Variational Monte Carlo methods

Morten Hjorth-Jensen Email morten.hjorth-jensen@fys.uio.no[1,2]

[1]Department of Physics, University of Oslo, Norway
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University, USA

May 16-20 2016

Quantum Monte Carlo Motivation

Given a hamiltonian H and a trial wave function Ψ_T, the variational principle states that the expectation value of $\langle H \rangle$, defined through

$$E[H] = \langle H \rangle = \frac{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R})H(\boldsymbol{R})\Psi_T(\boldsymbol{R})}{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R})\Psi_T(\boldsymbol{R})},$$

is an upper bound to the ground state energy E_0 of the hamiltonian H, that is

$$E_0 \leq \langle H \rangle.$$

In general, the integrals involved in the calculation of various expectation values are multi-dimensional ones. Traditional integration methods such as the Gauss-Legendre will not be adequate for say the computation of the energy of a many-body system.

Quantum Monte Carlo Motivation

The trial wave function can be expanded in the eigenstates of the hamiltonian since they form a complete set, viz.,

$$\Psi_T(\boldsymbol{R}) = \sum_i a_i \Psi_i(\boldsymbol{R}),$$

and assuming the set of eigenfunctions to be normalized one obtains

$$\frac{\sum_{nm} a_m^* a_n \int d\boldsymbol{R}\Psi_m^*(\boldsymbol{R})H(\boldsymbol{R})\Psi_n(\boldsymbol{R})}{\sum_{nm} a_m^* a_n \int d\boldsymbol{R}\Psi_m^*(\boldsymbol{R})\Psi_n(\boldsymbol{R})} = \frac{\sum_n a_n^2 E_n}{\sum_n a_n^2} \geq E_0,$$

where we used that $H(\boldsymbol{R})\Psi_n(\boldsymbol{R}) = E_n\Psi_n(\boldsymbol{R})$. In general, the integrals involved in the calculation of various expectation values are multi-dimensional ones. The variational principle yields the lowest state of a given symmetry.

Quantum Monte Carlo Motivation

In most cases, a wave function has only small values in large parts of configuration space, and a straightforward procedure which uses homogenously distributed random points in configuration space will most likely lead to poor results. This may suggest that some kind of importance sampling combined with e.g., the Metropolis algorithm may be a more efficient way of obtaining the ground state energy. The hope is then that those regions of configurations space where the wave function assumes appreciable values are sampled more efficiently.

Quantum Monte Carlo Motivation

The tedious part in a VMC calculation is the search for the variational minimum. A good knowledge of the system is required in order to carry out reasonable VMC calculations. This is not always the case, and often VMC calculations serve rather as the starting point for so-called diffusion Monte Carlo calculations (DMC). DMC is a way of solving exactly the many-body Schroedinger equation by means of a stochastic procedure. A good guess on the binding energy and its wave function is however necessary. A carefully performed VMC calculation can aid in this context.

Quantum Monte Carlo Motivation

- Construct first a trial wave function $\psi_T(\boldsymbol{R},\boldsymbol{\alpha})$, for a many-body system consisting of N particles located at positions

$\boldsymbol{R} = (\boldsymbol{R}_1,\dots,\boldsymbol{R}_N)$. The trial wave function depends on α variational parameters $\boldsymbol{\alpha} = (\alpha_1,\dots,\alpha_M)$.

- Then we evaluate the expectation value of the hamiltonian H

$$E[H] = \langle H\rangle = \frac{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R},\boldsymbol{\alpha})H(\boldsymbol{R})\Psi_T(\boldsymbol{R},\boldsymbol{\alpha})}{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R},\boldsymbol{\alpha})\Psi_T(\boldsymbol{R},\boldsymbol{\alpha})}.$$

- Thereafter we vary α according to some minimization algorithm and return to the first step.

Quantum Monte Carlo Motivation

Basic steps. Choose a trial wave function $\psi_T(\boldsymbol{R})$.

$$P(\boldsymbol{R}) = \frac{|\psi_T(\boldsymbol{R})|^2}{\int |\psi_T(\boldsymbol{R})|^2\, d\boldsymbol{R}}.$$

This is our new probability distribution function (PDF). The approximation to the expectation value of the Hamiltonian is now

$$E[H(\boldsymbol{\alpha})] = \frac{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R},\boldsymbol{\alpha})H(\boldsymbol{R})\Psi_T(\boldsymbol{R},\boldsymbol{\alpha})}{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R},\boldsymbol{\alpha})\Psi_T(\boldsymbol{R},\boldsymbol{\alpha})}.$$

Quantum Monte Carlo Motivation

Define a new quantity

$$E_L(\boldsymbol{R},\boldsymbol{\alpha}) = \frac{1}{\psi_T(\boldsymbol{R},\boldsymbol{\alpha})} H\psi_T(\boldsymbol{R},\boldsymbol{\alpha}),$$

called the local energy, which, together with our trial PDF yields

$$E[H(\boldsymbol{\alpha})] = \int P(\boldsymbol{R})E_L(\boldsymbol{R})d\boldsymbol{R} \approx \frac{1}{N}\sum_{i=1}^{N} P(\boldsymbol{R_i},\boldsymbol{\alpha})E_L(\boldsymbol{R_i},\boldsymbol{\alpha})$$

with N being the number of Monte Carlo samples.

Quantum Monte Carlo

The Algorithm for performing a variational Monte Carlo calculations runs thus as this

- Initialisation: Fix the number of Monte Carlo steps. Choose an initial $\boldsymbol{R}$ and variational parameters α and calculate $|\psi_T^{\alpha}(\boldsymbol{R})|^2$.
- Initialise the energy and the variance and start the Monte Carlo calculation.
  - Calculate a trial position $\boldsymbol{R}_p = \boldsymbol{R} + r * step$ where r is a random variable $r \in [0,1]$.
  - Metropolis algorithm to accept or reject this move $w = P(\boldsymbol{R}_p)/P(\boldsymbol{R})$.
  - If the step is accepted, then we set $\boldsymbol{R} = \boldsymbol{R}_p$.
  - Update averages
- Finish and compute final averages.

Observe that the jumping in space is governed by the variable *step*. This is Called brute-force sampling. Need importance sampling to get more relevant sampling, see lectures below.

Quantum Monte Carlo: hydrogen atom

The radial Schroedinger equation for the hydrogen atom can be written as

$$-\frac{\hbar^2}{2m}\frac{\partial^2 u(r)}{\partial r^2} - \left(\frac{ke^2}{r} - \frac{\hbar^2 l(l+1)}{2mr^2}\right)u(r) = Eu(r),$$

or with dimensionless variables

$$-\frac{1}{2}\frac{\partial^2 u(\rho)}{\partial \rho^2} - \frac{u(\rho)}{\rho} + \frac{l(l+1)}{2\rho^2}u(\rho) - \lambda u(\rho) = 0,$$

with the hamiltonian

$$H = -\frac{1}{2}\frac{\partial^2}{\partial\rho^2} - \frac{1}{\rho} + \frac{l(l+1)}{2\rho^2}.$$

Use variational parameter α in the trial wave function

$$u_T^{\alpha}(\rho) = \alpha\rho e^{-\alpha\rho}.$$

Quantum Monte Carlo: hydrogen atom

Inserting this wave function into the expression for the local energy E_L gives

$$E_L(\rho) = -\frac{1}{\rho} - \frac{\alpha}{2}\left(\alpha - \frac{2}{\rho}\right).$$

A simple variational Monte Carlo calculation results in

α	$\langle H\rangle$	σ^2	$\sigma/\sqrt{N}$
7.00000E-01	-4.57759E-01	4.51201E-02	6.71715E-04
8.00000E-01	-4.81461E-01	3.05736E-02	5.52934E-04
9.00000E-01	-4.95899E-01	8.20497E-03	2.86443E-04
1.00000E-00	-5.00000E-01	0.00000E+00	0.00000E+00
1.10000E+00	-4.93738E-01	1.16989E-02	3.42036E-04
1.20000E+00	-4.75563E-01	8.85899E-02	9.41222E-04
1.30000E+00	-4.54341E-01	1.45171E-01	1.20487E-03

Quantum Monte Carlo: hydrogen atom

We note that at $\alpha = 1$ we obtain the exact result, and the variance is zero, as it should. The reason is that we then have the exact wave function, and the action of the hamiltionan on the wave function

$$H\psi = \text{constant} \times \psi,$$

yields just a constant. The integral which defines various expectation values involving moments of the hamiltonian becomes then

$$\langle H^n\rangle = \frac{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R})H^n(\boldsymbol{R})\Psi_T(\boldsymbol{R})}{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R})\Psi_T(\boldsymbol{R})} = \text{constant}\times\frac{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R})\Psi_T(\boldsymbol{R})}{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R})\Psi_T(\boldsymbol{R})} = \text{constant}.$$

This gives an important information: the exact wave function leads to zero variance! Variation is then performed by minimizing both the energy and the variance.

Quantum Monte Carlo for bosons

For boson in a harmonic oscillator-like trap we will use is a spherical (S) or an elliptical (E) harmonic trap in one, two and finally three dimensions, with the latter given by

$$V_{ext}(\mathbf{r}) = \begin{cases} \frac{1}{2}m\omega_{ho}^2 r^2 & (S) \\ \frac{1}{2}m[\omega_{ho}^2(x^2+y^2)+\omega_z^2 z^2] & (E) \end{cases} \tag{1}$$

where (S) stands for symmetric and

$$\hat{H} = \sum_i^N \left(\frac{-\hbar^2}{2m}\bigtriangledown_i^2 + V_{ext}(\mathbf{r}_i) \right) + \sum_{i<j}^N V_{int}(\mathbf{r}_i, \mathbf{r}_j), \tag{2}$$

as the two-body Hamiltonian of the system.

Quantum Monte Carlo for bosons

We will represent the inter-boson interaction by a pairwise, repulsive potential

$$V_{int}(|\mathbf{r}_i - \mathbf{r}_j|) = \begin{cases} \infty & |\mathbf{r}_i - \mathbf{r}_j| \leq a \\ 0 & |\mathbf{r}_i - \mathbf{r}_j| > a \end{cases} \tag{3}$$

where a is the so-called hard-core diameter of the bosons. Clearly, $V_{int}(|\mathbf{r}_i - \mathbf{r}_j|)$ is zero if the bosons are separated by a distance $|\mathbf{r}_i - \mathbf{r}_j|$ greater than a but infinite if they attempt to come within a distance $|\mathbf{r}_i - \mathbf{r}_j| \leq a$.

Quantum Monte Carlo for bosons

Our trial wave function for the ground state with N atoms is given by

$$\Psi_T(\mathbf{R}) = \Psi_T(\mathbf{r}_1, \mathbf{r}_2, \ldots \mathbf{r}_N, \alpha, \beta) = \prod_i g(\alpha, \beta, \mathbf{r}_i) \prod_{i<j} f(a, |\mathbf{r}_i - \mathbf{r}_j|), \tag{4}$$

where α and β are variational parameters. The single-particle wave function is proportional to the harmonic oscillator function for the ground state

$$g(\alpha, \beta, \mathbf{r}_i) = \exp\left[-\alpha(x_i^2 + y_i^2 + \beta z_i^2)\right]. \tag{5}$$

Quantum Monte Carlo for bosons

For spherical traps we have $\beta = 1$ and for non-interacting bosons ($a = 0$) we have $\alpha = 1/2a_{ho}^2$. The correlation wave function is

$$f(a, |\mathbf{r}_i - \mathbf{r}_j|) = \begin{cases} 0 & |\mathbf{r}_i - \mathbf{r}_j| \leq a \\ (1 - \frac{a}{|\mathbf{r}_i - \mathbf{r}_j|}) & |\mathbf{r}_i - \mathbf{r}_j| > a. \end{cases} \tag{6}$$

Quantum Monte Carlo: the helium atom

The helium atom consists of two electrons and a nucleus with charge $Z = 2$. The contribution to the potential energy due to the attraction from the nucleus is

$$-\frac{2ke^2}{r_1} - \frac{2ke^2}{r_2},$$

and if we add the repulsion arising from the two interacting electrons, we obtain the potential energy

$$V(r_1, r_2) = -\frac{2ke^2}{r_1} - \frac{2ke^2}{r_2} + \frac{ke^2}{r_{12}},$$

with the electrons separated at a distance $r_{12} = |\boldsymbol{r}_1 - \boldsymbol{r}_2|$.

Quantum Monte Carlo: the helium atom

The hamiltonian becomes then

$$\hat{H} = -\frac{\hbar^2\nabla_1^2}{2m} - \frac{\hbar^2\nabla_2^2}{2m} - \frac{2ke^2}{r_1} - \frac{2ke^2}{r_2} + \frac{ke^2}{r_{12}},$$

and Schroedingers equation reads

$$\hat{H}\psi = E\psi.$$

All observables are evaluated with respect to the probability distribution

$$P(\boldsymbol{R}) = \frac{|\psi_T(\boldsymbol{R})|^2}{\int |\psi_T(\boldsymbol{R})|^2 \, d\boldsymbol{R}}.$$

generated by the trial wave function. The trial wave function must approximate an exact eigenstate in order that accurate results are to be obtained.

Quantum Monte Carlo: the helium atom

Choice of trial wave function for Helium: Assume $r_1 \to 0$.

$$E_L(\boldsymbol{R}) = \frac{1}{\psi_T(\boldsymbol{R})} H\psi_T(\boldsymbol{R}) = \frac{1}{\psi_T(\boldsymbol{R})}\left(-\frac{1}{2}\nabla_1^2 - \frac{Z}{r_1}\right)\psi_T(\boldsymbol{R}) + \text{finite terms}.$$

$$E_L(R) = \frac{1}{\mathbf{R}_T(r_1)}\left(-\frac{1}{2}\frac{d^2}{dr_1^2} - \frac{1}{r_1}\frac{d}{dr_1} - \frac{Z}{r_1}\right)\mathbf{R}_T(r_1) + \text{finite terms}$$

For small values of r_1, the terms which dominate are

$$\lim_{r_1 \to 0} E_L(R) = \frac{1}{\mathbf{R}_T(r_1)}\left(-\frac{1}{r_1}\frac{d}{dr_1} - \frac{Z}{r_1}\right)\mathbf{R}_T(r_1),$$

since the second derivative does not diverge due to the finiteness of Ψ at the origin.

