cpp
// Vary beta holding Z and alpha fixed
file.open("beta.data");
Z = 2;
beta = 0;
cout << " Varying beta with Z = 2 and alpha = 0.1" << endl;
while (beta < 1.1) {
    runMonteCarlo();
    file << beta << ' ' << eAve << ' ' << eVar << '
';
    cout << " beta = " << beta << " \langle E \rangle = " << eAve
    << " Variance = " << eVar << endl;
    beta += 0.1;
}
file.close();

// Vary alpha holding Z and beta fixed
file.open("alpha.data");
beta = 0.5;
Z = 2;
alpha = 0;
cout << " Varying alpha with Z = 2 and beta = 0.5" << endl;
while (alpha < 0.6) {
    runMonteCarlo();
    file << alpha << ' ' << eAve << ' ' << eVar << '
';
    cout << " alpha = " << alpha << " \langle E \rangle = " << eAve
    << " Variance = " << eVar << endl;
    alpha += 0.05;
}
file.close();
```

**Output of the program**

The following plots show the effects of varying $Z$ with $\alpha = 0.1$ and $\beta = 0.5$, varying $\beta$ with $Z = 2$ and $\alpha = 0.1$, and varying $\alpha$ with $Z = 2$ and $\beta = 0.5$, respectively.
Note that the average energy is most sensitive to variations in the parameter $Z$: this shows that it is most important to use the cusp condition on the single particle trial functions. The $\beta$ dependence of the average energy is not as dramatic as the $Z$ dependence: this shows that the electron-electron interaction is not as important as the electron-nucleus interactions, as might be expected. With the cusp conditions $Z = 2$ and $\beta = 0.5$ imposed, the $\alpha$ dependence of the average energy is very small: this shows the importance of using the cusp conditions.

The following plots show the variance in the energy for the corresponding runs:

Note the very large dependence on $Z$, the moderate dependence on $\beta$ and the relatively small dependence on $\alpha$. Once again, this shows the importance of using the cusp conditions on $Z$ and $\beta$: if this is done, one does not need to waste a lot of computer time searching for the minimum in $\alpha$ or trying to reduce the variance.
The Diffusion Monte Carlo (DMC) Method

In this approach, the ground state of the system is found by modeling a diffusion process.

**Diffusion and random walks**

Consider a random walk on a lattice with spacing \( a \) in one dimension. The rule for the walk is that if the walker is at position \( x \) at time \( t \), then at time \( t + h \), the walker moves to the neighbor sites \( x \pm a \) with equal probabilities \( \alpha \) and remains at \( x \) with probability \( 1 - 2\alpha \): the sum of the three probabilities add up to 1.

Let’s consider an ensemble of a large number of such walkers. The number density of walkers is \( \rho(x, t) \), which means that, at time \( t \), the number of walkers between \( x \) and \( x + dx \) is \( \rho(x, t)dx \). Note: each walker moves on a lattice, but the lattices of different walkers are in general different.

The master equation

\[
\rho(x, t + h) - \rho(x, t) = \alpha \rho(x + a, t) + \alpha \rho(x - a, t) - 2\alpha \rho(x, t) ,
\]

says that the density of walkers at \( x \) increases in one time step \( h \) due to walkers from \( x \pm a \) moving to \( x \) with probability \( \alpha \), and decreases due to walkers moving from \( x \) to \( x \pm a \) with probability \( \alpha \).

If \( h \) and \( a \) are both small we can use Taylor expansions

\[
\rho(x, t + h) = \rho(x, t) + h \frac{\partial \rho}{\partial t} + \ldots \quad \rho(x \pm a, t) = \rho(x, t) \pm a \frac{\partial \rho}{\partial x} + \frac{1}{2} a^2 \frac{\partial^2 \rho}{\partial x^2} + \ldots
\]

In the continuum limit \( h \to 0, a \to 0 \) with \( a^2/h \) held constant, we obtain the diffusion equation

\[
\frac{\partial \rho}{\partial t} = \gamma \frac{\partial^2 \rho}{\partial x^2} ,
\]

where

\[
\gamma \equiv \lim_{h,a \to 0} \frac{\alpha a^2}{h} ,
\]

is called the diffusion constant for the system of walkers.

