Diffusion Monte Carlo

We have seen that the diffusion and Fokker-Planck equations can be solved using random walks to generate any desired trial wave function.

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

$$\frac{\partial \psi(x,t)}{\partial t} = \frac{i\hbar}{2m} \frac{\partial^2 \psi(x,t)}{\partial x^2} = \gamma_{\rm im} \frac{\partial^2 \psi(x,t)}{\partial x^2} \ ,$$

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

$$\gamma_{\rm im} = \frac{i\hbar}{2m} \ .$$

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 $t \to -i\tau$:

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

As $\tau \to \infty$, 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:

$$\lim_{\tau\to\infty} e^{E_0\tau/\hbar}\Psi(x,\tau) = \lim_{\tau\to\infty} \sum_n c_n e^{-(E_n-E_0)\tau/\hbar}\psi_n(x) = c_0\psi_0(x) \ .$$

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 in the last lecture that if $V = 0$, then this equation can be solved using a Green's function

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

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:

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

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 $\int \mathrm{d}x\,|\psi(x,\tau)|^2$ 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:

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

where the quantity E_{T} is adjusted as a function of τ 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 τ, i.e., $\partial\psi/\partial\tau = 0$, then

$$-\frac{1}{2}\frac{\mathrm{d}^2\psi(x)}{\mathrm{d}x^2} + V(x)\psi(x) = E_{\mathrm{T}}\psi(x) \;,$$

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 $N_{\rm T}$ 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 $E_{\rm T}$.
- **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 Δt, exactly as in the case of a free particle.
  - **Branching Step:** The potential energy, modified by the $E_{\rm T}$ parameter, causes a growth or decay in the number of walkers. This effect is implemented by computing

$$q = e^{-\Delta\tau[V(x)-E_{\rm T}]} \ .$$

    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 $E_{\rm T}$:** At the end of the time step, the number of walkers N will have changed due to branching. If $N > N_{\rm T}$, then we need to *increase* $E_{\rm T}$ which will tend to *reduce* q and hence tend to kill walkers. Conversely, if $N < N_{\rm T}$, then *reducing* $E_{\rm T}$ will *increase* q and hence tend to generate more clones. The textbook suggests

$$E_{\rm T} \longrightarrow E_{\rm T} + \alpha \ln\left(\frac{N_{\rm T}}{N}\right) \ ,$$

  where α 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 an wave function

$$E_0 = \frac{3}{2} \ , \qquad \psi_0 = \frac{e^{-r^2/2}}{(2\pi)^{3/2}} \ ,$$

using units with $m = \omega = \hbar = 1$.

dmc.cpp

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

#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include "rng.h"

using namespace std;

int seed = -987654321;          // for ran2 and gasdev
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 $\mathbf{r}$ of the oscillator.

dmc.cpp

```
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
double E_T;                     // target energy
```

```
// random walkers                                                    24
int N;                          // current number of walkers        25
int N_T;                        // desired target number of walkers 26
double **r;                     // x,y,z positions of walkers       27
bool *alive;                    // is this walker alive?            28
```

Dynamical adjustment of array storage

Since the number of walkers N will change with time, we can either allocated large-enough arrays to accomodate 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 accomodate that index, it is replaced with a larger array with the values of the original elements preserved.

dmc.cpp

```
void ensureCapacity(int index) {                                    30

    static int maxN = 0;        // remember the size of the array   32

    if (index < maxN)           // no need to expand array          34
        return;                 // do nothing                       35

    int oldMaxN = maxN;         // remember the old capacity        37
    if (maxN > 0)                                                   38
        maxN *= 2;              // double capacity                  39
    else                                                            40
        maxN = 1;                                                   41
    if (index > maxN - 1)       // if this is not sufficient        42
        maxN = index + 1;       // increase it so it is sufficient  43

    // allocate new storage                                         45
```

```
    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[n];     // release old memory
        }
    }
    delete [] r;                // 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.

dmc.cpp

```
// 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() {
    ESum = ESqdSum = 0;
    for (int i = 0; i < NPSI; i++)
```

```
        psi[i] = 0;
}

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] = ran2(seed) - 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.