This results in

$$\frac{1}{\mathbf{R}_T(r_1)}\frac{d\mathbf{R}_T(r_1)}{dr_1} = -Z,$$

and

$$\mathbf{R}_T(r_1) \propto e^{-Zr_1}.$$

A similar condition applies to electron 2 as well. For orbital momenta $l > 0$ we have

$$\frac{1}{\mathbf{R}_T(r)}\frac{d\mathbf{R}_T(r)}{dr} = -\frac{Z}{l+1}.$$

Similarly, studying the case $r_{12} \to 0$ we can write a possible trial wave function as

$$\psi_T(\boldsymbol{R}) = e^{-\alpha(r_1+r_2)}e^{\beta r_{12}}.$$

The last equation can be generalized to

$$\psi_T(\boldsymbol{R}) = \phi(\boldsymbol{r}_1)\phi(\boldsymbol{r}_2)\dots\phi(\boldsymbol{r}_N)\prod_{i<j} f(r_{ij}),$$

for a system with N electrons or particles.

The first attempt at solving the helium atom

During the development of our code we need to make several checks. It is also very instructive to compute a closed form expression for the local energy. Since our wave function is rather simple it is straightforward to find an analytic expressions. Consider first the case of the simple helium function

$$\Psi_T(\boldsymbol{r}_1, \boldsymbol{r}_2) = e^{-\alpha(r_1+r_2)}$$

The local energy is for this case

$$E_{L1} = (\alpha - Z)\left(\frac{1}{r_1} + \frac{1}{r_2}\right) + \frac{1}{r_{12}} - \alpha^2$$

which gives an expectation value for the local energy given by

$$\langle E_{L1}\rangle = \alpha^2 - 2\alpha\left(Z - \frac{5}{16}\right)$$

The first attempt at solving the Helium atom

With closed form formulae we can speed up the computation of the correlation. In our case we write it as

$$\Psi_C = \exp\left\{\sum_{i<j}\frac{ar_{ij}}{1+\beta r_{ij}}\right\},$$

which means that the gradient needed for the so-called quantum force and local energy can be calculated analytically. This will speed up your code since the computation of the correlation part and the Slater determinant are the most time consuming parts in your code.

We will refer to this correlation function as Ψ_C or the *linear Pade-Jastrow*.

The first attempt at solving the Helium atom

We can test this by computing the local energy for our helium wave function

$$\psi_T(\boldsymbol{r}_1,\boldsymbol{r}_2) = \exp\left(-\alpha(r_1+r_2)\right)\exp\left(\frac{r_{12}}{2(1+\beta r_{12})}\right),$$

with α and β as variational parameters.

The local energy is for this case

$$E_{L2} = E_{L1} + \frac{1}{2(1+\beta r_{12})^2}\left\{\frac{\alpha(r_1+r_2)}{r_{12}}(1-\frac{\boldsymbol{r}_1\boldsymbol{r}_2}{r_1r_2}) - \frac{1}{2(1+\beta r_{12})^2} - \frac{2}{r_{12}} + \frac{2\beta}{1+\beta r_{12}}\right\}$$

It is very useful to test your code against these expressions. It means also that you don't need to compute a derivative numerically as discussed in the code example below.

The first attempt at solving the Helium atom

For the computation of various derivatives with different types of wave functions, you will find it useful to use python with symbolic python, that is sympy, see online manual. Using sympy allows you autogenerate both Latex code as well c++, python or Fortran codes. Here you will find some simple examples. We choose the $2s$ hydrogen-orbital (not normalized) as an example

$$\phi_{2s}(\boldsymbol{r}) = (Zr-2)\exp -(\frac{1}{2}Zr),$$

with $r^2 = x^2+y^2+z^2$.

```
from sympy import symbols, diff, exp, sqrt
x, y, z, Z = symbols('x y z Z')
r = sqrt(x*x + y*y + z*z)
r
phi = (Z*r - 2)*exp(-Z*r/2)
phi
diff(phi, x)
```

This doesn't look very nice, but sympy provides several functions that allow for improving and simplifying the output.

The first attempt at solving the Helium atom

We can improve our output by factorizing and substituting expressions

```
from sympy import symbols, diff, exp, sqrt, factor, Symbol, printing
x, y, z, Z = symbols('x y z Z')
r = sqrt(x*x + y*y + z*z)
phi = (Z*r - 2)*exp(-Z*r/2)
R = Symbol('r') #Creates a symbolic equivalent of r
#print latex and c++ code
print printing.latex(diff(phi, x).factor().subs(r, R))
print printing.ccode(diff(phi, x).factor().subs(r, R))
```

The first attempt at solving the Helium atom

We can in turn look at second derivatives

```
from sympy import symbols, diff, exp, sqrt, factor, Symbol, printing
x, y, z, Z = symbols('x y z Z')
r = sqrt(x*x + y*y + z*z)
phi = (Z*r - 2)*exp(-Z*r/2)
R = Symbol('r') #Creates a symbolic equivalent of r
(diff(diff(phi, x), x) + diff(diff(phi, y), y) + diff(diff(phi, z), z)).factor().subs(r, R)
# Collect the Z values
(diff(diff(phi, x), x) + diff(diff(phi, y), y) +diff(diff(phi, z), z)).factor().collect(Z).subs(r,
# Factorize also the r**2 terms
(diff(diff(phi, x), x) + diff(diff(phi, y), y) + diff(diff(phi, z), z)).factor().collect(Z).subs(r
print printing.ccode((diff(diff(phi, x), x) + diff(diff(phi, y), y) + diff(diff(phi, z), z)).facto
```

With some practice this allows one to be able to check one's own calculation and translate automatically into code lines.

The first attempt at solving the Helium atom

The c++ code with a VMC Solver class, main program first.

```
#include "vmcsolver.h"
#include <iostream>
using namespace std;

int main()
{
    VMCSolver *solver = new VMCSolver();
    solver->runMonteCarloIntegration();
    return 0;
}
```

The first attempt at solving the Helium atom

The c++ code with a VMC Solver class, the VMCSolver header file.

```
#ifndef VMCSOLVER_H
#define VMCSOLVER_H
#include <armadillo>
using namespace arma;
class VMCSolver
{
```

```
public:
    VMCSolver();
    void runMonteCarloIntegration();

private:
    double waveFunction(const mat &r);
    double localEnergy(const mat &r);
    int nDimensions;
    int charge;
    double stepLength;
    int nParticles;
    double h;
    double h2;
    long idum;
    double alpha;
    int nCycles;
    mat rOld;
    mat rNew;
};
#endif // VMCSOLVER_H
```

The first attempt at solving the Helium atom

The c++ code with a VMC Solver class, VMCSolver codes, initialize.

```
#include "vmcsolver.h"
#include "lib.h"
#include <armadillo>
#include <iostream>
using namespace arma;
using namespace std;

VMCSolver::VMCSolver() :
    nDimensions(3),
    charge(2),
    stepLength(1.0),
    nParticles(2),
    h(0.001),
    h2(1000000),
    idum(-1),
    alpha(0.5*charge),
    nCycles(1000000)
{
}
```

The first attempt at solving the Helium atom

The c++ code with a VMC Solver class, VMCSolver codes.

```
void VMCSolver::runMonteCarloIntegration()
{
    rOld = zeros<mat>(nParticles, nDimensions);
    rNew = zeros<mat>(nParticles, nDimensions);
    double waveFunctionOld = 0;
    double waveFunctionNew = 0;
    double energySum = 0;
    double energySquaredSum = 0;
    double deltaE;
    // initial trial positions
```

```
    for(int i = 0; i < nParticles; i++) {
        for(int j = 0; j < nDimensions; j++) {
            rOld(i,j) = stepLength * (ran2(&idum) - 0.5);
        }
    }
    rNew = rOld;
    // loop over Monte Carlo cycles
    for(int cycle = 0; cycle < nCycles; cycle++) {
        // Store the current value of the wave function
        waveFunctionOld = waveFunction(rOld);
        // New position to test
        for(int i = 0; i < nParticles; i++) {
            for(int j = 0; j < nDimensions; j++) {
                rNew(i,j) = rOld(i,j) + stepLength*(ran2(&idum) - 0.5);
            }
            // Recalculate the value of the wave function
            waveFunctionNew = waveFunction(rNew);
            // Check for step acceptance (if yes, update position, if no, reset position)
            if(ran2(&idum) <= (waveFunctionNew*waveFunctionNew) / (waveFunctionOld*waveFunctionOld
                for(int j = 0; j < nDimensions; j++) {
                    rOld(i,j) = rNew(i,j);
                    waveFunctionOld = waveFunctionNew;
                }
            } else {
                for(int j = 0; j < nDimensions; j++) {
                    rNew(i,j) = rOld(i,j);
                }
            }
            // update energies
            deltaE = localEnergy(rNew);
            energySum += deltaE;
            energySquaredSum += deltaE*deltaE;
        }
    }
    double energy = energySum/(nCycles * nParticles);
    double energySquared = energySquaredSum/(nCycles * nParticles);
    cout << "Energy: " << energy << " Energy (squared sum): " << energySquared << endl;
}
```

The first attempt at solving the Helium atom

The c++ code with a VMC Solver class, VMCSolver codes.

```
double VMCSolver::localEnergy(const mat &r)
{
    mat rPlus = zeros<mat>(nParticles, nDimensions);
    mat rMinus = zeros<mat>(nParticles, nDimensions);
    rPlus = rMinus = r;
    double waveFunctionMinus = 0;
    double waveFunctionPlus = 0;
    double waveFunctionCurrent = waveFunction(r);
    // Kinetic energy, brute force derivations
    double kineticEnergy = 0;
    for(int i = 0; i < nParticles; i++) {
        for(int j = 0; j < nDimensions; j++) {
            rPlus(i,j) += h;
            rMinus(i,j) -= h;
            waveFunctionMinus = waveFunction(rMinus);
            waveFunctionPlus = waveFunction(rPlus);
            kineticEnergy -= (waveFunctionMinus + waveFunctionPlus - 2 * waveFunctionCurrent);
```

```
            rPlus(i,j) = r(i,j);
            rMinus(i,j) = r(i,j);
        }
    }
    kineticEnergy = 0.5 * h2 * kineticEnergy / waveFunctionCurrent;
    // Potential energy
    double potentialEnergy = 0;
    double rSingleParticle = 0;
    for(int i = 0; i < nParticles; i++) {
        rSingleParticle = 0;
        for(int j = 0; j < nDimensions; j++) {
            rSingleParticle += r(i,j)*r(i,j);
        }
        potentialEnergy -= charge / sqrt(rSingleParticle);
    }
    // Contribution from electron-electron potential
    double r12 = 0;
    for(int i = 0; i < nParticles; i++) {
        for(int j = i + 1; j < nParticles; j++) {
            r12 = 0;
            for(int k = 0; k < nDimensions; k++) {
                r12 += (r(i,k) - r(j,k)) * (r(i,k) - r(j,k));
            }
            potentialEnergy += 1 / sqrt(r12);
        }
    }
    return kineticEnergy + potentialEnergy;
}
```

The first attempt at solving the Helium atom

The c++ code with a VMC Solver class, VMCSolver codes.

```
double VMCSolver::waveFunction(const mat &r)
{
    double argument = 0;
    for(int i = 0; i < nParticles; i++) {
        double rSingleParticle = 0;
        for(int j = 0; j < nDimensions; j++) {
            rSingleParticle += r(i,j) * r(i,j);
        }
        argument += sqrt(rSingleParticle);
    }
    return exp(-argument * alpha);
}
```

