Choice of variational trial wave function

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 $1/\sqrt{N}$, 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 is a product of single-particle wave functions multiplied by an exponential of many-particle correlation factors:

$$\Psi(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \Psi_{\rm AS}(\mathbf{x}_1, \ldots, \mathbf{x}_N) \exp\left[\frac{1}{2}\sum_{i,j}^{N} \phi(r_{ij})\right] \ .$$

Here $\mathbf{x}$ denotes the position $\mathbf{r}$ *and* spin coordinates of an electron. $\Psi_{\rm AS}$ 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

$$\Psi(\mathbf{r}_1, \mathbf{r}_2) = e^{-2r_1} e^{-2r_2} e^{\frac{r_{12}}{2(1+\alpha r_{12})}}$$

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

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

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* function

$$\phi(r_{ij}) = \frac{r_{ij}}{1 + \alpha r_{ij}} \ .$$

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,

$$\phi(r_{ij} \to 0) = 0, \qquad \text{and} \qquad \phi(r_{ij} \to \infty) = 1/\alpha \ ,$$

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 $(r_{ij} \to 0)$:

$$\Psi(\mathbf{r}_1, \mathbf{r}_2) = \begin{cases} e^{-2r_1} e^{-2r_2} & \text{when } r_{12} = 0 \\ e^{-2r_1} e^{-2r_2} \times e^{1/\alpha} & \text{when } r_{ij} \gg 1 \end{cases}$$

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

Coulomb singularities and 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.

These singularities can cause instabilities in the random work and give rise to large errors. It is therefore very important to choose the trial wave function so that these Coulomb singularities do not occur in the expression for the local energy. This requirement leads to *cusp conditions* on the parameters in the trial wave function.

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

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

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.,

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

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:

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

The discontinuity from negative to positive x is 2α.

Substituting the trial function in singular part of the wave equation

$$-\frac{1}{r}\left[\frac{\mathrm{d}}{\mathrm{d}r}+Z\right]e^{-\alpha r}=-\frac{1}{r}\left[-\alpha+Z\right]e^{-\alpha r}=\text{finite}\ ,$$

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:

$$\Psi(\mathbf{r}_1, \mathbf{r}_2) = e^{-Zr_1} e^{-Zr_2} e^{\frac{\beta r_{12}}{(1+\alpha r_{12})}} ,$$

where Z and β are taken to be two additional variational parameters. With a little effort, it is straightforward to obtain an expression for the local energy:

$$\begin{aligned} E_{\rm L}(\mathbf{r}_1, \mathbf{r}_2) = & -Z^2 + \frac{(Z-2)}{r_1} + \frac{(Z-2)}{r_2} + \frac{1}{r_{12}}\left[1 - \frac{2\beta}{(1+\alpha r_{12})^2}\right] \\ & + \frac{2\alpha\beta}{(1+\alpha r_{12})^3} - \frac{\beta^2}{(1+\alpha r_{12})^4} + \frac{Z\beta}{(1+\alpha r_{12})^2}\hat{\mathbf{r}}_{12} \cdot (\hat{\mathbf{r}}_1 - \hat{\mathbf{r}}_2) . \end{aligned}$$

This expression has obvious Coulomb singularities. However, if we impose the cusp conditions

$$Z = 2 , \qquad \beta = \frac{1}{2} ,$$

then all the singular terms cancel and we obtain the local energy used in the Helium VMC program:

$$\begin{aligned} E_{\rm L}(\mathbf{r}_1, \mathbf{r}_2) = & -4 + \frac{\alpha}{(1+\alpha r_{12})} + \frac{\alpha}{(1+\alpha r_{12})^2} + \frac{\alpha}{(1+\alpha r_{12})^3} \\ & - \frac{1}{4(1+\alpha r_{12})^4} + \frac{\hat{\mathbf{r}}_{12} \cdot (\hat{\mathbf{r}}_1 - \hat{\mathbf{r}}_2)}{(1+\alpha r_{12})^2} . \end{aligned}$$

Note: Eqn. (12.10) in the textbook is not correct.