dmc.cpp

```
void oneMonteCarloStep(int n) {

    // Diffusive step
    for (int d = 0; d < DIM; d++)
        r[n][d] += gasdev(seed) * sqrt(dt);

    // Branching step
```

```
        double q = exp(- dt * (V(r[n]) - E_T));
        int survivors = int(q);
        if (q - survivors > ran2(seed))
            ++survivors;

        // append survivors-1 copies of the walker to the end of the array
        for (int i = 0; i < survivors - 1; i++) {
            ensureCapacity(N);
            for (int d = 0; d < DIM; d++)
                r[N][d] = r[n][d];
            alive[N] = true;
            ++N;
        }

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

One time step Δt

One time step Δt 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.

dmc.cpp

```
void oneTimeStep() {                                                    114

    // DMC step for each walker                                         116
    int N_0 = N;                                                        117
    for (int n = 0; n < N_0; n++)                                       118
        oneMonteCarloStep(n);                                           119

    // remove all dead walkers from the arrays                          121
    int newN = 0;                                                       122
    for (int n = 0; n < N; n++)                                         123
    if (alive[n]) {                                                     124
        if (n != newN) {                                                125
            for (int d = 0; d < DIM; d++)                               126
                r[newN][d] = r[n][d];                                   127
            alive[newN] = true;                                         128
        }                                                               129
        ++newN;                                                         130
    }                                                                   131
    N = newN;                                                           132

    // adjust E_T                                                       134
    E_T += log(N_T / double(N)) / 10;                                   135

    // measure energy, wave function                                    137
    ESum += E_T;                                                        138
    ESqdSum += E_T * E_T;                                               139
    for (int n = 0; n < N; n++) {                                       140
        double rSqd = 0;                                                141
        for (int d = 0; d < DIM; d++)                                   142
```

```
            rSqd = r[n][d] * r[n][d];                                    143
        int i = int(sqrt(rSqd) / rMax * NPSI);                           144
        if (i < NPSI)                                                    145
            psi[i] += 1;                                                 146
    }                                                                    147
}                                                                        148
```

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.

dmc.cpp

```
int main() {                                                             150

    cout << " Diffusion Monte Carlo for the 3-D Harmonic Oscillator\n"   152
         << " ------------------------------------------------\n";       153
    cout << " Enter desired target number of walkers:  ";                154
    cin >> N_T;                                                          155
    cout << " Enter time step dt:  ";                                    156
    cin >> dt;                                                           157
    cout << " Enter total number of time steps:  ";                      158
    int timeSteps;                                                       159
    cin >> timeSteps;                                                    160