Green’s function for the diffusion equation

The density of walkers at time \( t \) can be computed from the initial density using the formula

\[
\rho(y, t) = \int dx \ G(x, y; t) \rho(x, 0) , \quad G(x, y; t) = \frac{1}{\sqrt{4\pi \gamma t}} e^{-(x-y)^2/(4\gamma t)} ,
\]

where \( G(x, y; t) \) is a Green’s function with the properties

\[
G(x, y; 0) = \delta(x - y) , \quad \text{and} \quad \int dx \ G(x, y; t) = 1 .
\]
In fact, $G(x, y; t)$ is the probability that a walker at $x$ (or $y$) at time $t = 0$ moves to $y$ (or $x$) at time $t$. This provides a way of implementing the random walk:

- Choose a step size $\Delta t$ in time

Let’s consider this step as a trial step in the Metropolis algorithm. Do we need to make a Metropolis type test before accepting the step? The answer is no, because the step is chosen according to a probability function which drives the distribution exactly to equilibrium as a function of time $t$.

Another way of seeing that every step can be accepted is from the physical meaning of diffusion. Typically, we have a dilute collection of non-interacting particles in a medium which can be considered to be a heat bath at constant temperature $T$. The particles undergo random thermal motion due to collisions with the molecules of the medium. The temperature of the medium determines the diffusion constant via Einstein’s relation

$$\gamma = \frac{k_B T}{\beta},$$

where $\beta$ is the drag coefficient, e.g., $\beta = 6\pi \eta R$ for Brownian spheres of radius $R$ moving in fluid with kinematic viscosity $\eta$ (not to be confused with the Gaussian deviate in the step). Since the diffusing particles are non-interacting, there is no energy cost when they move.
Connection with quantum mechanics

Consider the time-dependent Schrödinger equation for a free particle moving in one dimensions:

\[ i\hbar \frac{\partial \psi(x,t)}{\partial t} = -\frac{\hbar^2}{2m} \frac{\partial^2 \psi(x,t)}{\partial x^2}, \]

where \( m \) is the mass of the particle. This equation can be written

\[ \frac{\partial \psi(x,t)}{\partial t} = i\hbar \frac{\partial^2 \psi(x,t)}{\partial x^2} = \gamma_{\text{im}} \frac{\partial^2 \psi(x,t)}{\partial x^2}, \]

which is exactly of the form of a diffusion equation, but with and imaginary diffusion constant

\[ \gamma_{\text{im}} = \frac{i\hbar}{2m}. \]

Another way to write this equation with a real diffusion constant is to analytically continue the time \( t \rightarrow -i\tau \) to imaginary values:

\[ \frac{\partial \psi(x,\tau)}{\partial \tau} = \frac{\hbar}{2m} \frac{\partial^2 \psi(x,\tau)}{\partial x^2}. \]

Thus the motion of a quantum particle is equivalent to diffusion of a cloud of particles in imaginary time!

Diffusion leads the system into its ground state

Any initial wave function of the system can be expanded in a complete set of energy eigenfunctions:

\[ \Psi(x,0) = \sum_{n=0}^{\infty} c_n \psi_n(x). \]

The solution of the real time Schrödinger equation is then

\[ \Psi(x,t) = \sum_{n=0}^{\infty} c_n e^{-iE_n t/\hbar} \psi_n(x). \]

The solution of the imaginary time equation is got by analytically continuing this solution to imaginary time \( t \rightarrow -i\tau \):

\[ \Psi(x,\tau) = \sum_{n=0}^{\infty} c_n e^{-E_n \tau/\hbar} \psi_n(x). \]

As \( \tau \rightarrow \infty \), each mode in this equation is exponentially damped, with higher energies damped faster than lower energies. The ground state wave function can be extracted using the following limit:

\[ \lim_{\tau \rightarrow \infty} e^{E_0 \tau/\hbar} \Psi(x,\tau) = \lim_{\tau \rightarrow \infty} \sum_{n} c_n e^{-(E_n - E_0) \tau/\hbar} \psi_n(x) = c_0 \psi_0(x). \]

This result is the basis of the diffusion Monte Carlo approach.
Diffusion with a potential energy term

The equations considered above were for a free particle. A free particle is not very interesting, so let’s generalize this approach to a particle moving in a potential $V(x)$ for which the imaginary time equation to be solved is

$$\frac{\partial \psi(x, \tau)}{\partial \tau} = \frac{1}{2} \frac{\partial^2 \psi(x, \tau)}{\partial x^2} - V(x)\psi(x, \tau) ,$$

where we have set $\hbar = 1$ and $m = 1$.