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 β

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

vmc-he2.cpp

```
// Variational Monte Carlo for the Helium Atom with Z and beta

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

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 parmeter
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] = qadran() - 0.5;                                    26
        delta = 1;                                                          27
    }                                                                       28

    double eSum;                                                            30
    double eSqdSum;                                                         31

    void zeroAccumulators() {                                               33
        eSum = eSqdSum = 0;                                                 34
    }                                                                       35

    double Psi(double *rElectron1, double *rElectron2) {                    37

        // value of trial wave function for walker n                        39
        double r1 = 0, r2 = 0, r12 = 0;                                     40
        for (int d = 0; d < 3; d++) {                                       41
            r1 += rElectron1[d] * rElectron1[d];                            42
            r2 += rElectron2[d] * rElectron2[d];                            43
            r12 += (rElectron1[d] - rElectron2[d])                          44
                   * (rElectron1[d] - rElectron2[d]);                       45
        }                                                                   46
        r1 = sqrt(r1);                                                      47
        r2 = sqrt(r2);                                                      48
        r12 = sqrt(r12);                                                    49
        double Psi = - Z*r1 - Z*r2 + beta * r12 / (1 + alpha*r12);          50
        return exp(Psi);                                                    51

    }                                                                       53
```

```
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) {                                          85

    // make a trial move of each electron                                  87
    double rElectron1[3], rElectron2[3], rTrial1[3], rTrial2[3];           88
    for (int d = 0; d < 3; d++) {                                          89
        rElectron1[d] = r[walker][0][d];                                   90
        rTrial1[d] = rElectron1[d] + delta * (2 * qadran() - 1);           91
        rElectron2[d] = r[walker][1][d];                                   92
        rTrial2[d] = rElectron2[d] + delta * (2 * qadran() - 1);           93
    }                                                                      94

    // Metropolis test                                                     96
    double w = Psi(rTrial1, rTrial2) / Psi(rElectron1, rElectron2);        97
    if (qadran() < w * w) {                                                98
        for (int d = 0; d < 3; d++) {                                      99
            r[walker][0][d] = rElectron1[d] = rTrial1[d];                 100
            r[walker][1][d] = rElectron2[d] = rTrial2[d];                 101
        }                                                                 102
        ++nAccept;                                                        103
    }                                                                     104

    // accumulate local energy                                            106
    double e = eLocal(rElectron1, rElectron2);                            107
    eSum += e;                                                            108
    eSqdSum += e * e;                                                     109
}                                                                         110

void oneMonteCarloStep() {                                                112
```

```
    // do Metropolis step for each walker                    114
    for (int n = 0; n < N; n++)                              115
        MetropolisStep(n);                                   116
}                                                            117
```

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.

vmc-he2.cpp

```
double eAve;              // average energy                  119
double eVar;              // variance in the energy          120
int MCSteps = 10000;      // number of Monte Carlo steps per walker  121

void runMonteCarlo() {                                       123

    // perform 20% of MCSteps as thermalization steps        125
    // and adjust step size so acceptance ratio ~50%         126
    int thermSteps = int(0.2 * MCSteps);                     127
    int adjustInterval = int(0.1 * thermSteps) + 1;          128
    nAccept = 0;                                             129
    for (int i = 0; i < thermSteps; i++) {                   130
        oneMonteCarloStep();                                 131
        if ((i+1) % adjustInterval == 0) {                   132
            delta *= nAccept / (0.5 * N * adjustInterval);   133
            nAccept = 0;                                     134
        }                                                    135
    }                                                        136
```

```
    // production steps                                                    138
    zeroAccumulators();                                                    139
    nAccept = 0;                                                           140
    for (int i = 0; i < MCSteps; i++)                                      141
        oneMonteCarloStep();                                               142
    eAve = eSum / double(N) / MCSteps;                                     143
    eVar = eSqdSum / double(N) / MCSteps - eAve * eAve;                    144
}                                                                          145
```

