Computational Physics 2: Variational Monte Carlo methods

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

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

Feb 7, 2019

Quantum Monte Carlo Motivation

We start with the variational principle. 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 bosons 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

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

A simple Python code that solves the two-boson or two-fermion case in two-dimensions

```
# Importing various packages
from math import exp, sqrt
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import sys

#Trial wave function for quantum dots in two dims
def WaveFunction(r,alpha,beta):
    r1 = r[0,0]**2 + r[0,1]**2
    r2 = r[1,0]**2 + r[1,1]**2
    r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2)
    deno = r12/(1+beta*r12)
    return exp(-0.5*alpha*(r1+r2)+deno)

#Local energy  for quantum dots in two dims, using analytical local energy
def LocalEnergy(r,alpha,beta):

    r1 = (r[0,0]**2 + r[0,1]**2)
    r2 = (r[1,0]**2 + r[1,1]**2)
    r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2)
    deno = 1.0/(1+beta*r12)
    deno2 = deno*deno
    return 0.5*(1-alpha*alpha)*(r1 + r2) +2.0*alpha + 1.0/r12+deno2*(alpha*r12-deno2+2*beta*deno-1

# The Monte Carlo sampling with the Metropolis algo
def MonteCarloSampling():

    NumberMCcycles= 100000
    StepSize = 1.0
    # positions
    PositionOld = np.zeros((NumberParticles,Dimension), np.double)
    PositionNew = np.zeros((NumberParticles,Dimension), np.double)
    # seed for rng generator
    seed()
    # start variational parameter
    alpha = 0.9
    for ia in range(MaxVariations):
        alpha += .025
        AlphaValues[ia] = alpha
        beta = 0.2
        for jb in range(MaxVariations):
            beta += .01
            BetaValues[jb] = beta
            energy = energy2 = 0.0
            DeltaE = 0.0
            #Initial position
            for i in range(NumberParticles):
                for j in range(Dimension):
                    PositionOld[i,j] = StepSize * (random() - .5)
            wfold = WaveFunction(PositionOld,alpha,beta)

            #Loop over MC MCcycles
            for MCcycle in range(NumberMCcycles):
```

```
                #Trial position
                for i in range(NumberParticles):
                    for j in range(Dimension):
                        PositionNew[i,j] = PositionOld[i,j] + StepSize * (random() - .5)
                wfnew = WaveFunction(PositionNew,alpha,beta)

                #Metropolis test to see whether we accept the move
                if random() < wfnew**2 / wfold**2:
                   PositionOld = PositionNew.copy()
                   wfold = wfnew
                   DeltaE = LocalEnergy(PositionOld,alpha,beta)
                energy += DeltaE
                energy2 += DeltaE**2

            #We calculate mean, variance and error ...
            energy /= NumberMCcycles
            energy2 /= NumberMCcycles
            variance = energy2 - energy**2
            error = sqrt(variance/NumberMCcycles)
            Energies[ia,jb] = energy
    return Energies, AlphaValues, BetaValues


#Here starts the main program with variable declarations
NumberParticles = 2
Dimension = 2
MaxVariations = 10
Energies = np.zeros((MaxVariations,MaxVariations))
AlphaValues = np.zeros(MaxVariations)
BetaValues = np.zeros(MaxVariations)
(Energies, AlphaValues, BetaValues) = MonteCarloSampling()

# Prepare for plots
fig = plt.figure()
ax = fig.gca(projection='3d')
# Plot the surface.
X, Y = np.meshgrid(AlphaValues, BetaValues)
surf = ax.plot_surface(X, Y, Energies,cmap=cm.coolwarm,linewidth=0, antialiased=False)
# Customize the z axis.
zmin = np.matrix(Energies).min()
zmax = np.matrix(Energies).max()
ax.set_zlim(zmin, zmax)
ax.set_xlabel(r'$\alpha$')
ax.set_ylabel(r'$\beta$')
ax.set_zlabel(r'$\langle E \rangle$')
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))
# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.show()
```

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

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

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 Metropolis algorithm

The Metropolis algorithm , see the original article 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 $A_{i\to j}$ and use it as the next sample. With probability $1 - A_{i\to j}$ 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 $\mathbf{P}_i^{(n\to\infty)} \to p_i$ 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:

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

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

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

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

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

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,

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

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

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

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

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

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 $\mathbf{g}$ 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