We have seen that if $V = 0$, then this equation can be solved using a Green’s function

$$\rho(y, \tau) = \int dx \ G(x, y; \tau) \rho(x, 0) , \quad G(x, y; \tau) = \frac{1}{\sqrt{2\pi \tau}} e^{-\frac{(x-y)^2}{2\tau}} ,$$

for the probability density $\rho(x, \tau) = |\psi(x, \tau)|^2$. This solution preserves probability (or total number of particles in the diffusion problem).

The problem with adding the potential energy term is that it spoils this conservation of probability. This can be seen by neglecting the kinetic energy term:

$$\frac{\partial \psi(x, \tau)}{\partial \tau} = -V(x)\psi(x, \tau) , \quad \psi(x, \tau) = e^{-V(x)\tau}\psi(x, 0) ,$$

which implies that

$$\lim_{\tau \to \infty} \psi(x, \tau) = \begin{cases} 0 & \text{where } V(x) > 0 \\ \psi(x, 0) & \text{where } V(x) = 0 \\ \infty & \text{where } V(x) < 0 \end{cases}$$

Depending on the potential, the net probability $\int dx \ |\psi(x, \tau)|^2$ could go to zero or to infinity!

In the diffusion Monte Carlo method, this problem with the potential energy term is solved by modifying the equation as follows:

$$\frac{\partial \psi(x, \tau)}{\partial \tau} = \frac{1}{2} \frac{\partial^2 \psi(x, \tau)}{\partial x^2} - (V(x) - E_T)\psi(x, \tau) ,$$

where the quantity $E_T$ is adjusted as a function of $\tau$ so that the probability (number of walkers in the diffusion approach) remains constant. If in the limit $\tau \to \infty$ the solution $\psi(x, \tau) \to \psi(x)$ becomes independent of $\tau$, i.e., $\partial \psi / \partial \tau = 0$, then

$$\frac{1}{2} \frac{d^2 \psi(x)}{dx^2} + V(x)\psi(x) = E_T\psi(x) ,$$

that is, $\psi(x, \tau)$ tends to an eigenfunction of the quantum mechanical problem, and $E_T$ is the energy eigenvalue!
**Diffusion Monte Carlo algorithm**

The DMC algorithm is based on the ideas that the kinetic energy term can be represented by diffusion of random walkers, and the potential energy causes the number of walkers at a given point \( x \) to grow or decay. A simple form of the algorithm is as follows:

**Initialization:** Choose a time step \( \Delta \tau \) and a target number \( N_T \) of random walkers which are randomly located in a region where the wave function is expected to be large. Also choose a value for the parameter \( E_T \).

**Time Step:** The following two operations are carried out on each of the current number \( N \) of walkers:

- **Diffusion Step:** The kinetic energy shifts the walker to a new position with a step chosen at random from a Gaussian distribution with variance \( \Delta t \), exactly as in the case of a free particle.
- **Branching Step:** The potential energy, modified by the \( E_T \) parameter, causes a growth or decay in the number of walkers. This effect is implemented by computing

\[
q = e^{-\Delta \tau [V(x) - E_T]},
\]

The value of \( q \) determines whether this walker dies, survives, or is cloned. Note that \( q > 0 \). Let \( \lfloor q \rfloor \) be its integer part. Then \( q - \lfloor q \rfloor \) lies between 0 and 1. The walker is replaced with \( \lfloor q \rfloor \) identical copies with probability \( 1 - (q - \lfloor q \rfloor) \) and \( \lfloor q \rfloor + 1 \) copies with probability \( q - \lfloor q \rfloor \).

**Adjusting the value of \( E_T \):** At the end of the time step, the number of walkers \( N \) will have changed due to branching. If \( N > N_T \), then we need to increase \( E_T \) which will tend to reduce \( q \) and hence tend to kill walkers. Conversely, if \( N < N_T \), then reducing \( E_T \) will increase \( q \) and hence tend to generate more clones. This can be done for example by letting

\[
E_T \longrightarrow E_T + \alpha \ln \left( \frac{N_T}{N} \right),
\]

where \( \alpha \) is a small positive parameter.