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, β, α holding the other two constant.

vmc-he2.cpp

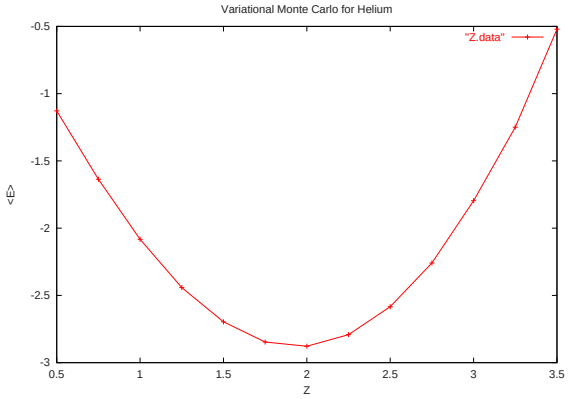
```
int main() {                                                               147

    cout << " Variational Monte Carlo for Helium Atom\n"                   149
         << " ------------------------------------\n";                    150
    N = 300;                                                               151
    cout << " Number of walkers = " << N << endl;                          152
    cout << " Number of MCSteps = " << MCSteps << endl;                    153
    initialize();                                                          154

    // Vary Z holding beta and alpha fixed                                 156
    ofstream file("Z.data");                                               157
    beta = 0.5;                                                            158
    alpha = 0.1;                                                           159
    Z = 0.5;                                                               160
    cout << " Varying Z with beta = 0.5 and alpha = 0.1" << endl;          161
```

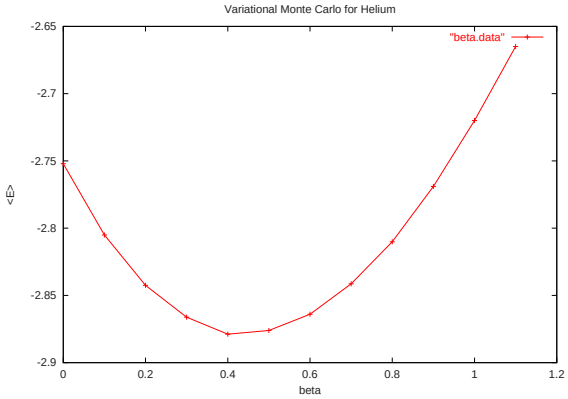
```
while (Z < 3.6) {                                                    162
    runMonteCarlo();                                                 163
    file << Z << '\t' << eAve << '\t' << eVar << '\n';               164
    cout << " Z = " << Z << "\t<E> = " << eAve                       165
         << "\tVariance = " << eVar << endl;                         166
    Z += 0.25;                                                       167
}                                                                    168
file.close();                                                        169

// Vary beta holding Z and alpha fixed                               171
file.open("beta.data");                                              172
Z = 2;                                                               173
beta = 0;                                                            174
cout << " Varying beta with Z = 2 and alpha = 0.1" << endl;          175
while (beta < 1.1) {                                                 176
    runMonteCarlo();                                                 177
    file << beta << '\t' << eAve << '\t' << eVar << '\n';            178
    cout << " beta = " << beta << "\t<E> = " << eAve                  179
         << "\tVariance = " << eVar << endl;                         180
    beta += 0.1;                                                     181
}                                                                    182
file.close();                                                        183

// Vary alpha holding Z and beta fixed                               185
file.open("alpha.data");                                             186
beta = 0.5;                                                          187
Z = 2;                                                               188
alpha = 0;                                                           189
cout << " Varying alpha with Z = 2 and beta = 0.5" << endl;          190
```