The first attempt at solving the Helium atom

The c++ code with a VMC Solver class, the VMCSolver header file.

```
#include <armadillo>
#include <iostream>
using namespace arma;
using namespace std;
double ran2(long *);

class VMCSolver
{
public:
```

```
    VMCSolver();
    void runMonteCarloIntegration();

private:
    double waveFunction(const mat &r);
    double localEnergy(const mat &r);
    int nDimensions;
    int charge;
    double stepLength;
    int nParticles;
    double h;
    double h2;
    long idum;
    double alpha;
    int nCycles;
    mat rOld;
    mat rNew;
};

VMCSolver::VMCSolver() :
    nDimensions(3),
    charge(2),
    stepLength(1.0),
    nParticles(2),
    h(0.001),
    h2(1000000),
    idum(-1),
    alpha(0.5*charge),
    nCycles(1000000)
{
}

void VMCSolver::runMonteCarloIntegration()
{
    rOld = zeros<mat>(nParticles, nDimensions);
    rNew = zeros<mat>(nParticles, nDimensions);
    double waveFunctionOld = 0;
    double waveFunctionNew = 0;
    double energySum = 0;
    double energySquaredSum = 0;
    double deltaE;
    // initial trial positions
    for(int i = 0; i < nParticles; i++) {
        for(int j = 0; j < nDimensions; j++) {
            rOld(i,j) = stepLength * (ran2(&idum) - 0.5);
        }
    }
    rNew = rOld;
    // loop over Monte Carlo cycles
    for(int cycle = 0; cycle < nCycles; cycle++) {
        // Store the current value of the wave function
        waveFunctionOld = waveFunction(rOld);
        // New position to test
        for(int i = 0; i < nParticles; i++) {
            for(int j = 0; j < nDimensions; j++) {
                rNew(i,j) = rOld(i,j) + stepLength*(ran2(&idum) - 0.5);
            }
            // Recalculate the value of the wave function
            waveFunctionNew = waveFunction(rNew);
            // Check for step acceptance (if yes, update position, if no, reset position)
            if(ran2(&idum) <= (waveFunctionNew*waveFunctionNew) / (waveFunctionOld*waveFunctionOld
```

```
                for(int j = 0; j < nDimensions; j++) {
                    rOld(i,j) = rNew(i,j);
                    waveFunctionOld = waveFunctionNew;
                }
            } else {
                for(int j = 0; j < nDimensions; j++) {
                    rNew(i,j) = rOld(i,j);
                }
            }
            // update energies
            deltaE = localEnergy(rNew);
            energySum += deltaE;
            energySquaredSum += deltaE*deltaE;
        }
    }
    double energy = energySum/(nCycles * nParticles);
    double energySquared = energySquaredSum/(nCycles * nParticles);
    cout << "Energy: " << energy << " Energy (squared sum): " << energySquared << endl;
}

double VMCSolver::localEnergy(const mat &r)
{
    mat rPlus = zeros<mat>(nParticles, nDimensions);
    mat rMinus = zeros<mat>(nParticles, nDimensions);
    rPlus = rMinus = r;
    double waveFunctionMinus = 0;
    double waveFunctionPlus = 0;
    double waveFunctionCurrent = waveFunction(r);
    // Kinetic energy, brute force derivations
    double kineticEnergy = 0;
    for(int i = 0; i < nParticles; i++) {
        for(int j = 0; j < nDimensions; j++) {
            rPlus(i,j) += h;
            rMinus(i,j) -= h;
            waveFunctionMinus = waveFunction(rMinus);
            waveFunctionPlus = waveFunction(rPlus);
            kineticEnergy -= (waveFunctionMinus + waveFunctionPlus - 2 * waveFunctionCurrent);
            rPlus(i,j) = r(i,j);
            rMinus(i,j) = r(i,j);
        }
    }
    kineticEnergy = 0.5 * h2 * kineticEnergy / waveFunctionCurrent;
    // Potential energy
    double potentialEnergy = 0;
    double rSingleParticle = 0;
    for(int i = 0; i < nParticles; i++) {
        rSingleParticle = 0;
        for(int j = 0; j < nDimensions; j++) {
            rSingleParticle += r(i,j)*r(i,j);
        }
        potentialEnergy -= charge / sqrt(rSingleParticle);
    }
    // Contribution from electron-electron potential
    double r12 = 0;
    for(int i = 0; i < nParticles; i++) {
        for(int j = i + 1; j < nParticles; j++) {
            r12 = 0;
            for(int k = 0; k < nDimensions; k++) {
                r12 += (r(i,k) - r(j,k)) * (r(i,k) - r(j,k));
            }
            potentialEnergy += 1 / sqrt(r12);
```

```
        }
    }
    return kineticEnergy + potentialEnergy;
}

double VMCSolver::waveFunction(const mat &r)
{
    double argument = 0;
    for(int i = 0; i < nParticles; i++) {
        double rSingleParticle = 0;
        for(int j = 0; j < nDimensions; j++) {
            rSingleParticle += r(i,j) * r(i,j);
        }
        argument += sqrt(rSingleParticle);
    }
    return exp(-argument * alpha);
}

/*
** The function
**         ran2()
** is a long periode (> 2 x 10^18) random number generator of
** L'Ecuyer and Bays-Durham shuffle and added safeguards.
** Call with idum a negative integer to initialize; thereafter,
** do not alter idum between sucessive deviates in a
** sequence. RNMX should approximate the largest floating point value
** that is less than 1.
** The function returns a uniform deviate between 0.0 and 1.0
** (exclusive of end-point values).
*/

#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)

double ran2(long *idum)
{
  int             j;
  long            k;
  static long     idum2 = 123456789;
  static long     iy=0;
  static long     iv[NTAB];
  double          temp;

  if(*idum <= 0) {
    if(-(*idum) < 1) *idum = 1;
    else             *idum = -(*idum);
    idum2 = (*idum);
    for(j = NTAB + 7; j >= 0; j--) {
      k     = (*idum)/IQ1;
```

```
      *idum = IA1*(*idum - k*IQ1) - k*IR1;
      if(*idum < 0) *idum +=  IM1;
      if(j < NTAB)  iv[j]  = *idum;
    }
    iy=iv[0];
  }
  k     = (*idum)/IQ1;
  *idum = IA1*(*idum - k*IQ1) - k*IR1;
  if(*idum < 0) *idum += IM1;
  k     = idum2/IQ2;
  idum2 = IA2*(idum2 - k*IQ2) - k*IR2;
  if(idum2 < 0) idum2 += IM2;
  j     = iy/NDIV;
  iy    = iv[j] - idum2;
  iv[j] = *idum;
  if(iy < 1) iy += IMM1;
  if((temp = AM*iy) > RNMX) return RNMX;
  else return temp;
}
#undef IM1
#undef IM2
#undef AM
#undef IMM1
#undef IA1
#undef IA2
#undef IQ1
#undef IQ2
#undef IR1
#undef IR2
#undef NTAB
#undef NDIV
#undef EPS
#undef RNMX

// End: function ran2()


#include <iostream>
using namespace std;

int main()
{
    VMCSolver *solver = new VMCSolver();
    solver->runMonteCarloIntegration();
    return 0;
}
```

The python program for the two-electron quantum dot problem

```
#VMC for electrons in harmonic oscillator potentials with oscillator
#frequency = 1
#"Computational Physics", Morten Hjorth-Jensen

import numpy
import math
import sys
from random import random
```

```
#Read name of output file from command line
if len(sys.argv) == 2:
    outfilename = sys.argv[1]
else:
    print '\nError: Name of output file must be given as command line argument.\n'

#Initialisation function
def initialize():
    number_particles = eval(raw_input('Number of particles: '))
    dimension = eval(raw_input('Dimensionality: '))
    max_variations = eval(raw_input('Number of variational parameter values: '))
    number_cycles = eval(raw_input('Number of MC cycles: '))
    step_length = eval(raw_input('Step length: '))

    return number_particles,dimension,max_variations,number_cycles,step_length

#Trial wave function
def wave_function(r):
    argument = 0.0
    for i in xrange(number_particles):
        r_single_particle = 0.0
        for j in xrange(dimension):
            r_single_particle += r[i,j]**2
        argument += r_single_particle
    wf = math.exp(-argument*alpha*0.5)
    #Jastrow factor
    for i1 in xrange(number_particles-1):
        for i2 in xrange(i1+1,number_particles):
            r_12 = 0.0
            for k in xrange(dimension):
                r_12 += (r[i1,k] - r[i2,k])**2
            argument = math.sqrt(r_12)
            wf *= math.exp(argument/(1.0+0.3*argument))

    return wf

#Local energy (numerical derivative)
#the argument wf is the wave function value at r (so we don't need to calculate it again)
def local_energy(r,wf):
    #Kinetic energy
    r_plus = r.copy()
    r_minus = r.copy()
    e_kinetic = 0.0
    for i in xrange(number_particles):
        for j in xrange(dimension):
            r_plus[i,j] = r[i,j] + h
            r_minus[i,j] = r[i,j] - h
            wf_minus = wave_function(r_minus)
            wf_plus = wave_function(r_plus)
            e_kinetic -= wf_minus+wf_plus-2*wf;
            r_plus[i,j] = r[i,j]
            r_minus[i,j] = r[i,j]
    e_kinetic = .5*h2*e_kinetic/wf
    #Potential energy
    e_potential = 0.0

    #harmonic oscillator  contribution
    for i in xrange(number_particles):
        r_single_particle = 0.0
        for j in xrange(dimension):
            r_single_particle += r[i,j]**2
```

```
            e_potential += 0.5*r_single_particle

        #Electron-electron contribution
        for i1 in xrange(number_particles-1):
            for i2 in xrange(i1+1,number_particles):
                r_12 = 0.0
                for j in xrange(dimension):
                    r_12 += (r[i1,j] - r[i2,j])**2
                e_potential += 1/math.sqrt(r_12)

        return e_potential + e_kinetic


    #Here starts the main program

    number_particles,dimension,max_variations,number_cycles,step_length = initialize()

    outfile = open(outfilename,'w')

    alpha = 0.5  #variational parameter

    #Step length for numerical differentiation and its inverse squared
    h = .001
    h2 = 1/(h**2)

    r_old = numpy.zeros((number_particles,dimension), numpy.double)
    r_new = numpy.zeros((number_particles,dimension), numpy.double)

    #Loop over alpha values
    for variate in xrange(max_variations):

        alpha += .1
        energy = energy2 = 0.0
        accept = 0.0
        delta_e = 0.0

        #Initial position
        for i in xrange(number_particles):
            for j in xrange(dimension):
                r_old[i,j] = step_length * (random() - .5)

        wfold = wave_function(r_old)

        #Loop over MC cycles
        for cycle in xrange(number_cycles):

            #Trial position
            for i in xrange(number_particles):
                for j in xrange(dimension):
                    r_new[i,j] = r_old[i,j] + step_length * (random() - .5)

            wfnew = wave_function(r_new)

            #Metropolis test to see whether we accept the move
            if random() < wfnew**2 / wfold**2:
                r_old = r_new.copy()
                wfold = wfnew
                accept += 1
            #update expectation values
            delta_e = local_energy(r_old,wfold)
            energy += delta_e
```

```
            energy2 += delta_e**2

        #We calculate mean, variance and error ...
        energy /= number_cycles
        energy2 /= number_cycles
        variance = energy2 - energy**2
        error = math.sqrt(variance/number_cycles)

        #...and write them to file
        outfile.write('%f %f %f %f %f\n' %(alpha,energy,variance,error,accept*1.0/(number_cycles)))

    outfile.close()

    print('\nDone. Results are in the file "%s", formatted as:\n\
    alpha, <energy>, variance, error, acceptance ratio' %(outfilename))
```

The Metropolis algorithm

The Metropolis algorithm , see the original article (see also the FYS3150 lectures) was invented by Metropolis et. al and is often simply called the Metropolis algorithm. It is a method to sample a normalized probability distribution by a stochastic process. We define $\mathbf{P}_i^{(n)}$ to be the probability for finding the system in the state i at step n. The algorithm is then

- Sample a possible new state j with some probability $T_{i\rightarrow j}$.
- Accept the new state j with probability $A_{i\rightarrow j}$ and use it as the next sample. With probability $1-A_{i\rightarrow j}$ the move is rejected and the original state i is used again as a sample.

The Metropolis algorithm

We wish to derive the required properties of T and A such that $\mathbf{P}_i^{(n\rightarrow\infty)}\rightarrow p_i$ so that starting from any distribution, the method converges to the correct distribution. Note that the description here is for a discrete probability distribution. Replacing probabilities p_i with expressions like $p(x_i)dx_i$ will take all of these over to the corresponding continuum expressions.

The Metropolis algorithm

The dynamical equation for $\mathbf{P}_i^{(n)}$ can be written directly from the description above. The probability of being in the state i at step n is given by the probability of being in any state j at the previous step, and making an accepted transition to i added to the probability of being in the state i, making a transition to any state j and rejecting the move:

$$\mathbf{P}_i^{(n)} = \sum_j \left[\mathbf{P}_j^{(n-1)}T_{j\rightarrow i}A_{j\rightarrow i} + \mathbf{P}_i^{(n-1)}T_{i\rightarrow j}\left(1-A_{i\rightarrow j}\right)\right] .$$

Since the probability of making some transition must be 1, $\sum_j T_{i\to j} = 1$, and the above equation becomes

$$\mathbf{P}_i^{(n)} = \mathbf{P}_i^{(n-1)} + \sum_j \left[\mathbf{P}_j^{(n-1)} T_{j\to i} A_{j\to i} - \mathbf{P}_i^{(n-1)} T_{i\to j} A_{i\to j}\right] .$$

The Metropolis algorithm

For large n we require that $\mathbf{P}_i^{(n\to\infty)} = p_i$, the desired probability distribution. Taking this limit, gives the balance requirement

$$\sum_j \left[p_j T_{j\to i} A_{j\to i} - p_i T_{i\to j} A_{i\to j}\right] = 0 .$$

The balance requirement is very weak. Typically the much stronger detailed balance requirement is enforced, that is rather than the sum being set to zero, we set each term separately to zero and use this to determine the acceptance probabilities. Rearranging, the result is

$$\frac{A_{j\to i}}{A_{i\to j}} = \frac{p_i T_{i\to j}}{p_j T_{j\to i}} .$$

The Metropolis algorithm

The Metropolis choice is to maximize the A values, that is

$$A_{j\to i} = \min\left(1, \frac{p_i T_{i\to j}}{p_j T_{j\to i}}\right) .$$

Other choices are possible, but they all correspond to multilplying $A_{i\to j}$ and $A_{j\to i}$ by the same constant smaller than unity.[1]

The Metropolis algorithm

Having chosen the acceptance probabilities, we have guaranteed that if the $\mathbf{P}_i^{(n)}$ has equilibrated, that is if it is equal to p_i, it will remain equilibrated. Next we need to find the circumstances for convergence to equilibrium.

The dynamical equation can be written as

$$\mathbf{P}_i^{(n)} = \sum_j M_{ij} \mathbf{P}_j^{(n-1)}$$

with the matrix M given by

$$M_{ij} = \delta_{ij}\left[1 - \sum_k T_{i\to k} A_{i\to k}\right] + T_{j\to i} A_{j\to i} .$$

[1] The penalty function method uses just such a factor to compensate for p_i that are evaluated stochastically and are therefore noisy.

Summing over i shows that $\sum_i M_{ij} = 1$, and since $\sum_k T_{i\to k} = 1$, and $A_{i\to k} \leq 1$, the elements of the matrix satisfy $M_{ij} \geq 0$. The matrix M is therefore a stochastic matrix.

The Metropolis algorithm

The Metropolis method is simply the power method for computing the right eigenvector of M with the largest magnitude eigenvalue. By construction, the correct probability distribution is a right eigenvector with eigenvalue 1. Therefore, for the Metropolis method to converge to this result, we must show that M has only one eigenvalue with this magnitude, and all other eigenvalues are smaller.

Importance sampling

We need to replace the brute force Metropolis algorithm with a walk in coordinate space biased by the trial wave function. This approach is based on the Fokker-Planck equation and the Langevin equation for generating a trajectory in coordinate space. The link between the Fokker-Planck equation and the Langevin equations are explained, only partly, in the slides below. An excellent reference on topics like Brownian motion, Markov chains, the Fokker-Planck equation and the Langevin equation is the text by Van Kampen Here we will focus first on the implementation part first.

For a diffusion process characterized by a time-dependent probability density $P(x,t)$ in one dimension the Fokker-Planck equation reads (for one particle /walker)

$$\frac{\partial P}{\partial t} = D\frac{\partial}{\partial x}\left(\frac{\partial}{\partial x} - F\right)P(x,t),$$

where F is a drift term and D is the diffusion coefficient.