    initialize();                                                        162

    // do 20% of timeSteps as thermalization steps                       164
    int thermSteps = int(0.2 * timeSteps);                               165
```

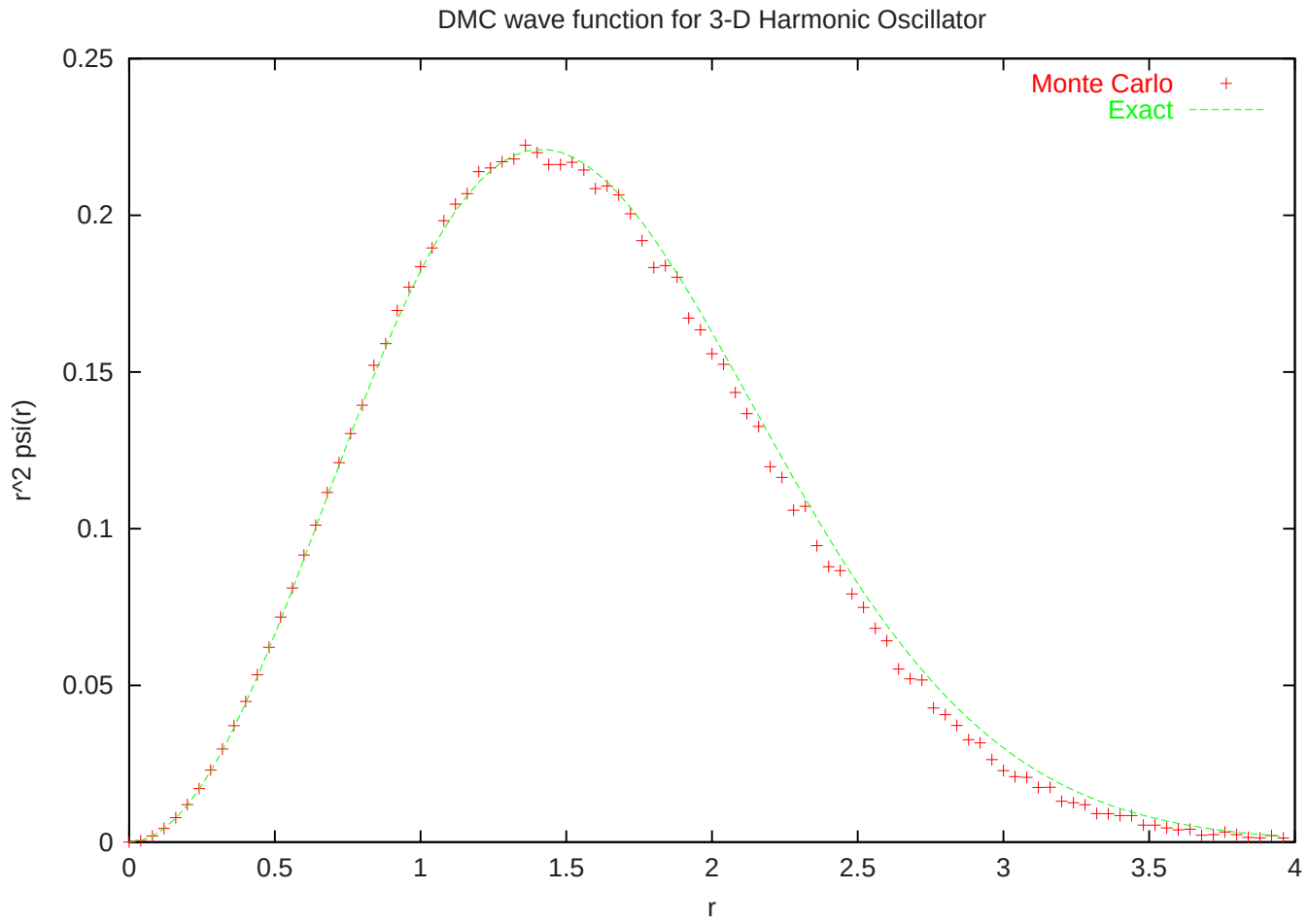
```
for (int i = 0; i < thermSteps; i++)                                    166
    oneTimeStep();                                                      167

// production steps                                                     169
zeroAccumulators();                                                     170
for (int i = 0; i < timeSteps; i++) {                                   171
    oneTimeStep();                                                      172
}                                                                       173

// compute averages                                                     175
double EAve = ESum / timeSteps;                                         176
double EVar = ESqdSum / timeSteps - EAve * EAve;                        177
cout << " <E> = " << EAve << " +/- " << sqrt(EVar / timeSteps) << endl; 178
cout << " <E^2> - <E>^2 = " << EVar << endl;                            179
double psiNorm = 0, psiExactNorm = 0;                                   180
double dr = rMax / NPSI;                                                181
for (int i = 0; i < NPSI; i++) {                                        182
    double r = i * dr;                                                  183
    psiNorm += r * r * psi[i] * psi[i];                                 184
    psiExactNorm += r * r * exp(- r * r);                               185
}                                                                       186
psiNorm = sqrt(psiNorm);                                                187
psiExactNorm = sqrt(psiExactNorm);                                      188
ofstream file("psi.data");                                              189
for (int i = 0; i < NPSI; i++) {                                        190
    double r = i * dr;                                                  191
    file << r << '\t' << r * r * psi[i] / psiNorm << '\t'               192
         << r * r * exp(- r * r / 2) / psiExactNorm << '\n';            193
}                                                                       194
```

```
        file.close();                                                    195
    }                                                                    196
```

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
<E> = 1.49113 +/- 0.0127478
<E^2> - <E>^2 = 0.650031
```