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

Use the C++ random class for random number generations

```
// Initialize the seed and call the Mersienne algo
 std::random_device rd;
 std::mt19937_64 gen(rd());
 // Set up the uniform distribution for x \in [[0, 1]
 std::uniform_real_distribution<double> UniformNumberGenerator(0.0,1.0);
 std::normal_distribution<double> Normaldistribution(0.0,1.0);
```

Use the C++ random class for RNGs, the Mersenne twister class

Finding the new position for importance sampling

```
for (int cycles = 1; cycles <= NumberMCsamples; cycles++){
   // new position
   for (int i = 0; i < NumberParticles; i++) {
     for (int j = 0; j < Dimension; j++) {
       // gaussian deviate to compute new positions using a given timestep
       NewPosition(i,j) = OldPosition(i,j) + Normaldistribution(gen)*sqrt(timestep)+OldQuantumFo

     }
```

Use the C++ random class for RNGs, the Metropolis test

Using the uniform distribution for the Metropolis test

```
//  Metropolis-Hastings algorithm
double GreensFunction = 0.0;
for (int j = 0; j < Dimension; j++) {
  GreensFunction += 0.5*(OldQuantumForce(i,j)+NewQuantumForce(i,j))*
    (D*timestep*0.5*(OldQuantumForce(i,j)-NewQuantumForce(i,j))-NewPosition(i,j)+OldPositio
}
GreensFunction = exp(GreensFunction);
// The Metropolis test is performed by moving one particle at the time
if(UniformNumberGenerator(gen) <= GreensFunction*NewWaveFunction*NewWaveFunction/OldWaveFunc
  for (int  j = 0; j < Dimension; j++) {
    OldPosition(i,j) = NewPosition(i,j);
    OldQuantumForce(i,j) = NewQuantumForce(i,j);
  }
  OldWaveFunction = NewWaveFunction;
}
```

Importance sampling, Fokker-Planck and Langevin equations

A stochastic process is simply a function of two variables, one is the time, the other is a stochastic variable X, defined by specifying

- the set $\{x\}$ of possible values for X;
- the probability distribution, $w_X(x)$, over this set, or briefly $w(x)$

The set of values $\{x\}$ for X may be discrete, or continuous. If the set of values is continuous, then $w_X(x)$ is a probability density so that $w_X(x)dx$ is the probability that one finds the stochastic variable X to have values in the range $[x, x + dx]$.

Importance sampling, Fokker-Planck and Langevin equations

An arbitrary number of other stochastic variables may be derived from X. For example, any Y given by a mapping of X, is also a stochastic variable.

The mapping may also be time-dependent, that is, the mapping depends on an additional variable t

$$Y_X(t) = f(X, t).$$

The quantity $Y_X(t)$ is called a random function, or, since t often is time, a stochastic process. A stochastic process is a function of two variables, one is the time, the other is a stochastic variable X. Let x be one of the possible values of X then

$$y(t) = f(x, t),$$

is a function of t, called a sample function or realization of the process. In physics one considers the stochastic process to be an ensemble of such sample functions.

Importance sampling, Fokker-Planck and Langevin equations

For many physical systems initial distributions of a stochastic variable y tend to equilibrium distributions: $w(y, t) \to w_0(y)$ as $t \to \infty$. In equilibrium detailed balance constrains the transition rates

$$W(y \to y')w(y) = W(y' \to y)w_0(y),$$

where $W(y' \to y)$ is the probability, per unit time, that the system changes from a state $|y\rangle$, characterized by the value y for the stochastic variable Y , to a state $|y'\rangle$.

Note that for a system in equilibrium the transition rate $W(y' \to y)$ and the reverse $W(y \to y')$ may be very different.

Importance sampling, Fokker-Planck and Langevin equations

Consider, for instance, a simple system that has only two energy levels $\epsilon_0 = 0$ and $\epsilon_1 = \Delta E$.

For a system governed by the Boltzmann distribution we find (the partition function has been taken out)

$$W(0 \to 1) \exp -(\epsilon_0/kT) = W(1 \to 0) \exp -(\epsilon_1/kT)$$

We get then

$$\frac{W(1 \to 0)}{W(0 \to 1)} = \exp -(\Delta E/kT),$$

which goes to zero when T tends to zero.

Importance sampling, Fokker-Planck and Langevin equations

If we assume a discrete set of events, our initial probability distribution function can be given by

$$w_i(0) = \delta_{i,0},$$

and its time-development after a given time step $\Delta t = \epsilon$ is

$$w_i(t) = \sum_j W(j \rightarrow i) w_j(t = 0).$$

The continuous analog to $w_i(0)$ is

$$w(\mathbf{x}) \rightarrow \delta(\mathbf{x}),$$

where we now have generalized the one-dimensional position x to a generic-dimensional vector $\mathbf{x}$. The Kroenecker δ function is replaced by the δ distribution function $\delta(\mathbf{x})$ at $t = 0$.