**Diffusion Monte Carlo program for the 3-D harmonic oscillator**

The following program implements the DMC algorithm outlined above for the 3-D harmonic oscillator which has ground state energy and wave function

\[
E_0 = \frac{3}{2}, \quad \psi_0 = \frac{e^{-r^2/2}}{(2\pi)^{3/2}},
\]

using units with \( m = \omega = \hbar = 1 \).

```cpp
// Diffusion Monte Carlo program for the 3-D harmonic oscillator

#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iostream>

int main()
{
    // Code implementation...
    return 0;
}
```
#include <iostream>
using namespace std;

#include "../tools/random.hpp"

const int DIM = 3; // dimensionality of space

**Potential energy function**

This function evaluates the potential energy of the harmonic oscillator in $D$ dimensions given the position $r$ of the oscillator.

double V(double *r) { // harmonic oscillator in DIM dimensions
    double rSqd = 0;
    for (int d = 0; d < DIM; d++)
        rSqd += r[d] * r[d];
    return 0.5 * rSqd;
}

double dt; // Delta_t set by user

// random walkers
int N; // current number of walkers
int N_T; // desired target number of walkers
double **r; // x,y,z positions of walkers
bool *alive; // is this walker alive?

**Dynamical adjustment of array storage**

Since the number of walkers $N$ will change with time, we can either allocate large-enough arrays to accommodate this growth, or we can grow the arrays dynamically if necessary while the program is running. The following function is called when the $N$ might have changed and we wish to check whether an index is legal. If the array is too small to accommodate that index, it is replaced with a larger array with the values of the original elements preserved.

This can also be achieved automatically by using C++ **std::vector** objects instead of dynamically allocated arrays. The resize member function changes the number of components of the vector without changing the values of components in the smaller of the old and new vectors.
void ensureCapacity(int index) {
    static int maxN = 0; // remember the size of the array
    if (index < maxN) // no need to expand array
        return; // do nothing
    int oldMaxN = maxN; // remember the old capacity
    if (maxN > 0)
        maxN *= 2; // double capacity
    else
        maxN = 1;
    if (index > maxN - 1) // if this is not sufficient
        maxN = index + 1; // increase it so it is sufficient

    // allocate new storage
    double **rNew = new double* [maxN];
    bool *newAlive = new bool [maxN];
    for (int n = 0; n < maxN; n++) {
        rNew[n] = new double [DIM];
        if (n < oldMaxN) { // copy old values into new arrays
            for (int d = 0; d < DIM; d++)
                rNew[n][d] = r[n][d];
            newAlive[n] = alive[n];
        }
    }
    delete [] r; // release old memory
    delete [] r[n]; // release old memory
    r = rNew; // point r to the new memory
    delete [] alive;
    alive = newAlive;
}

We need to measure the energy, its variance, and the wave function of the ground state.

// observables
double ESum; // accumulator for energy
double ESqdSum; // accumulator for variance
double rMax = 4; // max value of r to measure psi
const int NPSI = 100; // number of bins for wave function
double psi[NPSI];// wave function histogram

void zeroAccumulators() {

\[
\begin{align*}
\text{ESum} &= \text{ESqdSum} = 0; \\
\text{for } (\text{int } i = 0; i < \text{NPSI}; i++) \\
\ &\quad \psi[i] = 0; \\
&} \]

```c

void initialize() {
    N = N_T; // set N to target number specified by user
    for (int n = 0; n < N; n++) {
        ensureCapacity(n);
        for (int d = 0; d < DIM; d++)
            r[n][d] = uniform_dist() - 0.5;
        alive[n] = true;
    }
    zeroAccumulators();
    E_T = 0; // initial guess for the ground state energy
}
```

**One Diffusion Monte Carlo step**

The following function implements the Diffusion Monte Carlo step algorithm on a particular walker. Recall that

- A Gaussian diffusive step is taken with step size \( \sqrt{\Delta t} \).
- A branching step is implemented with the walker dying, surviving or being cloned, depending on its potential energy.