Importance sampling

The new positions in coordinate space are given as the solutions of the Langevin equation using Euler's method, namely, we go from the Langevin equation

$$\frac{\partial x(t)}{\partial t} = DF(x(t)) + \eta,$$

with η a random variable, yielding a new position

$$y = x + DF(x)\Delta t + \xi\sqrt{\Delta t},$$

where ξ is gaussian random variable and Δt is a chosen time step. The quantity D is, in atomic units, equal to $1/2$ and comes from the factor $1/2$ in the kinetic energy operator. Note that Δt is to be viewed as a parameter. Values of $\Delta t \in [0.001, 0.01]$ yield in general rather stable values of the ground state energy.

Importance sampling

The process of isotropic diffusion characterized by a time-dependent probability density $P(\mathbf{x}, t)$ obeys (as an approximation) the so-called Fokker-Planck equation

$$\frac{\partial P}{\partial t} = \sum_i D \frac{\partial}{\partial \mathbf{x_i}} \left(\frac{\partial}{\partial \mathbf{x_i}} - \mathbf{F_i} \right) P(\mathbf{x}, t),$$

where $\mathbf{F_i}$ is the i^{th} component of the drift term (drift velocity) caused by an external potential, and D is the diffusion coefficient. The convergence to a stationary probability density can be obtained by setting the left hand side to zero. The resulting equation will be satisfied if and only if all the terms of the sum are equal zero,

$$\frac{\partial^2 P}{\partial \mathbf{x_i}^2} = P \frac{\partial}{\partial \mathbf{x_i}} \mathbf{F_i} + \mathbf{F_i} \frac{\partial}{\partial \mathbf{x_i}} P.$$

Importance sampling

The drift vector should be of the form $\mathbf{F} = g(\mathbf{x}) \frac{\partial P}{\partial \mathbf{x}}$. Then,

$$\frac{\partial^2 P}{\partial \mathbf{x_i}^2} = P \frac{\partial g}{\partial P} \left(\frac{\partial P}{\partial \mathbf{x}_i} \right)^2 + P g \frac{\partial^2 P}{\partial \mathbf{x}_i^2} + g \left(\frac{\partial P}{\partial \mathbf{x}_i} \right)^2.$$

The condition of stationary density means that the left hand side equals zero. In other words, the terms containing first and second derivatives have to cancel each other. It is possible only if $g = \frac{1}{P}$, which yields

$$\mathbf{F} = 2 \frac{1}{\Psi_T} \nabla \Psi_T,$$

which is known as the so-called *quantum force*. This term is responsible for pushing the walker towards regions of configuration space where the trial wave function is large, increasing the efficiency of the simulation in contrast to the Metropolis algorithm where the walker has the same probability of moving in every direction.

Importance sampling

The Fokker-Planck equation yields a (the solution to the equation) transition probability given by the Green's function

$$G(y, x, \Delta t) = \frac{1}{(4\pi D \Delta t)^{3N/2}} \exp\left(-(y - x - D\Delta t F(x))^2 / 4D\Delta t\right)$$

which in turn means that our brute force Metropolis algorithm

$$A(y, x) = \min(1, q(y, x))),$$

with $q(y, x) = |\Psi_T(y)|^2 / |\Psi_T(x)|^2$ is now replaced by the Metropolis-Hastings algorithm as well as Hasting's article,

$$q(y, x) = \frac{G(x, y, \Delta t) |\Psi_T(y)|^2}{G(y, x, \Delta t) |\Psi_T(x)|^2}$$

Importance sampling, program elements

The full code is this link. Here we include only the parts pertaining to the computation of the quantum force and the Metropolis update. The program is a modfication of our previous c++ program discussed previously. Here we display only the part from the *vmcsolver.cpp* file. Note the usage of the function *GaussianDeviate*.

```
void VMCSolver::runMonteCarloIntegration()
{
  rOld = zeros<mat>(nParticles, nDimensions);
  rNew = zeros<mat>(nParticles, nDimensions);
  QForceOld = zeros<mat>(nParticles, nDimensions);
  QForceNew = zeros<mat>(nParticles, nDimensions);

  double waveFunctionOld = 0;
  double waveFunctionNew = 0;

  double energySum = 0;
  double energySquaredSum = 0;

  double deltaE;

  // initial trial positions
  for(int i = 0; i < nParticles; i++) {
    for(int j = 0; j < nDimensions; j++) {
      rOld(i,j) = GaussianDeviate(&idum)*sqrt(timestep);
    }
  }
  rNew = rOld;
```

Importance sampling, program elements

```
  for(int cycle = 0; cycle < nCycles; cycle++) {

    // Store the current value of the wave function
    waveFunctionOld = waveFunction(rOld);
    QuantumForce(rOld, QForceOld); QForceOld = QForceOld*h/waveFunctionOld;
    // New position to test
    for(int i = 0; i < nParticles; i++) {
      for(int j = 0; j < nDimensions; j++) {
        rNew(i,j) = rOld(i,j) + GaussianDeviate(&idum)*sqrt(timestep)+QForceOld(i,j)*timestep*D;
      }
      //  for the other particles we need to set the position to the old position since
      //  we move only one particle at the time
      for (int k = 0; k < nParticles; k++) {
        if ( k != i) {
          for (int j=0; j < nDimensions; j++) {
            rNew(k,j) = rOld(k,j);
          }
        }
      }
```

Importance sampling, program elements

```
// loop over Monte Carlo cycles
    // Recalculate the value of the wave function and the quantum force
    waveFunctionNew = waveFunction(rNew);
    QuantumForce(rNew,QForceNew) = QForceNew*h/waveFunctionNew;
    //  we compute the log of the ratio of the greens functions to be used in the
    //  Metropolis-Hastings algorithm
    GreensFunction = 0.0;
    for (int j=0; j < nDimensions; j++) {
      GreensFunction += 0.5*(QForceOld(i,j)+QForceNew(i,j))*
        (D*timestep*0.5*(QForceOld(i,j)-QForceNew(i,j))-rNew(i,j)+rOld(i,j));
    }
    GreensFunction = exp(GreensFunction);

    // The Metropolis test is performed by moving one particle at the time
    if(ran2(&idum) <= GreensFunction*(waveFunctionNew*waveFunctionNew) / (waveFunctionOld*waveFu
      for(int j = 0; j < nDimensions; j++) {
        rOld(i,j) = rNew(i,j);
        QForceOld(i,j) = QForceNew(i,j);
        waveFunctionOld = waveFunctionNew;
      }
    } else {
      for(int j = 0; j < nDimensions; j++) {
        rNew(i,j) = rOld(i,j);
        QForceNew(i,j) = QForceOld(i,j);
      }
    }
```

Importance sampling, program elements

Note numerical derivatives.

```
double VMCSolver::QuantumForce(const mat &r, mat &QForce)
{
    mat rPlus = zeros<mat>(nParticles, nDimensions);
    mat rMinus = zeros<mat>(nParticles, nDimensions);
    rPlus = rMinus = r;
    double waveFunctionMinus = 0;
    double waveFunctionPlus = 0;
    double waveFunctionCurrent = waveFunction(r);

    // Kinetic energy

    double kineticEnergy = 0;
    for(int i = 0; i < nParticles; i++) {
        for(int j = 0; j < nDimensions; j++) {
            rPlus(i,j) += h;
            rMinus(i,j) -= h;
            waveFunctionMinus = waveFunction(rMinus);
            waveFunctionPlus = waveFunction(rPlus);
            QForce(i,j) =  (waveFunctionPlus-waveFunctionMinus);
            rPlus(i,j) = r(i,j);
            rMinus(i,j) = r(i,j);
        }
    }
}
```

Importance sampling, program elements

The general derivative formula of the Jastrow factor is (the subscript C stands for Correlation)

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} + \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_k}$$

However, with our written in way which can be reused later as

$$\Psi_C = \prod_{i<j} g(r_{ij}) = \exp\left\{\sum_{i<j} f(r_{ij})\right\},$$

the gradient needed for the quantum force and local energy is easy to compute. The function $f(r_{ij})$ will depends on the system under study. In the equations below we will keep this general form.

Importance sampling, program elements

In the Metropolis/Hasting algorithm, the *acceptance ratio* determines the probability for a particle to be accepted at a new position. The ratio of the trial wave functions evaluated at the new and current positions is given by (OB for the onebody part)

$$R \equiv \frac{\Psi_T^{new}}{\Psi_T^{old}} = \frac{\Psi_{OB}^{new}}{\Psi_{OB}^{old}}\frac{\Psi_C^{new}}{\Psi_C^{old}}$$

Here Ψ_{OB} is our onebody part (Slater determinant or product of boson single-particle states) while Ψ_C is our correlation function, or Jastrow factor. We need to optimize the $\nabla\Psi_T/\Psi_T$ ratio and the second derivative as well, that is the $\nabla^2\Psi_T/\Psi_T$ ratio. The first is needed when we compute the so-called quantum force in importance sampling. The second is needed when we compute the kinetic energy term of the local energy.

$$\frac{\nabla\Psi}{\Psi} = \frac{\nabla(\Psi_{OB}\,\Psi_C)}{\Psi_{OB}\,\Psi_C} = \frac{\Psi_C\nabla\Psi_{OB} + \Psi_{OB}\nabla\Psi_C}{\Psi_{OB}\Psi_C} = \frac{\nabla\Psi_{OB}}{\Psi_{OB}} + \frac{\nabla\Psi_C}{\Psi_C}$$

Importance sampling

The expectation value of the kinetic energy expressed in atomic units for electron i is

$$\langle\hat{K}_i\rangle = -\frac{1}{2}\frac{\langle\Psi|\nabla_i^2|\Psi\rangle}{\langle\Psi|\Psi\rangle},$$

$$\hat{K}_i = -\frac{1}{2}\frac{\nabla_i^2\Psi}{\Psi}.$$

Importance sampling

The second derivative which enters the definition of the local energy is

$$\frac{\nabla^2\Psi}{\Psi} = \frac{\nabla^2\Psi_{OB}}{\Psi_{OB}} + \frac{\nabla^2\Psi_C}{\Psi_C} + 2\frac{\nabla\Psi_{OB}}{\Psi_{OB}}\cdot\frac{\nabla\Psi_C}{\Psi_C}$$

We discuss here how to calculate these quantities in an optimal way,

Importance sampling

We have defined the correlated function as

$$\Psi_C = \prod_{i<j} g(r_{ij}) = \prod_{i<j}^{N} g(r_{ij}) = \prod_{i=1}^{N}\prod_{j=i+1}^{N} g(r_{ij}),$$

with $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j| = \sqrt{(x_i-x_j)^2+(y_i-y_j)^2+(z_i-z_j)^2}$ in three dimensions or $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j| = \sqrt{(x_i-x_j)^2+(y_i-y_j)^2}$ if we work with two-dimensional systems.

In our particular case we have

$$\Psi_C = \prod_{i<j} g(r_{ij}) = \exp\left\{\sum_{i<j} f(r_{ij})\right\}.$$

Importance sampling

The total number of different relative distances r_{ij} is $N(N-1)/2$. In a matrix storage format, the relative distances form a strictly upper triangular matrix

$$\mathbf{r} \equiv \begin{pmatrix} 0 & r_{1,2} & r_{1,3} & \cdots & r_{1,N} \\ \vdots & 0 & r_{2,3} & \cdots & r_{2,N} \\ \vdots & \vdots & 0 & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & r_{N-1,N} \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix}.$$

This applies to $\mathbf{g} = \mathbf{g}(r_{ij})$ as well.

In our algorithm we will move one particle at the time, say the *kth*-particle. This sampling will be seen to be particularly efficient when we are going to compute a Slater determinant.

Importance sampling

We have that the ratio between Jastrow factors R_C is given by

$$R_C = \frac{\Psi_C^{\mathrm{new}}}{\Psi_C^{\mathrm{cur}}} = \prod_{i=1}^{k-1}\frac{g_{ik}^{\mathrm{new}}}{g_{ik}^{\mathrm{cur}}}\prod_{i=k+1}^{N}\frac{g_{ki}^{\mathrm{new}}}{g_{ki}^{\mathrm{cur}}}.$$

For the Pade-Jastrow form

$$R_C = \frac{\Psi_C^{\rm new}}{\Psi_C^{\rm cur}} = \frac{\exp U_{new}}{\exp U_{cur}} = \exp \Delta U,$$

where

$$\Delta U = \sum_{i=1}^{k-1} \left(f_{ik}^{\rm new} - f_{ik}^{\rm cur} \right) + \sum_{i=k+1}^{N} \left(f_{ki}^{\rm new} - f_{ki}^{\rm cur} \right)$$

Importance sampling

One needs to develop a special algorithm that runs only through the elements of the upper triangular matrix **g** and have k as an index.

The expression to be derived in the following is of interest when computing the quantum force and the kinetic energy. It has the form

$$\frac{\nabla_i \Psi_C}{\Psi_C} = \frac{1}{\Psi_C} \frac{\partial \Psi_C}{\partial x_i},$$

for all dimensions and with i running over all particles.

Importance sampling

For the first derivative only $N-1$ terms survive the ratio because the g-terms that are not differentiated cancel with their corresponding ones in the denominator. Then,

$$\frac{1}{\Psi_C} \frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1} \frac{1}{g_{ik}} \frac{\partial g_{ik}}{\partial x_k} + \sum_{i=k+1}^{N} \frac{1}{g_{ki}} \frac{\partial g_{ki}}{\partial x_k}.$$

An equivalent equation is obtained for the exponential form after replacing g_{ij} by $\exp(f_{ij})$, yielding:

$$\frac{1}{\Psi_C} \frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1} \frac{\partial g_{ik}}{\partial x_k} + \sum_{i=k+1}^{N} \frac{\partial g_{ki}}{\partial x_k},$$

with both expressions scaling as $\mathcal{O}(N)$.