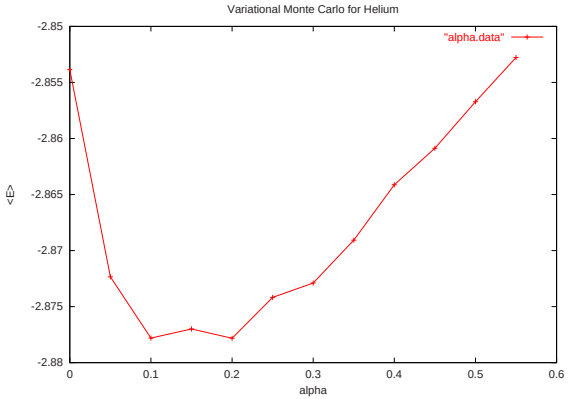
```
        while (alpha < 0.6) {                                                    191
            runMonteCarlo();                                                     192
            file << alpha << '\t' << eAve << '\t' << eVar << '\n';               193
            cout << " alpha = " << alpha << "\t<E> = " << eAve                   194
                 << "\tVariance = " << eVar << endl;                             195
            alpha += 0.05;                                                       196
        }                                                                        197
        file.close();                                                            198
    }                                                                            199
```

Output of the program

The following plots show the effects of varying Z with $\alpha = 0.1$ and $\beta = 0.5$, varying β with $Z = 2$ and $\alpha = 0.1$, and varying α with $Z = 2$ and $\beta = 0.5$, respectively.

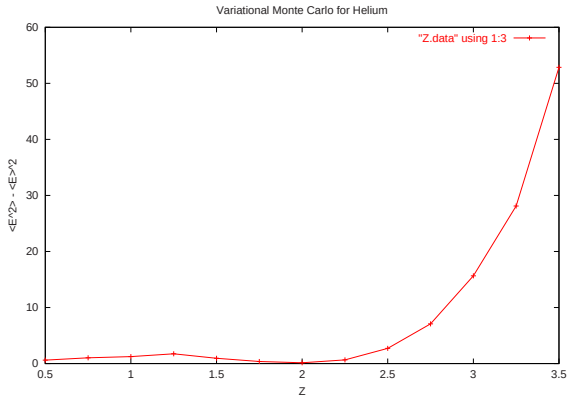


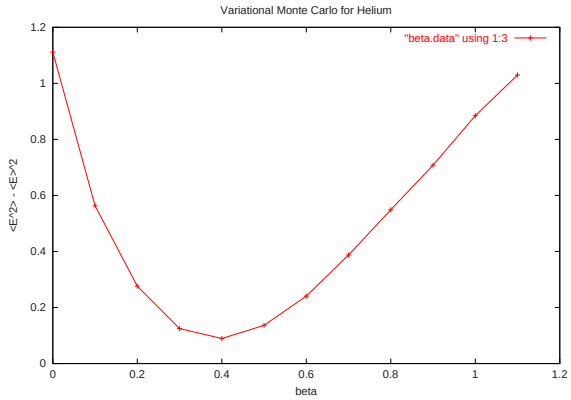


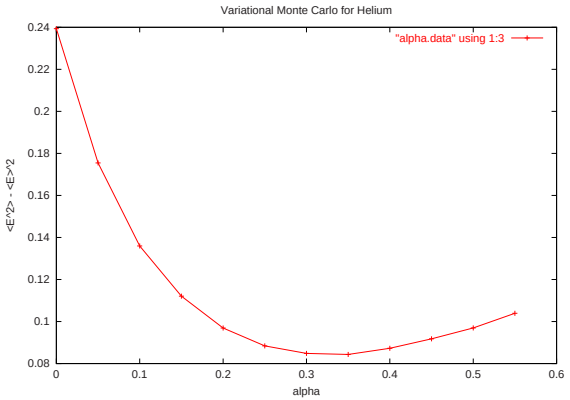


Note that the average energy is most sensitive to variations in the parameter Z: this shows that it is most important to use the cusp condition on the single particle trial functions. The β dependence of the average energy is not as dramatic as the Z dependence: this shows that the electron-electron 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 α 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 β and the relatively small dependence on α. Once again, this shows the importance of using the cusp conditions on Z and β: if this is done, one does not need to waste a lot of computer time searching for the minimum in α or trying to reduce the variance.