Importance sampling, Fokker-Planck and Langevin equations

The transition from a state j to a state i is now replaced by a transition to a state with position $\mathbf{y}$ from a state with position $\mathbf{x}$. The discrete sum of transition probabilities can then be replaced by an integral and we obtain the new distribution at a time $t + \Delta t$ as

$$w(\mathbf{y}, t + \Delta t) = \int W(\mathbf{y}, t + \Delta t | \mathbf{x}, t) w(\mathbf{x}, t) d\mathbf{x},$$

and after m time steps we have

$$w(\mathbf{y}, t + m\Delta t) = \int W(\mathbf{y}, t + m\Delta t | \mathbf{x}, t) w(\mathbf{x}, t) d\mathbf{x}.$$

When equilibrium is reached we have

$$w(\mathbf{y}) = \int W(\mathbf{y} | \mathbf{x}, t) w(\mathbf{x}) d\mathbf{x},$$

that is no time-dependence. Note our change of notation for W

Importance sampling, Fokker-Planck and Langevin equations

We can solve the equation for $w(\mathbf{y}, t)$ by making a Fourier transform to momentum space. The PDF $w(\mathbf{x}, t)$ is related to its Fourier transform $\tilde{w}(\mathbf{k}, t)$ through

$$w(\mathbf{x}, t) = \int_{-\infty}^{\infty} d\mathbf{k} \exp(i\mathbf{k}\mathbf{x}) \tilde{w}(\mathbf{k}, t),$$

and using the definition of the δ-function

$$\delta(\mathbf{x}) = \frac{1}{2\pi}\int_{-\infty}^{\infty} d\mathbf{k}\exp{(i\mathbf{kx})},$$

we see that

$$\tilde{w}(\mathbf{k},0) = 1/2\pi.$$

Importance sampling, Fokker-Planck and Langevin equations

We can then use the Fourier-transformed diffusion equation

$$\frac{\partial \tilde{w}(\mathbf{k},t)}{\partial t} = -D\mathbf{k}^2\tilde{w}(\mathbf{k},t),$$

with the obvious solution

$$\tilde{w}(\mathbf{k},t) = \tilde{w}(\mathbf{k},0)\exp\left[-(D\mathbf{k}^2t)\right) = \frac{1}{2\pi}\exp\left[-(D\mathbf{k}^2t)\right].$$

Importance sampling, Fokker-Planck and Langevin equations

With the Fourier transform we obtain

$$w(\mathbf{x},t) = \int_{-\infty}^{\infty} d\mathbf{k}\exp\left[i\mathbf{kx}\right]\frac{1}{2\pi}\exp\left[-(D\mathbf{k}^2t)\right] = \frac{1}{\sqrt{4\pi Dt}}\exp\left[-(\mathbf{x}^2/4Dt)\right],$$

with the normalization condition

$$\int_{-\infty}^{\infty} w(\mathbf{x},t)d\mathbf{x} = 1.$$

Importance sampling, Fokker-Planck and Langevin equations

The solution represents the probability of finding our random walker at position $\mathbf{x}$ at time t if the initial distribution was placed at $\mathbf{x} = 0$ at $t = 0$.

There is another interesting feature worth observing. The discrete transition probability W itself is given by a binomial distribution. The results from the central limit theorem state that transition probability in the limit $n \to \infty$ converges to the normal distribution. It is then possible to show that

$$W(il - jl, n\epsilon) \to W(\mathbf{y}, t + \Delta t|\mathbf{x}, t) = \frac{1}{\sqrt{4\pi D\Delta t}}\exp\left[-((\mathbf{y}-\mathbf{x})^2/4D\Delta t)\right],$$

and that it satisfies the normalization condition and is itself a solution to the diffusion equation.