Importance sampling

Using the identity

$$\frac{\partial}{\partial x_i} g_{ij} = -\frac{\partial}{\partial x_j} g_{ij},$$

we get expressions where all the derivatives acting on the particle are represented by the *second* index of g:

$$\frac{1}{\Psi_C} \frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1} \frac{1}{g_{ik}} \frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N} \frac{1}{g_{ki}} \frac{\partial g_{ki}}{\partial x_i},$$

and for the exponential case:

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_i}.$$

Importance sampling

For correlation forms depending only on the scalar distances r_{ij} we can use the chain rule. Noting that

$$\frac{\partial g_{ij}}{\partial x_j} = \frac{\partial g_{ij}}{\partial r_{ij}}\frac{\partial r_{ij}}{\partial x_j} = \frac{x_j - x_i}{r_{ij}}\frac{\partial g_{ij}}{\partial r_{ij}},$$

we arrive at

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{1}{g_{ik}}\frac{\mathbf{r_{ik}}}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}} - \sum_{i=k+1}^{N}\frac{1}{g_{ki}}\frac{\mathbf{r_{ki}}}{r_{ki}}\frac{\partial g_{ki}}{\partial r_{ki}}.$$

Importance sampling

Note that for the Pade-Jastrow form we can set $g_{ij} \equiv g(r_{ij}) = e^{f(r_{ij})} = e^{f_{ij}}$ and

$$\frac{\partial g_{ij}}{\partial r_{ij}} = g_{ij}\frac{\partial f_{ij}}{\partial r_{ij}}.$$

Therefore,

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{\mathbf{r_{ik}}}{r_{ik}}\frac{\partial f_{ik}}{\partial r_{ik}} - \sum_{i=k+1}^{N}\frac{\mathbf{r_{ki}}}{r_{ki}}\frac{\partial f_{ki}}{\partial r_{ki}},$$

where

$$\mathbf{r}_{ij} = |\mathbf{r}_j - \mathbf{r}_i| = (x_j - x_i)\mathbf{e}_1 + (y_j - y_i)\mathbf{e}_2 + (z_j - z_i)\mathbf{e}_3$$

is the relative distance.

Importance sampling

The second derivative of the Jastrow factor divided by the Jastrow factor (the way it enters the kinetic energy) is

$$\left[\frac{\nabla^2\Psi_C}{\Psi_C}\right]_x = 2\sum_{k=1}^{N}\sum_{i=1}^{k-1}\frac{\partial^2 g_{ik}}{\partial x_k^2} + \sum_{k=1}^{N}\left(\sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_i}\right)^2$$

Importance sampling

But we have a simple form for the function, namely

$$\Psi_C = \prod_{i<j}\exp f(r_{ij}),$$

and it is easy to see that for particle k we have

$$\frac{\nabla_k^2\Psi_C}{\Psi_C} = \sum_{ij\neq k}\frac{(\mathbf{r}_k-\mathbf{r}_i)(\mathbf{r}_k-\mathbf{r}_j)}{r_{ki}r_{kj}}f'(r_{ki})f'(r_{kj}) + \sum_{j\neq k}\left(f''(r_{kj})+\frac{2}{r_{kj}}f'(r_{kj})\right)$$

Why blocking?

Statistical analysis.

- Monte Carlo simulations can be treated as *computer experiments*
- The results can be analysed with the same statistical tools as we would use analysing experimental data.
- As in all experiments, we are looking for expectation values and an estimate of how accurate they are, i.e., possible sources for errors.

A very good article which explains blocking is H. Flyvbjerg and H. G. Petersen, *Error estimates on averages of correlated data*, Journal of Chemical Physics 91, 461-466 (1989).

Why blocking?

Statistical analysis.

- As in other experiments, Monte Carlo experiments have two classes of errors:
  - Statistical errors
  - Systematical errors
- Statistical errors can be estimated using standard tools from statistics
- Systematical errors are method specific and must be treated differently from case to case. (In VMC a common source is the step length or time step in importance sampling)

Statistics and blocking

The *probability distribution function (PDF)* is a function $p(x)$ on the domain which, in the discrete case, gives us the probability or relative frequency with which these values of X occur:

$$p(x) = \text{prob}(X = x)$$

In the continuous case, the PDF does not directly depict the actual probability. Instead we define the probability for the stochastic variable to assume any value on an infinitesimal interval around x to be $p(x)dx$. The continuous function $p(x)$

then gives us the *density* of the probability rather than the probability itself. The probability for a stochastic variable to assume any value on a non-infinitesimal interval $[a,\, b]$ is then just the integral:

$$\text{prob}(a \leq X \leq b) = \int_a^b p(x)dx$$

Qualitatively speaking, a stochastic variable represents the values of numbers chosen as if by chance from some specified PDF so that the selection of a large set of these numbers reproduces this PDF.

Statistics and blocking

Also of interest to us is the *cumulative probability distribution function (CDF)*, $P(x)$, which is just the probability for a stochastic variable X to assume any value less than x:

$$P(x) = \text{Prob}(X \leq x) = \int_{-\infty}^{x} p(x')dx'$$

The relation between a CDF and its corresponding PDF is then:

$$p(x) = \frac{d}{dx}P(x)$$

Statistics and blocking

A particularly useful class of special expectation values are the *moments*. The n-th moment of the PDF p is defined as follows:

$$\langle x^n \rangle \equiv \int x^n p(x)\, dx$$

The zero-th moment $\langle 1 \rangle$ is just the normalization condition of p. The first moment, $\langle x \rangle$, is called the *mean* of p and often denoted by the letter μ:

$$\langle x \rangle = \mu \equiv \int x p(x)\, dx$$

Statistics and blocking

A special version of the moments is the set of *central moments*, the n-th central moment defined as:

$$\langle (x - \langle x \rangle)^n \rangle \equiv \int (x - \langle x \rangle)^n p(x)\, dx$$

The zero-th and first central moments are both trivial, equal 1 and 0, respectively. But the second central moment, known as the *variance* of p, is of particular

interest. For the stochastic variable X, the variance is denoted as σ_X^2 or $\mathrm{var}(X)$:

$$\begin{aligned}\sigma_X^2 &= \mathrm{var}(X) = \langle (x - \langle x \rangle)^2 \rangle = \int (x - \langle x \rangle)^2 p(x)\, dx && (7)\\ &= \int \left(x^2 - 2x\langle x \rangle^2 + \langle x \rangle^2 \right) p(x)\, dx && (8)\\ &= \langle x^2 \rangle - 2\langle x \rangle \langle x \rangle + \langle x \rangle^2 && (9)\\ &= \langle x^2 \rangle - \langle x \rangle^2 && (10)\end{aligned}$$

The square root of the variance, $\sigma = \sqrt{\langle (x - \langle x \rangle)^2 \rangle}$ is called the *standard deviation* of p. It is clearly just the RMS (root-mean-square) value of the deviation of the PDF from its mean value, interpreted qualitatively as the *spread* of p around its mean.

Statistics and blocking

Another important quantity is the so called covariance, a variant of the above defined variance. Consider again the set $\{X_i\}$ of n stochastic variables (not necessarily uncorrelated) with the multivariate PDF $P(x_1, \ldots, x_n)$. The *covariance* of two of the stochastic variables, X_i and X_j, is defined as follows:

$$\begin{aligned}\mathrm{cov}(X_i, X_j) &\equiv \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \\ &= \int \cdots \int (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle)\, P(x_1, \ldots, x_n)\, dx_1 \ldots dx_n \qquad (11)\end{aligned}$$

with

$$\langle x_i \rangle = \int \cdots \int x_i\, P(x_1, \ldots, x_n)\, dx_1 \ldots dx_n$$

Statistics and blocking

If we consider the above covariance as a matrix $C_{ij} = \mathrm{cov}(X_i, X_j)$, then the diagonal elements are just the familiar variances, $C_{ii} = \mathrm{cov}(X_i, X_i) = \mathrm{var}(X_i)$. It turns out that all the off-diagonal elements are zero if the stochastic variables are uncorrelated. This is easy to show, keeping in mind the linearity of the expectation value. Consider the stochastic variables X_i and X_j, $(i \neq j)$:

$$\begin{aligned}\mathrm{cov}(X_i, X_j) &= \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle && (12)\\ &= \langle x_i x_j - x_i \langle x_j \rangle - \langle x_i \rangle x_j + \langle x_i \rangle \langle x_j \rangle \rangle && (13)\\ &= \langle x_i x_j \rangle - \langle x_i \langle x_j \rangle \rangle - \langle \langle x_i \rangle x_j \rangle + \langle \langle x_i \rangle \langle x_j \rangle \rangle && (14)\\ &= \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle - \langle x_i \rangle \langle x_j \rangle + \langle x_i \rangle \langle x_j \rangle && (15)\\ &= \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle && (16)\end{aligned}$$

Statistics and blocking

If X_i and X_j are independent, we get $\langle x_i x_j \rangle = \langle x_i \rangle \langle x_j \rangle$, resulting in $\mathrm{cov}(X_i, X_j) = 0 \quad (i \neq j)$.

Also useful for us is the covariance of linear combinations of stochastic variables. Let $\{X_i\}$ and $\{Y_i\}$ be two sets of stochastic variables. Let also $\{a_i\}$ and $\{b_i\}$ be two sets of scalars. Consider the linear combination:

$$U = \sum_i a_i X_i \qquad V = \sum_j b_j Y_j$$

By the linearity of the expectation value

$$\mathrm{cov}(U, V) = \sum_{i,j} a_i b_j \mathrm{cov}(X_i, Y_j)$$

Statistics and blocking

Now, since the variance is just $\mathrm{var}(X_i) = \mathrm{cov}(X_i, X_i)$, we get the variance of the linear combination $U = \sum_i a_i X_i$:

$$\mathrm{var}(U) = \sum_{i,j} a_i a_j \mathrm{cov}(X_i, X_j) \tag{17}$$

And in the special case when the stochastic variables are uncorrelated, the off-diagonal elements of the covariance are as we know zero, resulting in:

$$\mathrm{var}(U) = \sum_i a_i^2 \mathrm{cov}(X_i, X_i) = \sum_i a_i^2 \mathrm{var}(X_i)$$

$$\mathrm{var}(\sum_i a_i X_i) = \sum_i a_i^2 \mathrm{var}(X_i)$$

which will become very useful in our study of the error in the mean value of a set of measurements.

Statistics and blocking

A *stochastic process* is a process that produces sequentially a chain of values:

$$\{x_1, x_2, \dots x_k, \dots\}.$$

We will call these values our *measurements* and the entire set as our measured *sample*. The action of measuring all the elements of a sample we will call a stochastic *experiment* since, operationally, they are often associated with results of empirical observation of some physical or mathematical phenomena; precisely an experiment. We assume that these values are distributed according to some PDF $p_X(x)$, where X is just the formal symbol for the stochastic variable whose PDF is $p_X(x)$. Instead of trying to determine the full distribution p we are often only interested in finding the few lowest moments, like the mean μ_X and the variance σ_X.

Statistics and blocking

In practical situations a sample is always of finite size. Let that size be n. The expectation value of a sample, the *sample mean*, is then defined as follows:

$$\bar{x}_n \equiv \frac{1}{n}\sum_{k=1}^{n} x_k$$

The *sample variance* is:

$$\mathrm{var}(x) \equiv \frac{1}{n}\sum_{k=1}^{n}(x_k - \bar{x}_n)^2$$

its square root being the *standard deviation of the sample*. The *sample covariance* is:

$$\mathrm{cov}(x) \equiv \frac{1}{n}\sum_{kl}(x_k - \bar{x}_n)(x_l - \bar{x}_n)$$

Statistics and blocking

Note that the sample variance is the sample covariance without the cross terms. In a similar manner as the covariance in Eq. (11) is a measure of the correlation between two stochastic variables, the above defined sample covariance is a measure of the sequential correlation between succeeding measurements of a sample.

These quantities, being known experimental values, differ significantly from and must not be confused with the similarly named quantities for stochastic variables, mean μ_X, variance $\mathrm{var}(X)$ and covariance $\mathrm{cov}(X, Y)$.

Statistics and blocking

The law of large numbers states that as the size of our sample grows to infinity, the sample mean approaches the true mean μ_X of the chosen PDF:

$$\lim_{n\to\infty} \bar{x}_n = \mu_X$$

The sample mean $\bar{x}_n$ works therefore as an estimate of the true mean μ_X.

What we need to find out is how good an approximation $\bar{x}_n$ is to μ_X. In any stochastic measurement, an estimated mean is of no use to us without a measure of its error. A quantity that tells us how well we can reproduce it in another experiment. We are therefore interested in the PDF of the sample mean itself. Its standard deviation will be a measure of the spread of sample means, and we will simply call it the *error* of the sample mean, or just sample error, and denote it by err_X. In practice, we will only be able to produce an *estimate* of the sample error since the exact value would require the knowledge of the true PDFs behind, which we usually do not have.

Statistics and blocking

The straight forward brute force way of estimating the sample error is simply by producing a number of samples, and treating the mean of each as a measurement. The standard deviation of these means will then be an estimate of the original sample error. If we are unable to produce more than one sample, we can split it up sequentially into smaller ones, treating each in the same way as above. This procedure is known as *blocking* and will be given more attention shortly. At this point it is worth while exploring more indirect methods of estimation that will help us understand some important underlying principles of correlational effects.

Statistics and blocking

Let us first take a look at what happens to the sample error as the size of the sample grows. In a sample, each of the measurements x_i can be associated with its own stochastic variable X_i. The stochastic variable $\overline{X}_n$ for the sample mean $\bar{x}_n$ is then just a linear combination, already familiar to us:

$$\overline{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$$

All the coefficients are just equal $1/n$. The PDF of $\overline{X}_n$, denoted by $p_{\overline{X}_n}(x)$ is the desired PDF of the sample means.

Statistics and blocking

The probability density of obtaining a sample mean $\bar{x}_n$ is the product of probabilities of obtaining arbitrary values $x_1, x_2, \ldots, x_n$ with the constraint that the mean of the set $\{x_i\}$ is $\bar{x}_n$:

$$p_{\overline{X}_n}(x) = \int p_X(x_1)\cdots\int p_X(x_n)\ \delta\left(x - \frac{x_1 + x_2 + \cdots + x_n}{n}\right) dx_n \cdots dx_1$$

And in particular we are interested in its variance $\mathrm{var}(\overline{X}_n)$.