```c

void oneMonteCarloStep(int n) {
    // Diffusive step
    for (int d = 0; d < DIM; d++)
        r[n][d] += normal_dist() * sqrt(dt);

    // Branching step
    double q = exp(- dt * (V(r[n]) - E_T));
    int survivors = int(q);
    if (q - survivors > uniform_dist())
        ++survivors;
}
```

`dmc.cpp`
void oneTimeStep() {
    // DMC step for each walker
    int N_0 = N;
    for (int n = 0; n < N_0; n++)
        oneMonteCarloStep(n);

    // remove all dead walkers from the arrays
    int newN = 0;
    for (int n = 0; n < N; n++)
        if (alive[n]) {
            if (n != newN) {
                for (int d = 0; d < DIM; d++)
                    r[newN][d] = r[n][d];
                alive[newN] = true;
                ++newN;
            }
        }

    // if survivors is zero, then kill the walker
    if (survivors == 0)
        alive[n] = false;
}

One time step $\Delta t$

One time step $\Delta t$ consists in the following:

- One DMC step is performed on each walker in turn.
- To make the living walkers easier to access, dead walkers are removed from the arrays.
- $E_T$ is adjusted to drive $N$ towards $N_T$.
- Data is accumulated to measure $\langle E \rangle$, its variance, and the ground state wave function.
The main function to steer the calculation

The user specifies the number of walkers, the time step size, and number of time steps. After initialization, 20% of the specified number of time steps are run to equilibrate the walkers. Then the production steps are taken. The Monte Carlo wave function and the exact wave function, both normalized unity in the plotting interval, are output to a file.

```cpp
int main() {
    cout << " Diffusion Monte Carlo for the 3-D Harmonic Oscillator\n";
    cout << " Enter desired target number of walkers: ";
    cin >> N_T;
    cout << " Enter time step dt: ";
    cin >> dt;
    cout << " Enter total number of time steps: ";
    int timeSteps;
    cin >> timeSteps;
    initialize();
```
// do 20% of timeSteps as thermalization steps
int thermSteps = int(0.2 * timeSteps);
for (int i = 0; i < thermSteps; i++)
    oneTimeStep();

// production steps
zeroAccumulators();
for (int i = 0; i < timeSteps; i++) {
    oneTimeStep();
}

// compute averages
double EAve = ESum / timeSteps;
double EVar = ESqdSum / timeSteps - EAve * EAve;
cout << " <E> = " << EAve << " +/- " << sqrt(EVar / timeSteps) << endl;
cout << " <E^2> - <E>^2 = " << EVar << endl;
double psiNorm = 0, psiExactNorm = 0;
double dr = rMax / NPSI;
for (int i = 0; i < NPSI; i++) {
    double r = i * dr;
    psiNorm += pow(r, DIM-1) * psi[i] * psi[i];
    psiExactNorm += pow(r, DIM-1) * exp(- r * r);
}
psiNorm = sqrt(psiNorm);
psiExactNorm = sqrt(psiExactNorm);
ofstream file("psi.data");
for (int i = 0; i < NPSI; i++) {
    double r = i * dr;
    file << r << '	' << pow(r, DIM-1) * psi[i] / psiNorm << '	' << pow(r, DIM-1) * exp(- r * r / 2) / psiExactNorm << '
';
}
file.close();
Output of the program

Diffusion Monte Carlo for the 3-D Harmonic Oscillator
-----------------------------------------------------

Enter desired target number of walkers: 300
Enter time step dt: 0.05
Enter total number of time steps: 4000
\( \langle E \rangle = 1.49113 \pm 0.0127478 \)
\( \langle E^2 \rangle - \langle E \rangle^2 = 0.650031 \)
The Path Integral Monte Carlo Method

The Path Integral formulation of quantum mechanics was suggested by Dirac Rev. Mod. Phys. 17, 195-199 (1945), and extensively developed by Feynman, Rev. Mod. Phys. 20, 367-387 (1948).

The method relates quantum mechanics of particles that move in in $d$ spatial dimensions to classical statistical mechanics of a corresponding system in $d + 1$ spatial dimensions, where the extra dimension can be viewed as an imaginary time for the quantum system.

The Path Integral Monte Carlo (PIMC) method then uses classical Monte Carlo (Topic 2) to compute the properties of the quantum system. The PIMC method can be used to compute time-dependent properties of the quantum system as well as properties of an ensemble of quantum systems in thermal equilibrium at finite temperature.

A good review of the PIMC method with many applications can be found in Ceperley Rev. Mod. Phys. 67, 279-355 (1995).