Statistics and blocking

It is generally not possible to express $p_{\overline{X}_n}(x)$ in a closed form given an arbitrary PDF p_X and a number n. But for the limit $n \to \infty$ it is possible to make an approximation. The very important result is called *the central limit theorem*. It tells us that as n goes to infinity, $p_{\overline{X}_n}(x)$ approaches a Gaussian distribution whose mean and variance equal the true mean and variance, μ_X and σ_X^2, respectively:

$$\lim_{n\to\infty} p_{\overline{X}_n}(x) = \left(\frac{n}{2\pi\mathrm{var}(X)}\right)^{1/2} e^{-\frac{n(x-\bar{x}_n)^2}{2\mathrm{var}(X)}} \tag{18}$$

Statistics and blocking

The desired variance $\mathrm{var}(\overline{X}_n)$, i.e. the sample error squared err_X^2, is given by:

$$\mathrm{err}_X^2 = \mathrm{var}(\overline{X}_n) = \frac{1}{n^2}\sum_{ij}\mathrm{cov}(X_i, X_j) \tag{19}$$

We see now that in order to calculate the exact error of the sample with the above expression, we would need the true means μ_{X_i} of the stochastic variables X_i. To calculate these requires that we know the true multivariate PDF of all the X_i. But this PDF is unknown to us, we have only got the measurements of one sample. The best we can do is to let the sample itself be an estimate of the PDF of each of the X_i, estimating all properties of X_i through the measurements of the sample.

Statistics and blocking

Our estimate of μ_{X_i} is then the sample mean $\bar{x}$ itself, in accordance with the the central limit theorem:

$$\mu_{X_i} = \langle x_i \rangle \approx \frac{1}{n}\sum_{k=1}^{n} x_k = \bar{x}$$

Using $\bar{x}$ in place of μ_{X_i} we can give an *estimate* of the covariance in Eq. (19)

$$\mathrm{cov}(X_i, X_j) = \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \approx \langle (x_i - \bar{x})(x_j - \bar{x}) \rangle,$$

resulting in

$$\frac{1}{n}\sum_{l}^{n}\left(\frac{1}{n}\sum_{k}^{n}(x_k - \bar{x}_n)(x_l - \bar{x}_n)\right) = \frac{1}{n}\frac{1}{n}\sum_{kl}(x_k - \bar{x}_n)(x_l - \bar{x}_n) = \frac{1}{n}\mathrm{cov}(x)$$

Statistics and blocking

By the same procedure we can use the sample variance as an estimate of the variance of any of the stochastic variables X_i

$$\mathrm{var}(X_i) = \langle x_i - \langle x_i \rangle \rangle \approx \langle x_i - \bar{x}_n \rangle,$$

which is approximated as

$$\mathrm{var}(X_i) \approx \frac{1}{n}\sum_{k=1}^{n}(x_k - \bar{x}_n) = \mathrm{var}(x) \tag{20}$$

Now we can calculate an estimate of the error err_X of the sample mean $\bar{x}_n$:

$$\begin{aligned}\mathrm{err}_X^2 &= \frac{1}{n^2}\sum_{ij}\mathrm{cov}(X_i, X_j) \\ &\approx \qquad\qquad \frac{1}{n^2}\sum_{ij}\frac{1}{n}\mathrm{cov}(x) = \frac{1}{n^2}n^2\frac{1}{n}\mathrm{cov}(x) \\ &= \frac{1}{n}\mathrm{cov}(x) \qquad (21)\end{aligned}$$

which is nothing but the sample covariance divided by the number of measurements in the sample.

Statistics and blocking

In the special case that the measurements of the sample are uncorrelated (equivalently the stochastic variables X_i are uncorrelated) we have that the off-diagonal elements of the covariance are zero. This gives the following estimate of the sample error:

$$\mathrm{err}_X^2 = \frac{1}{n^2}\sum_{ij}\mathrm{cov}(X_i, X_j) = \frac{1}{n^2}\sum_{i}\mathrm{var}(X_i),$$

resulting in

$$\mathrm{err}_X^2 \approx \frac{1}{n^2}\sum_{i}\mathrm{var}(x) = \frac{1}{n}\mathrm{var}(x) \qquad (22)$$

where in the second step we have used Eq. (20). The error of the sample is then just its standard deviation divided by the square root of the number of measurements the sample contains. This is a very useful formula which is easy to compute. It acts as a first approximation to the error, but in numerical experiments, we cannot overlook the always present correlations.

Statistics and blocking

For computational purposes one usually splits up the estimate of err_X^2, given by Eq. (21), into two parts

$$\mathrm{err}_X^2 = \frac{1}{n}\mathrm{var}(x) + \frac{1}{n}(\mathrm{cov}(x) - \mathrm{var}(x)),$$

which equals

$$\frac{1}{n^2}\sum_{k=1}^{n}(x_k - \bar{x}_n)^2 + \frac{2}{n^2}\sum_{k<l}(x_k - \bar{x}_n)(x_l - \bar{x}_n) \qquad (23)$$

The first term is the same as the error in the uncorrelated case, Eq. (22). This means that the second term accounts for the error correction due to correlation between the measurements. For uncorrelated measurements this second term is zero.

Statistics and blocking

Computationally the uncorrelated first term is much easier to treat efficiently than the second.

$$\mathrm{var}(x) = \frac{1}{n}\sum_{k=1}^{n}(x_k - \bar{x}_n)^2 = \left(\frac{1}{n}\sum_{k=1}^{n}x_k^2\right) - \bar{x}_n^2$$

We just accumulate separately the values x^2 and x for every measurement x we receive. The correlation term, though, has to be calculated at the end of the experiment since we need all the measurements to calculate the cross terms. Therefore, all measurements have to be stored throughout the experiment.

Statistics and blocking

Let us analyze the problem by splitting up the correlation term into partial sums of the form:

$$f_d = \frac{1}{n-d}\sum_{k=1}^{n-d}(x_k - \bar{x}_n)(x_{k+d} - \bar{x}_n)$$

The correlation term of the error can now be rewritten in terms of f_d

$$\frac{2}{n}\sum_{k<l}(x_k - \bar{x}_n)(x_l - \bar{x}_n) = 2\sum_{d=1}^{n-1} f_d$$

The value of f_d reflects the correlation between measurements separated by the distance d in the sample samples. Notice that for $d = 0$, f is just the sample variance, $\mathrm{var}(x)$. If we divide f_d by $\mathrm{var}(x)$, we arrive at the so called *autocorrelation function*

$$\kappa_d = \frac{f_d}{\mathrm{var}(x)}$$

which gives us a useful measure of the correlation pair correlation starting always at 1 for $d = 0$.

Statistics and blocking

The sample error (see eq. (23)) can now be written in terms of the autocorrelation function:

$$\begin{aligned}
\mathrm{err}_X^2 &= \frac{1}{n}\mathrm{var}(x) + \frac{2}{n}\cdot\mathrm{var}(x)\sum_{d=1}^{n-1}\frac{f_d}{\mathrm{var}(x)} \\
&= \left(1 + 2\sum_{d=1}^{n-1}\kappa_d\right)\frac{1}{n}\mathrm{var}(x) \\
&= \frac{\tau}{n}\cdot\mathrm{var}(x) \qquad (24)
\end{aligned}$$

and we see that err_X can be expressed in terms the uncorrelated sample variance times a correction factor τ which accounts for the correlation between measurements. We call this correction factor the *autocorrelation time*:

$$\tau = 1 + 2\sum_{d=1}^{n-1} \kappa_d \tag{25}$$

Statistics and blocking

For a correlation free experiment, τ equals 1. From the point of view of eq. (24) we can interpret a sequential correlation as an effective reduction of the number of measurements by a factor τ. The effective number of measurements becomes:

$$n_{\mathrm{eff}} = \frac{n}{\tau}$$

To neglect the autocorrelation time τ will always cause our simple uncorrelated estimate of $\mathrm{err}_X^2 \approx \mathrm{var}(x)/n$ to be less than the true sample error. The estimate of the error will be too *good*. On the other hand, the calculation of the full autocorrelation time poses an efficiency problem if the set of measurements is very large.

Can we understand this? Time Auto-correlation Function

The so-called time-displacement autocorrelation $\phi(t)$ for a quantity $\mathbf{M}$ is given by

$$\phi(t) = \int dt' \left[\mathbf{M}(t') - \langle \mathbf{M} \rangle\right] \left[\mathbf{M}(t'+t) - \langle \mathbf{M} \rangle\right],$$

which can be rewritten as

$$\phi(t) = \int dt' \left[\mathbf{M}(t')\mathbf{M}(t'+t) - \langle \mathbf{M} \rangle^2\right],$$

where $\langle \mathbf{M} \rangle$ is the average value and $\mathbf{M}(t)$ its instantaneous value. We can discretize this function as follows, where we used our set of computed values $\mathbf{M}(t)$ for a set of discretized times (our Monte Carlo cycles corresponding to moving all electrons?)

$$\phi(t) = \frac{1}{t_{\mathrm{max}} - t}\sum_{t'=0}^{t_{\mathrm{max}}-t} \mathbf{M}(t')\mathbf{M}(t'+t) - \frac{1}{t_{\mathrm{max}} - t}\sum_{t'=0}^{t_{\mathrm{max}}-t} \mathbf{M}(t') \times \frac{1}{t_{\mathrm{max}} - t}\sum_{t'=0}^{t_{\mathrm{max}}-t} \mathbf{M}(t'+t).$$

Time Auto-correlation Function

One should be careful with times close to t_{max}, the upper limit of the sums becomes small and we end up integrating over a rather small time interval. This means that the statistical error in $\phi(t)$ due to the random nature of the fluctuations in $\mathbf{M}(t)$ can become large.

One should therefore choose $t \ll t_{\max}$.

Note that the variable $\mathbf{M}$ can be any expectation values of interest.

The time-correlation function gives a measure of the correlation between the various values of the variable at a time t' and a time $t'+t$. If we multiply the values of $\mathbf{M}$ at these two different times, we will get a positive contribution if they are fluctuating in the same direction, or a negative value if they fluctuate in the opposite direction. If we then integrate over time, or use the discretized version of, the time correlation function $\phi(t)$ should take a non-zero value if the fluctuations are correlated, else it should gradually go to zero. For times a long way apart the different values of $\mathbf{M}$ are most likely uncorrelated and $\phi(t)$ should be zero.

Time Auto-correlation Function

We can derive the correlation time by observing that our Metropolis algorithm is based on a random walk in the space of all possible spin configurations. Our probability distribution function $\hat{\mathbf{w}}(t)$ after a given number of time steps t could be written as

$$\hat{\mathbf{w}}(t) = \hat{\mathbf{W}}^{\mathbf{t}}\hat{\mathbf{w}}(0),$$

with $\hat{\mathbf{w}}(0)$ the distribution at $t = 0$ and $\hat{\mathbf{W}}$ representing the transition probability matrix. We can always expand $\hat{\mathbf{w}}(0)$ in terms of the right eigenvectors of $\hat{\mathbf{v}}$ of $\hat{\mathbf{W}}$ as

$$\hat{\mathbf{w}}(0) = \sum_i \alpha_i \hat{\mathbf{v}}_i,$$

resulting in

$$\hat{\mathbf{w}}(t) = \hat{\mathbf{W}}^t\hat{\mathbf{w}}(0) = \hat{\mathbf{W}}^t \sum_i \alpha_i \hat{\mathbf{v}}_i = \sum_i \lambda_i^t \alpha_i \hat{\mathbf{v}}_i,$$

with λ_i the i^{th} eigenvalue corresponding to the eigenvector $\hat{\mathbf{v}}_i$.

Time Auto-correlation Function

If we assume that λ_0 is the largest eigenvector we see that in the limit $t \to \infty$, $\hat{\mathbf{w}}(t)$ becomes proportional to the corresponding eigenvector $\hat{\mathbf{v}}_0$. This is our steady state or final distribution.

We can relate this property to an observable like the mean energy. With the probabilty $\hat{\mathbf{w}}(t)$ (which in our case is the squared trial wave function) we can write the expectation values as

$$\langle \mathbf{M}(t) \rangle = \sum_\mu \hat{\mathbf{w}}(t)_\mu \mathbf{M}_\mu,$$

or as the scalar of a vector product

$$\langle \mathbf{M}(t) \rangle = \hat{\mathbf{w}}(t)\mathbf{m},$$

with $\mathbf{m}$ being the vector whose elements are the values of $\mathbf{M}_\mu$ in its various microstates μ.

Time Auto-correlation Function

We rewrite this relation as

$$\langle \mathbf{M}(t) \rangle = \hat{\mathbf{w}}(t)\mathbf{m} = \sum_i \lambda_i^t \alpha_i \hat{\mathbf{v}}_i \mathbf{m}_i.$$

If we define $m_i = \hat{\mathbf{v}}_i \mathbf{m}_i$ as the expectation value of $\mathbf{M}$ in the i^{th} eigenstate we can rewrite the last equation as

$$\langle \mathbf{M}(t) \rangle = \sum_i \lambda_i^t \alpha_i m_i.$$

Since we have that in the limit $t \to \infty$ the mean value is dominated by the the largest eigenvalue λ_0, we can rewrite the last equation as

$$\langle \mathbf{M}(t) \rangle = \langle \mathbf{M}(\infty) \rangle + \sum_{i \neq 0} \lambda_i^t \alpha_i m_i.$$

We define the quantity

$$\tau_i = -\frac{1}{log \lambda_i},$$

and rewrite the last expectation value as

$$\langle \mathbf{M}(t) \rangle = \langle \mathbf{M}(\infty) \rangle + \sum_{i \neq 0} \alpha_i m_i e^{-t/\tau_i}.$$

Time Auto-correlation Function

The quantities τ_i are the correlation times for the system. They control also the auto-correlation function discussed above. The longest correlation time is obviously given by the second largest eigenvalue τ_1, which normally defines the correlation time discussed above. For large times, this is the only correlation time that survives. If higher eigenvalues of the transition matrix are well separated from λ_1 and we simulate long enough, τ_1 may well define the correlation time. In other cases we may not be able to extract a reliable result for τ_1. Coming back to the time correlation function $\phi(t)$ we can present a more general definition in terms of the mean magnetizations $\langle \mathbf{M}(t) \rangle$. Recalling that the mean value is equal to $\langle \mathbf{M}(\infty) \rangle$ we arrive at the expectation values

$$\phi(t) = \langle \mathbf{M}(0) - \mathbf{M}(\infty) \rangle \langle \mathbf{M}(t) - \mathbf{M}(\infty) \rangle,$$

resulting in

$$\phi(t) = \sum_{i,j \neq 0} m_i \alpha_i m_j \alpha_j e^{-t/\tau_i},$$

which is appropriate for all times.

Correlation Time

If the correlation function decays exponentially

$$\phi(t) \sim \exp(-t/\tau)$$

then the exponential correlation time can be computed as the average

$$\tau_{\exp} = -\langle \frac{t}{log|\frac{\phi(t)}{\phi(0)}|} \rangle.$$

If the decay is exponential, then

$$\int_0^\infty dt\phi(t) = \int_0^\infty dt\phi(0)\exp(-t/\tau) = \tau\phi(0),$$

which suggests another measure of correlation

$$\tau_{\text{int}} = \sum_k \frac{\phi(k)}{\phi(0)},$$

called the integrated correlation time.

What is blocking?

Blocking.

- Say that we have a set of samples from a Monte Carlo experiment
- Assuming (wrongly) that our samples are uncorrelated our best estimate of the standard deviation of the mean $\langle \mathbf{M} \rangle$ is given by

$$\sigma = \sqrt{\frac{1}{n}\left(\langle \mathbf{M}^2 \rangle - \langle \mathbf{M} \rangle^2\right)}$$

- If the samples are correlated we can rewrite our results to show that

$$\sigma = \sqrt{\frac{1 + 2\tau/\Delta t}{n}\left(\langle \mathbf{M}^2 \rangle - \langle \mathbf{M} \rangle^2\right)}$$

where τ is the correlation time (the time between a sample and the next uncorrelated sample) and Δt is time between each sample

What is blocking?

Blocking.

- If $\Delta t \gg \tau$ our first estimate of σ still holds
- Much more common that $\Delta t < \tau$

- In the method of data blocking we divide the sequence of samples into blocks
- We then take the mean $\langle \mathbf{M}_i \rangle$ of block $i = 1 \ldots n_{blocks}$ to calculate the total mean and variance
- The size of each block must be so large that sample j of block i is not correlated with sample j of block $i + 1$
- The correlation time τ would be a good choice

What is blocking?

Blocking.

- Problem: We don't know τ or it is too expensive to compute
- Solution: Make a plot of std. dev. as a function of blocksize
- The estimate of std. dev. of correlated data is too low $\rightarrow$ the error will increase with increasing block size until the blocks are uncorrelated, where we reach a plateau
- When the std. dev. stops increasing the blocks are uncorrelated

Implementation

- Do a Monte Carlo simulation, storing all samples to file
- Do the statistical analysis on this file, independently of your Monte Carlo program
- Read the file into an array
- Loop over various block sizes
- For each block size n_b, loop over the array in steps of n_b taking the mean of elements $in_b, \ldots, (i + 1)n_b$
- Take the mean and variance of the resulting array
- Write the results for each block size to file for later analysis

Actual implementation with code, main function

When the file gets large, it can be useful to write your data in binary mode instead of ascii characters. The following file reads data from a binary file with the output from every Monte Carlo cycle.

```
int main (int nargs, char* args[])
{

  int min_block_size, max_block_size, n_block_samples;
  //   Read from screen a possible new vaue of n
  if (nargs > 3) {
    min_block_size  = atoi(args[1]);
    max_block_size  = atoi(args[2]);
    n_block_samples = atoi(args[3]);
  }
  else{
    cerr << "usage: ./mcint_blocking.x <min_bloc_size> "
         << "<max_block_size> <n_block_samples>" << endl;
    exit(1);
  }

  // get file size using stat
  struct stat result;
  int n;
  // find number of data points
  if(stat("result.dat", &result) == 0){
    n = result.st_size/sizeof(double);
  }
  else{
    cerr << "error in getting file size" << endl;
    exit(1);
  }

  // get all mc results from file
  double* mc_results = new double[n];

  ifstream infile;
  infile.open("result.dat", ios::in | ios::binary);
//   additional lines omitted
  infile.close();
  }
  double mean, sigma;
  double res[2];
  meanvar(mc_results, n, res);
  mean = res[0]; sigma= res[1];
  cout << "Value of integral = " <<  mean  <<  endl;
  cout << "Value of variance = " << sigma << endl;
  cout << "Standard deviation = " <<  sqrt(sigma/(n-1.0)) << endl;
  // Open file for writing, writing results in formated output for plotting:
  ofstream outfile;
  outfile.open("blockres.dat", ios::out);
  outfile << setprecision(10);
  double* block_results = new double[n_block_samples];
  int block_size, block_step_length;
  block_step_length = (max_block_size-min_block_size)/n_block_samples;

  // loop over block sizes
  for(int i=0; i<n_block_samples; i++){
    block_size = min_block_size+i*block_step_length;
```

```
    blocking(mc_results, n, block_size, res);
    mean  = res[0];
    sigma = res[1];
    // formated output
    outfile << block_size << "\t" << sqrt(sigma/((n/block_size)-1.0)) << endl;
  }
  outfile.close();
  return 0;
}
```

Actual implementation with code, mean value and variance

```
// find mean of values in vals
double mean(double *vals, int n_vals){

  double m=0;
  for(int i=0; i<n_vals; i++){
    m+=vals[i];
  }

  return m/double(n_vals);
}

// calculate mean and variance of vals, results stored in res
void meanvar(double *vals, int n_vals, double *res){
  double m2=0, m=0, val;
  for(int i=0; i<n_vals; i++){
    val=vals[i];
    m+=val;
    m2+=val*val;
  }
  m  /= double(n_vals);
  m2 /= double(n_vals);
  res[0] = m;
  res[1] = m2-(m*m);

}
```

Actual implementation with code, blocking part

```
// find mean and variance of blocks of size block_size.
// mean and variance are stored in res
void blocking(double *vals, int n_vals, int block_size, double *res){

  // note: integer division will waste some values
  int n_blocks = n_vals/block_size;

  /*
  cerr << "n_vals=" << n_vals << ", block_size=" << block_size << endl;
  if(n_vals%block_size > 0)
    cerr << "lost " << n_vals%block_size << " values due to integer division"
         << endl;
  */

  double* block_vals = new double[n_blocks];

  for(int i=0; i<n_blocks; i++){
    block_vals[i] = mean(vals+i*block_size, block_size);
  }
```

```
  meanvar(block_vals, n_blocks, res);

  delete block_vals;
}
```

Efficient calculation of Slater determinants

The potentially most time-consuming part is the evaluation of the gradient and the Laplacian of an N-particle Slater determinant.

We have to differentiate the determinant with respect to all spatial coordinates of all particles. A brute force differentiation would involve $N \cdot d$ evaluations of the entire determinant which would even worsen the already undesirable time scaling, making it $Nd \cdot O(N^3) \sim O(d \cdot N^4)$.

This poses serious hindrances to the overall efficiency of our code.

Matrix elements of Slater determinants

The efficiency can be improved however if we move only one electron at the time. The Slater determinant matrix $\hat{D}$ is defined by the matrix elements

$$d_{ij} = \phi_j(x_i)$$

where $\phi_j(\mathbf{r}_i)$ is a single particle wave function. The columns correspond to the position of a given particle while the rows stand for the various quantum numbers.

Efficient calculation of Slater determinants

What we need to realize is that when differentiating a Slater determinant with respect to some given coordinate, only one row of the corresponding Slater matrix is changed.

Therefore, by recalculating the whole determinant we risk producing redundant information. The solution turns out to be an algorithm that requires to keep track of the *inverse* of the Slater matrix.

Efficient calculation of Slater determinants

Let the current position in phase space be represented by the $(N \cdot d)$-element vector $\mathbf{r}^{\text{old}}$ and the new suggested position by the vector $\mathbf{r}^{\text{new}}$.

The inverse of $\hat{D}$ can be expressed in terms of its cofactors C_{ij} and its determinant (this our notation for a determinant) $|\hat{D}|$:

$$d_{ij}^{-1} = \frac{C_{ji}}{|\hat{D}|} \tag{26}$$

Notice that the interchanged indices indicate that the matrix of cofactors is to be transposed.

Efficient calculation of Slater determinants

If $\hat{D}$ is invertible, then we must obviously have $\hat{D}^{-1}\hat{D} = \mathbf{1}$, or explicitly in terms of the individual elements of $\hat{D}$ and $\hat{D}^{-1}$:

$$\sum_{k=1}^{N} d_{ik} d_{kj}^{-1} = \delta_{ij} \tag{27}$$

Efficient calculation of Slater determinants

Consider the ratio, which we shall call R, between $|\hat{D}(\mathbf{r}^{\text{new}})|$ and $|\hat{D}(\mathbf{r}^{\text{old}})|$. By definition, each of these determinants can individually be expressed in terms of the i-th row of its cofactor matrix

$$R \equiv \frac{|\hat{D}(\mathbf{r}^{\text{new}})|}{|\hat{D}(\mathbf{r}^{\text{old}})|} = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})\, C_{ij}(\mathbf{r}^{\text{new}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}})\, C_{ij}(\mathbf{r}^{\text{old}})} \tag{28}$$

Efficient calculation of Slater determinants

Suppose now that we move only one particle at a time, meaning that $\mathbf{r}^{\text{new}}$ differs from $\mathbf{r}^{\text{old}}$ by the position of only one, say the i-th, particle . This means that $\hat{D}(\mathbf{r}^{\text{new}})$ and $\hat{D}(\mathbf{r}^{\text{old}})$ differ only by the entries of the i-th row. Recall also that the i-th row of a cofactor matrix $\hat{C}$ is independent of the entries of the i-th row of its corresponding matrix $\hat{D}$. In this particular case we therefore get that the i-th row of $\hat{C}(\mathbf{r}^{\text{new}})$ and $\hat{C}(\mathbf{r}^{\text{old}})$ must be equal. Explicitly, we have:

$$C_{ij}(\mathbf{r}^{\text{new}}) = C_{ij}(\mathbf{r}^{\text{old}}) \quad \forall\ j \in \{1, \ldots, N\} \tag{29}$$

Efficient calculation of Slater determinants

Inserting this into the numerator of eq. (28) and using eq. (26) to substitute the cofactors with the elements of the inverse matrix, we get:

$$R = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})\, C_{ij}(\mathbf{r}^{\text{old}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}})\, C_{ij}(\mathbf{r}^{\text{old}})} = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}})} \tag{30}$$

Efficient calculation of Slater determinants

Now by eq. (27) the denominator of the rightmost expression must be unity, so that we finally arrive at:

$$R = \sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}}) = \sum_{j=1}^{N} \phi_j(\mathbf{r}_i^{\text{new}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}}) \tag{31}$$

What this means is that in order to get the ratio when only the i-th particle has been moved, we only need to calculate the dot product of the vector

$(\phi_1(\mathbf{r}_i^{\rm new}), \ldots, \phi_N(\mathbf{r}_i^{\rm new}))$ of single particle wave functions evaluated at this new position with the i-th column of the inverse matrix $\hat{D}^{-1}$ evaluated at the original position. Such an operation has a time scaling of $O(N)$. The only extra thing we need to do is to maintain the inverse matrix $\hat{D}^{-1}(\mathbf{x}^{\rm old})$.

Efficient calculation of Slater determinants

If the new position $\mathbf{r}^{\rm new}$ is accepted, then the inverse matrix can by suitably updated by an algorithm having a time scaling of $O(N^2)$. This algorithm goes as follows. First we update all but the i-th column of $\hat{D}^{-1}$. For each column $j \neq i$, we first calculate the quantity:

$$S_j = (\hat{D}(\mathbf{r}^{\rm new}) \times \hat{D}^{-1}(\mathbf{r}^{\rm old}))_{ij} = \sum_{l=1}^{N} d_{il}(\mathbf{r}^{\rm new})\, d_{lj}^{-1}(\mathbf{r}^{\rm old}) \tag{32}$$

Efficient calculation of Slater determinants

The new elements of the j-th column of $\hat{D}^{-1}$ are then given by:

$$d_{kj}^{-1}(\mathbf{r}^{\rm new}) = d_{kj}^{-1}(\mathbf{r}^{\rm old}) - \frac{S_j}{R}\, d_{ki}^{-1}(\mathbf{r}^{\rm old}) \qquad \begin{array}{l} \forall \;\; k \in \{1, \ldots, N\} \\ j \neq i \end{array} \tag{33}$$

Efficient calculation of Slater determinants

Finally the elements of the i-th column of $\hat{D}^{-1}$ are updated simply as follows:

$$d_{ki}^{-1}(\mathbf{r}^{\rm new}) = \frac{1}{R}\, d_{ki}^{-1}(\mathbf{r}^{\rm old}) \quad \forall \;\; k \in \{1, \ldots, N\} \tag{34}$$

We see from these formulas that the time scaling of an update of $\hat{D}^{-1}$ after changing one row of $\hat{D}$ is $O(N^2)$.

The scheme is also applicable for the calculation of the ratios involving derivatives. It turns out that differentiating the Slater determinant with respect to the coordinates of a single particle $\mathbf{r}_i$ changes only the i-th row of the corresponding Slater matrix.

The gradient and the Laplacian

The gradient and the Laplacian can therefore be calculated as follows:

$$\frac{\vec{\nabla}_i |\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N} \vec{\nabla}_i d_{ij}(\mathbf{r}) d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \vec{\nabla}_i \phi_j(\mathbf{r}_i) d_{ji}^{-1}(\mathbf{r})$$

and

$$\frac{\nabla_i^2 |\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N} \nabla_i^2 d_{ij}(\mathbf{r}) d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \nabla_i^2 \phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r})$$

How to compute the derivates of the Slater determinant

Thus, to calculate all the derivatives of the Slater determinant, we only need the derivatives of the single particle wave functions ($\vec{\nabla}_i\phi_j(\mathbf{r}_i)$ and $\nabla_i^2\phi_j(\mathbf{r}_i)$) and the elements of the corresponding inverse Slater matrix ($\hat{D}^{-1}(\mathbf{r}_i)$). A calculation of a single derivative is by the above result an $O(N)$ operation. Since there are $d \cdot N$ derivatives, the time scaling of the total evaluation becomes $O(d \cdot N^2)$. With an $O(N^2)$ updating algorithm for the inverse matrix, the total scaling is no worse, which is far better than the brute force approach yielding $O(d \cdot N^4)$.

Important note: In most cases you end with closed form expressions for the single-particle wave functions. It is then useful to calculate the various derivatives and make separate functions for them.