Path Integral Formulation in Imaginary Time

Consider a system of $N$ particles with positions $\mathcal{R} = \{r_i\}$ in $d = 3$ dimensions and Hamiltonian function

$$H = -\hbar^2 \sum_i \frac{1}{2m_i} \frac{\partial^2}{\partial r_i^2} + V(\mathcal{R}) .$$

A formal expression for the time evolution of the wavefunction of the system in the coordinate representation is

$$\psi(\mathcal{R}, t) = \langle \mathcal{R} | e^{-iHt/\hbar} | \psi \rangle .$$

There are two problems with this formal solution that complicate numerical application using Monte Carlo methods: * The exponential is complex valued and cannot be used as a real positive definite probability distribution. * $H$ is a quantum mechanical operator in an infinite dimensional Hilbert space, and the kinetic and potential energy operators do not commute.

Continuing to Imaginary Time

The first problem is solved by continuing to imaginary time $\tau = -it$ as was done in the DMC method. A connection is made to statistical mechanics by defining a partition function

$$Z(\beta) = \int d^N \mathcal{R} \langle \mathcal{R} | e^{-\tau H/\hbar} | \mathcal{R} \rangle = \text{Tr} e^{-\beta H} , \quad \beta = \frac{\tau}{\hbar} .$$
This partition function defines the quantum statistical mechanics of the system at temperature $k_B T = 1/\beta = \hbar/\tau$.

**Discretizing the Time Dimension**

The second problem is solved by replacing the finite time interval $t$ with a lattice of $M$ small time steps of size $\Delta \tau = \tau/M$. There are $M - 1$ intermediate time steps. At each intermediate time step a complete set of coordinate eigenstates

$$
1 = \int d^N R \langle R_i | R \rangle \langle R | R_i \rangle, \quad i = 1, \ldots, M - 1
$$

is inserted to factor the time evolution operator. The partition function is decomposed along the time direction into $M$ segments of duration $\tau$

$$
Z(\beta) = \int d^N R \langle R | e^{-\tau H/\hbar} | R \rangle = \int dR_0 \int dR_1 \cdots \int dR_M \langle R_0 | e^{-\Delta \tau H/\hbar} | R_{M-1} \rangle \cdots \langle R_2 | e^{-\Delta \tau H/\hbar} | R_1 \rangle \langle R_1 | e^{-\Delta \tau H/\hbar} | R_0 \rangle
$$

where the integration over $R_M = R_0 = R$ completes the original operator trace.

**Approximation for Short Time Evolution**

The Baker-Campbell-Hausdorff theorem states that

$$
e^{A} e^{B} = e^{C}
$$

if and only if

$$
C = A + B + \frac{1}{2} [A, B] + \cdots
$$

Applying this to the short-time evolution operator with

$$
C = -\Delta \tau H/\hbar, \quad B = -\Delta \tau V/\hbar, \quad A = -\Delta \tau K/\hbar,
$$

where the kinetic energy operator

$$
K = -\hbar^2 \sum_i \frac{1}{2m_i} \frac{\partial^2}{\partial r_i^2},
$$

we see that the commutators in the BCH formula are of order $(\Delta \tau)^2$ and higher. It seems reasonable to neglect these corrections in the limit of very small $\Delta \tau$, and this can be justified rigorously.

With this approximation the evolution in the first time step becomes

$$
\langle R_1 | e^{-\Delta \tau H/\hbar} | R_0 \rangle \simeq \langle R_1 | e^{-\Delta \tau K/\hbar} e^{-\Delta \tau V/\hbar} | R_0 \rangle = \langle R_1 | e^{-\Delta \tau K/\hbar} | R_0 \rangle e^{-\Delta \tau V(R_0)/\hbar}.
$$

To evaluate the matrix element of the kinetic energy operator, insert two complete sets of momentum
eigenstates
\[
\langle R_1 | e^{-\Delta \tau K/\hbar} | R_0 \rangle = \int dP_0 \int dP_1 \langle R_1 | P_1 \rangle \langle P_1 | e^{-\Delta \tau K/\hbar} | P_0 \rangle \langle P_0 | R_0 \rangle
\]
\[
= \left( \frac{m}{2\pi \hbar \Delta \tau} \right)^{Nd/2} \exp \left[ -\frac{m \Delta \tau}{2\hbar} \sum_i \left( \frac{r_{i1} - r_{i0}}{\Delta \tau} \right)^2 \right],
\]
where we have assumed for simplicity that all \( N \) particles have the same mass \( m_i = m \). Note that this is just a product of \( Nd \) free particle diffusion Green functions from the previous lecture
\[
G(x, y; t) = \frac{1}{\sqrt{4\pi \gamma t}} e^{-(x-y)^2/(4\gamma t)},
\]
one for each of the \( Nd \) coordinates.