The Slater determinant

The Slater determinant takes the form

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,,\mathbf{r}_3,\mathbf{r}_4,\alpha,\beta,\gamma,\delta) = \frac{1}{\sqrt{4!}}\begin{vmatrix} \psi_{100\uparrow}(\mathbf{r}_1) & \psi_{100\uparrow}(\mathbf{r}_2) & \psi_{100\uparrow}(\mathbf{r}_3) & \psi_{100\uparrow}(\mathbf{r}_4) \\ \psi_{100\downarrow}(\mathbf{r}_1) & \psi_{100\downarrow}(\mathbf{r}_2) & \psi_{100\downarrow}(\mathbf{r}_3) & \psi_{100\downarrow}(\mathbf{r}_4) \\ \psi_{200\uparrow}(\mathbf{r}_1) & \psi_{200\uparrow}(\mathbf{r}_2) & \psi_{200\uparrow}(\mathbf{r}_3) & \psi_{200\uparrow}(\mathbf{r}_4) \\ \psi_{200\downarrow}(\mathbf{r}_1) & \psi_{200\downarrow}(\mathbf{r}_2) & \psi_{200\downarrow}(\mathbf{r}_3) & \psi_{200\downarrow}(\mathbf{r}_4) \end{vmatrix}.$$

The Slater determinant as written is zero since the spatial wave functions for the spin up and spin down states are equal. But we can rewrite it as the product of two Slater determinants, one for spin up and one for spin down.

Rewriting the Slater determinant

We can rewrite it as

$$\begin{aligned}\Phi(\mathbf{r}_1,\mathbf{r}_2,,\mathbf{r}_3,\mathbf{r}_4,\alpha,\beta,\gamma,\delta) &= \det\uparrow(1,2)\det\downarrow(3,4) - \det\uparrow(1,3)\det\downarrow(2,4) \\ &- \det\uparrow(1,4)\det\downarrow(3,2) + \det\uparrow(2,3)\det\downarrow(1,4) - \det\uparrow(2,4)\det\downarrow(1,3) \\ &+ \det\uparrow(3,4)\det\downarrow(1,2),\end{aligned}$$

where we have defined

$$\det\uparrow(1,2) = \frac{1}{\sqrt{2}}\begin{vmatrix} \psi_{100\uparrow}(\mathbf{r}_1) & \psi_{100\uparrow}(\mathbf{r}_2) \\ \psi_{200\uparrow}(\mathbf{r}_1) & \psi_{200\uparrow}(\mathbf{r}_2) \end{vmatrix},$$

and

$$\det\downarrow(3,4) = \frac{1}{\sqrt{2}}\begin{vmatrix} \psi_{100\downarrow}(\mathbf{r}_3) & \psi_{100\downarrow}(\mathbf{r}_4) \\ \psi_{200\downarrow}(\mathbf{r}_3) & \psi_{200\downarrow}(\mathbf{r}_4) \end{vmatrix}.$$

The total determinant is still zero!

Splitting the Slater determinant

We want to avoid to sum over spin variables, in particular when the interaction does not depend on spin.

It can be shown, see for example Moskowitz and Kalos, Int. J. Quantum Chem. **20** 1107 (1981), that for the variational energy we can approximate the Slater determinant as

$$\Phi(\mathbf{r}_1, \mathbf{r}_2, , \mathbf{r}_3, \mathbf{r}_4, \alpha, \beta, \gamma, \delta) \propto \det\uparrow(1,2)\det\downarrow(3,4),$$

or more generally as

$$\Phi(\mathbf{r}_1, \mathbf{r}_2, \ldots \mathbf{r}_N) \propto \det\uparrow \det\downarrow,$$

where we have the Slater determinant as the product of a spin up part involving the number of electrons with spin up only (2 for beryllium and 5 for neon) and a spin down part involving the electrons with spin down.

This ansatz is not antisymmetric under the exchange of electrons with opposite spins but it can be shown (show this) that it gives the same expectation value for the energy as the full Slater determinant.

As long as the Hamiltonian is spin independent, the above is correct. It is rather straightforward to see this if you go back to the equations for the energy discussed earlier this semester.

Spin up and spin down parts

We will thus factorize the full determinant $|\hat{D}|$ into two smaller ones, where each can be identified with $\uparrow$ and $\downarrow$ respectively:

$$|\hat{D}| = |\hat{D}|_\uparrow \cdot |\hat{D}|_\downarrow$$

Factorization

The combined dimensionality of the two smaller determinants equals the dimensionality of the full determinant. Such a factorization is advantageous in that it makes it possible to perform the calculation of the ratio R and the updating of the inverse matrix separately for $|\hat{D}|_\uparrow$ and $|\hat{D}|_\downarrow$:

$$\frac{|\hat{D}|^{\text{new}}}{|\hat{D}|^{\text{old}}} = \frac{|\hat{D}|^{\text{new}}_\uparrow}{|\hat{D}|^{\text{old}}_\uparrow} \cdot \frac{|\hat{D}|^{\text{new}}_\downarrow}{|\hat{D}|^{\text{old}}_\downarrow}$$

This reduces the calculation time by a constant factor. The maximal time reduction happens in a system of equal numbers of $\uparrow$ and $\downarrow$ particles, so that the two factorized determinants are half the size of the original one.

Number of operations

Consider the case of moving only one particle at a time which originally had the following time scaling for one transition:

$$O_R(N) + O_{\rm inverse}(N^2)$$

For the factorized determinants one of the two determinants is obviously unaffected by the change so that it cancels from the ratio R.

Counting the number of FLOPS

Therefore, only one determinant of size $N/2$ is involved in each calculation of R and update of the inverse matrix. The scaling of each transition then becomes:

$$O_R(N/2) + O_{\rm inverse}(N^2/4)$$

and the time scaling when the transitions for all N particles are put together:

$$O_R(N^2/2) + O_{\rm inverse}(N^3/4)$$

which gives the same reduction as in the case of moving all particles at once.

Computation of ratios

Computing the ratios discussed above requires that we maintain the inverse of the Slater matrix evaluated at the current position. Each time a trial position is accepted, the row number i of the Slater matrix changes and updating its inverse has to be carried out. Getting the inverse of an $N \times N$ matrix by Gaussian elimination has a complexity of order of $\mathcal{O}(N^3)$ operations, a luxury that we cannot afford for each time a particle move is accepted. We will use the expression

$$d_{kj}^{-1}(\mathbf{x^{new}}) = \begin{cases} d_{kj}^{-1}(\mathbf{x^{old}}) - \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R}\sum_{l=1}^{N} d_{il}(\mathbf{x^{new}}) d_{lj}^{-1}(\mathbf{x^{old}}) & \text{if } j \neq i \\ \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R}\sum_{l=1}^{N} d_{il}(\mathbf{x^{old}}) d_{lj}^{-1}(\mathbf{x^{old}}) & \text{if } j = i \end{cases}$$

Scaling properties

This equation scales as $O(N^2)$. The evaluation of the determinant of an $N \times N$ matrix by standard Gaussian elimination requires $\mathbf{O}(N^3)$ calculations. As there are Nd independent coordinates we need to evaluate Nd Slater determinants for the gradient (quantum force) and Nd for the Laplacian (kinetic energy). With the updating algorithm we need only to invert the Slater determinant matrix once. This can be done by standard LU decomposition methods.

How to get the determinant

Determining a determinant of an $N \times N$ matrix by standard Gaussian elimination is of the order of $\mathbf{O}(N^3)$ calculations. As there are $N \cdot d$ independent coordinates we need to evaluate Nd Slater determinants for the gradient (quantum force) and $N \cdot d$ for the Laplacian (kinetic energy)

With the updating algorithm we need only to invert the Slater determinant matrix once. This is done by calling standard LU decomposition methods.

If you choose to implement the above recipe for the computation of the Slater determinant, you need to LU decompose the Slater matrix. This is described in chapter 6 of the lecture notes from FYS3150.

You need to call the function ludcmp in lib.cpp. You need to transfer the Slater matrix and its dimension. You get back an LU decomposed matrix.

LU decomposition and determinant

The LU decomposition method means that we can rewrite this matrix as the product of two matrices $\hat{B}$ and $\hat{C}$ where

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ b_{21} & 1 & 0 & 0 \\ b_{31} & b_{32} & 1 & 0 \\ b_{41} & b_{42} & b_{43} & 1 \end{pmatrix} \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ 0 & c_{22} & c_{23} & c_{24} \\ 0 & 0 & c_{33} & c_{34} \\ 0 & 0 & 0 & c_{44} \end{pmatrix}.$$

Determinant of a matrix

The matrix $\hat{A} \in \mathbb{R}^{n \times n}$ has an LU factorization if the determinant is different from zero. If the LU factorization exists and $\hat{A}$ is non-singular, then the LU factorization is unique and the determinant is given by

$$|\hat{A}| = c_{11}c_{22} \dots c_{nn}.$$

Expectation value of the kinetic energy

The expectation value of the kinetic energy expressed in atomic units for electron i is

$$\langle \hat{K}_i \rangle = -\frac{1}{2} \frac{\langle \Psi | \nabla_i^2 | \Psi \rangle}{\langle \Psi | \Psi \rangle},$$

$$K_i = -\frac{1}{2} \frac{\nabla_i^2 \Psi}{\Psi}. \tag{35}$$

$$\begin{aligned} \frac{\nabla^2 \Psi}{\Psi} &= \frac{\nabla^2 (\Psi_D \, \Psi_C)}{\Psi_D \, \Psi_C} = \frac{\nabla \cdot [\nabla (\Psi_D \, \Psi_C)]}{\Psi_D \, \Psi_C} = \frac{\nabla \cdot [\Psi_C \nabla \Psi_D + \Psi_D \nabla \Psi_C]}{\Psi_D \, \Psi_C} \\ &= \frac{\nabla \Psi_C \cdot \nabla \Psi_D + \Psi_C \nabla^2 \Psi_D + \nabla \Psi_D \cdot \nabla \Psi_C + \Psi_D \nabla^2 \Psi_C}{\Psi_D \, \Psi_C} \end{aligned} \tag{36}$$

$$\frac{\nabla^2\Psi}{\Psi} = \frac{\nabla^2\Psi_D}{\Psi_D} + \frac{\nabla^2\Psi_C}{\Psi_C} + 2\frac{\nabla\Psi_D}{\Psi_D}\cdot\frac{\nabla\Psi_C}{\Psi_C} \tag{37}$$

Second derivative of the Jastrow factor

The second derivative of the Jastrow factor divided by the Jastrow factor (the way it enters the kinetic energy) is

$$\left[\frac{\nabla^2\Psi_C}{\Psi_C}\right]_x = 2\sum_{k=1}^{N}\sum_{i=1}^{k-1}\frac{\partial^2 g_{ik}}{\partial x_k^2} + \sum_{k=1}^{N}\left(\sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_i}\right)^2$$

Functional form

But we have a simple form for the function, namely

$$\Psi_C = \prod_{i<j}\exp f(r_{ij}) = \exp\left\{\sum_{i<j}\frac{ar_{ij}}{1+\beta r_{ij}}\right\},$$

and it is easy to see that for particle k we have

$$\frac{\nabla_k^2\Psi_C}{\Psi_C} = \sum_{ij\neq k}\frac{(\mathbf{r}_k-\mathbf{r}_i)(\mathbf{r}_k-\mathbf{r}_j)}{r_{ki}r_{kj}}f'(r_{ki})f'(r_{kj}) + \sum_{j\neq k}\left(f''(r_{kj}) + \frac{2}{r_{kj}}f'(r_{kj})\right)$$

Second derivative of the Jastrow factor

Using

$$f(r_{ij}) = \frac{ar_{ij}}{1+\beta r_{ij}},$$

and $g'(r_{kj}) = dg(r_{kj})/dr_{kj}$ and $g''(r_{kj}) = d^2g(r_{kj})/dr_{kj}^2$ we find that for particle k we have

$$\frac{\nabla_k^2\Psi_C}{\Psi_C} = \sum_{ij\neq k}\frac{(\mathbf{r}_k-\mathbf{r}_i)(\mathbf{r}_k-\mathbf{r}_j)}{r_{ki}r_{kj}}\frac{a}{(1+\beta r_{ki})^2}\frac{a}{(1+\beta r_{kj})^2} + \sum_{j\neq k}\left(\frac{2a}{r_{kj}(1+\beta r_{kj})^2} - \frac{2a\beta}{(1+\beta r_{kj})^3}\right)$$

Gradient and Laplacian

The gradient and Laplacian can be calculated as follows:

$$\frac{\nabla_i|\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N}\vec{\nabla}_i d_{ij}(\mathbf{r})\, d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N}\vec{\nabla}_i\phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r})$$

and

$$\frac{\nabla_i^2|\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N}\nabla_i^2 d_{ij}(\mathbf{r})\, d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N}\nabla_i^2\phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r})$$

The gradient for the determinant

The gradient for the determinant is

$$\frac{\nabla_i|\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N} \nabla_i d_{ij}(\mathbf{r})\, d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \nabla_i \phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r}).$$

Jastrow gradient in quantum force

We have

$$\Psi_C = \prod_{i<j} g(r_{ij}) = \exp\left\{\sum_{i<j} \frac{a r_{ij}}{1+\beta r_{ij}}\right\},$$

the gradient needed for the quantum force and local energy is easy to compute. We get for particle k

$$\frac{\nabla_k \Psi_C}{\Psi_C} = \sum_{j\neq k} \frac{\mathbf{r}_{kj}}{r_{kj}} \frac{a}{(1+\beta r_{kj})^2},$$

which is rather easy to code. Remember to sum over all particles when you compute the local energy.

Metropolis Hastings part

We need to compute the ratio between wave functions, in particular for the Slater determinants.

$$R = \sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\mathrm{new}})\, d_{ji}^{-1}(\mathbf{r}^{\mathrm{old}}) = \sum_{j=1}^{N} \phi_j(\mathbf{r}_i^{\mathrm{new}})\, d_{ji}^{-1}(\mathbf{r}^{\mathrm{old}})$$

What this means is that in order to get the ratio when only the i-th particle has been moved, we only need to calculate the dot product of the vector $(\phi_1(\mathbf{r}_i^{\mathrm{new}}), \ldots, \phi_N(\mathbf{r}_i^{\mathrm{new}}))$ of single particle wave functions evaluated at this new position with the i-th column of the inverse matrix $\hat{D}^{-1}$ evaluated at the original position. Such an operation has a time scaling of $O(N)$. The only extra thing we need to do is to maintain the inverse matrix $\hat{D}^{-1}(\mathbf{x}^{\mathrm{old}})$.