Path Integral and Classical Action

The partition function can now be expressed entirely in terms of classical variables and free of any explicit quantum mechanical operators:
\[
Z(\beta) \simeq \left( \frac{m}{2\pi \hbar \Delta \tau} \right)^{MN/2} \int dR_0 \int dR_1 \ldots \int dR_{M-1}
\]
\[
\times \exp \left\{ -\frac{\Delta \tau}{\hbar} \sum_{j=0}^{M-1} \left[ \frac{m}{2} \left( \frac{R_{j+1} - R_j}{\Delta \tau} \right)^2 + V(R_j) \right] \right\},
\]
where \( L_{\text{classical}} = K - V \). This is Feynman’s Path Integral formulation of quantum mechanics. The probability amplitude for the quantum particle to travel from point \( x_1 \) at time \( t_1 \) to point \( x_M \) at time \( t_M \) is got by summing the imaginary exponential of the classical action in units of \( \hbar \) for all possible paths between the two points.

The partition function is got by continuing to imaginary time, and integrating also over \( x_1 = x_M \), i.e., by summing over all possible periodic paths with period \( \tau \).

Path Integral Monte Carlo Algorithm

The expression
\[
\sum_{j=0}^{M-1} \left[ \frac{m}{2} \left( \frac{R_{j+1} - R_j}{\Delta \tau} \right)^2 + V(R_j) \right]
\]
in the final formula for the partition function can be viewed as the potential energy function for a system of \( MNd \) classical particles with coordinates \( \{ R_j \} \), one for each of the \( d \) position vector components of each the \( N \) quantum particles at each of the \( M \) imaginary time steps.

The \( Nd \) particles at each time step are coupled by the the potential energy function \( V \) of the quantum problem, and the particles at neighboring time steps are coupled by a classical harmonic oscillator forces with force constant \( m/\Delta \tau \).

The system can now be simulated using the methods developed in Topic 2. The temperature \( k_B T = 1/\beta \) and \( d + 1 \) dimensional volume are fixed, and the forces are conservative. The simulation will give the finite temperature properties of the classical system in the canonical ensemble.

In the limit of zero temperature, i.e. large \( \beta = \tau/\hbar \) the classical system will tend to its lowest energy state. The corresponding quantum system will then be in its ground state, and the classical particle positions, averaged over the \( M \) time slices, will the distributed according to ground state wavefunction of the \( N \) quantum particles.

To obtain a reasonable approximation to the ground state, the temperature is chosen so that \( k_B T \) is much smaller than the level spacing of the low-lying energy eigenstates of the quantum system.

**PIMC Code for the Harmonic Oscillator**

The simple harmonic oscillator provides a good illustration of the PIMC method as shown in this PIMC Java Applet.

The time-sliced path integral (discretized partition function) can be evaluated analytically and the efficiency and error estimates of PIMC simulation can be checked against these exact results. This is done for example in M.F. Herman et al., J. Chem. Phys. 76, 5150-5155 (1982).

The Lagrangian in imaginary time is the energy of an oscillator in the finite temperature canonical ensemble of \( M \) “atoms”

\[-L = E = K + V = \frac{m}{2} \left( \frac{dx}{d\tau} \right)^2 + \frac{1}{2}m\omega_0^2x^2.\]

Choose units such that \( \hbar = m = \omega_0 = 1 \). The energy associated with the ensemble oscillator at time slice \( j \) is

\[E(x_j,j\Delta \tau) = \frac{1}{2} \left( \frac{x_{j+1} - x_j}{\Delta \tau} \right)^2 + V(x_j).\]

Herman et al., show that this formula for the energy is unstable when it is used to estimate fluctuations in the average energy. They introduce and alternative formula based on the **Virial theorem**

\[2 \langle K(x) \rangle = \left\langle x\frac{dV}{dx} \right\rangle,\]

which is generally preferred in PIMC simulations and is used in the following code.
// Path Integral Monte Carlo program for the 1-D harmonic oscillator

#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <vector>
using namespace std;
#include "../tools/random.hpp"

double V(double x) // potential energy function
{
    // use units such that m = 1 and \omega_0 = 1
    return 0.5 * pow(x, 2.0);
}

double dVdx(double x) // derivative dV(x)/dx used in virial theorem
{
    return x;
}

void initialize()
{
    Delta_tau = tau / M;
    x.resize(M);
    x_min = -x_max;
    dx = (x_max - x_min) / n_bins;
}
P.resize(n_bins);
cout << " Initializing atom positions using gsl::ran_uniform()" << endl;
for (int j = 0; j < M; ++j)
    x[j] = (2 * uniform_dist() - 1) * x_max;
}

bool Metropolis_step_accepted(double& x_new)
{
    // choose a time slice at random
    int j = int(uniform_dist() * M);
    // indexes of neighbors periodic in tau
    int j_minus = j - 1, j_plus = j + 1;
    if (j_minus < 0) j_minus = M - 1;
    if (j_plus > M - 1) j_plus = 0;
    // choose a random trial displacement
    double x_trial = x[j] + (2 * uniform_dist() - 1) * delta;
    // compute change in energy
    double Delta_E = V(x_trial) - V(x[j])
        + 0.5 * pow((x[j_plus] - x_trial) / Delta_tau, 2.0)
        + 0.5 * pow((x_trial - x[j_minus]) / Delta_tau, 2.0)
        - 0.5 * pow((x[j_plus] - x[j]) / Delta_tau, 2.0)
        - 0.5 * pow((x[j] - x[j_minus]) / Delta_tau, 2.0);
    if (Delta_E < 0.0 || exp(-Delta_tau * Delta_E) > uniform_dist()) {
        x_new = x[j] = x_trial;
        return true;
    } else {
        x_new = x[j];
        return false;
    }
}

int main()
{
    cout << " Path Integral Monte Carlo for the Harmonic Oscillator\n" << " -----------------------------------------------------\n";
    // set simulation parameters
    cout << " Imaginary time period tau = " << (tau = 10.0)
        << " Number of time slices M = " << (M = 100)
        << " Maximum displacement to bin x_max = " << (x_max = 4.0)
        << " Number of histogram bins in x = " << (n_bins = 100)
        << " Metropolis step size delta = " << (delta = 1.0)
        << " Number of Monte Carlo steps = " << (MC_steps = 100000)
        << endl;
initialize();
int therm_steps = MC_steps / 5, acceptances = 0;
double x_new = 0;
cout << " Doing " << therm_steps << " thermalization steps ...";
for (int step = 0; step < therm_steps; ++step)
    for (int j = 0; j < M; ++j)
        if (Metropolis_step_accepted(x_new))
            ++acceptances;
cout << "\n Percentage of accepted steps = "
    << acceptances / double(M * therm_steps) * 100.0 << endl;

double E_sum = 0, E_sqd_sum = 0;
P.clear();
acceptances = 0;
cout << " Doing " << MC_steps << " production steps ...";
for (int step = 0; step < MC_steps; ++step) {
    for (int j = 0; j < M; ++j) {
        if (Metropolis_step_accepted(x_new))
            ++acceptances;
        // add x_new to histogram bin
        int bin = int((x_new - x_min) / (x_max - x_min) * n_bins);
        if (bin >= 0 && bin < M)
            P[bin] += 1;
        // compute Energy using virial theorem formula and accumulate
        double E = V(x_new) + 0.5 * x_new * dVdx(x_new);
        E_sum += E;
        E_sqd_sum += E * E;
    }
    // compute averages
    double values = MC_steps * M;
    double E_ave = E_sum / values;
    double E_var = E_sqd_sum / values - E_ave * E_ave;
    cout << "\n <E> = " << E_ave << " +/- " << sqrt(E_var / values) << "
    <E^2> - <E>^2 = " << E_var << endl;
    ofstream ofs("pimc.out");
    E_ave = 0;
    for (int bin = 0; bin < n_bins; ++bin) {
        double x = x_min + dx * (bin + 0.5);
        ofs << " " << x << ' \t' << P[bin] / values << ' \n';
        E_ave += P[bin] / values * (0.5 * x * dVdx(x) + V(x));
    }
ofs.close();
cout << " \langle E \rangle from P(x) = " << E_ave << endl;
cout << " Probability histogram written to file pimc.out" << endl;

    return 0;
}