Importance sampling, Fokker-Planck and Langevin equations

Let us now assume that we have three PDFs for times $t_0 < t' < t$, that is $w(\mathbf{x}_0, t_0)$, $w(\mathbf{x}', t')$ and $w(\mathbf{x}, t)$. We have then

$$w(\mathbf{x}, t) = \int_{-\infty}^{\infty} W(\mathbf{x}.t|\mathbf{x}'.t')w(\mathbf{x}', t')d\mathbf{x}',$$

and

$$w(\mathbf{x}, t) = \int_{-\infty}^{\infty} W(\mathbf{x}.t|\mathbf{x}_0.t_0)w(\mathbf{x}_0, t_0)d\mathbf{x}_0,$$

and

$$w(\mathbf{x}', t') = \int_{-\infty}^{\infty} W(\mathbf{x}'.t'|\mathbf{x}_0, t_0)w(\mathbf{x}_0, t_0)d\mathbf{x}_0.$$

Importance sampling, Fokker-Planck and Langevin equations

We can combine these equations and arrive at the famous Einstein-Smoluchenski-Kolmogorov-Chapman (ESKC) relation

$$W(\mathbf{x}t|\mathbf{x}_0t_0) = \int_{-\infty}^{\infty} W(\mathbf{x}, t|\mathbf{x}', t')W(\mathbf{x}', t'|\mathbf{x}_0, t_0)d\mathbf{x}'.$$

We can replace the spatial dependence with a dependence upon say the velocity (or momentum), that is we have

$$W(\mathbf{v}, t|\mathbf{v}_0, t_0) = \int_{-\infty}^{\infty} W(\mathbf{v}, t|\mathbf{v}', t')W(\mathbf{v}', t'|\mathbf{v}_0, t_0)d\mathbf{x}'.$$

Importance sampling, Fokker-Planck and Langevin equations

We will now derive the Fokker-Planck equation. We start from the ESKC equation

$$W(\mathbf{x}, t|\mathbf{x}_0, t_0) = \int_{-\infty}^{\infty} W(\mathbf{x}, t|\mathbf{x}', t')W(\mathbf{x}', t'|\mathbf{x}_0, t_0)d\mathbf{x}'.$$

Define $s = t' - t_0$, $\tau = t - t'$ and $t - t_0 = s + \tau$. We have then

$$W(\mathbf{x}, s + \tau|\mathbf{x}_0) = \int_{-\infty}^{\infty} W(\mathbf{x}, \tau|\mathbf{x}')W(\mathbf{x}', s|\mathbf{x}_0)d\mathbf{x}'.$$

Importance sampling, Fokker-Planck and Langevin equations

Assume now that τ is very small so that we can make an expansion in terms of a small step xi, with $\mathbf{x}' = \mathbf{x} - \xi$, that is

$$W(\mathbf{x}, s|\mathbf{x}_0) + \frac{\partial W}{\partial s}\tau + O(\tau^2) = \int_{-\infty}^{\infty} W(\mathbf{x}, \tau|\mathbf{x} - \xi)W(\mathbf{x} - \xi, s|\mathbf{x}_0)d\mathbf{x}'.$$

We assume that $W(\mathbf{x}, \tau|\mathbf{x} - \xi)$ takes non-negligible values only when ξ is small. This is just another way of stating the Master equation!!

Importance sampling, Fokker-Planck and Langevin equations

We say thus that $\mathbf{x}$ changes only by a small amount in the time interval τ. This means that we can make a Taylor expansion in terms of ξ, that is we expand

$$W(\mathbf{x}, \tau|\mathbf{x} - \xi)W(\mathbf{x} - \xi, s|\mathbf{x}_0) = \sum_{n=0}^{\infty} \frac{(-\xi)^n}{n!}\frac{\partial^n}{\partial x^n}\left[W(\mathbf{x} + \xi, \tau|\mathbf{x})W(\mathbf{x}, s|\mathbf{x}_0)\right].$$

Importance sampling, Fokker-Planck and Langevin equations

We can then rewrite the ESKC equation as

$$\frac{\partial W}{\partial s}\tau = -W(\mathbf{x}, s|\mathbf{x}_0) + \sum_{n=0}^{\infty} \frac{(-\xi)^n}{n!}\frac{\partial^n}{\partial x^n}\left[W(\mathbf{x}, s|\mathbf{x}_0)\int_{-\infty}^{\infty}\xi^n W(\mathbf{x} + \xi, \tau|\mathbf{x})d\xi\right].$$

We have neglected higher powers of τ and have used that for $n = 0$ we get simply $W(\mathbf{x}, s|\mathbf{x}_0)$ due to normalization.

Importance sampling, Fokker-Planck and Langevin equations

We say thus that $\mathbf{x}$ changes only by a small amount in the time interval τ. This means that we can make a Taylor expansion in terms of ξ, that is we expand

$$W(\mathbf{x}, \tau|\mathbf{x} - \xi)W(\mathbf{x} - \xi, s|\mathbf{x}_0) = \sum_{n=0}^{\infty} \frac{(-\xi)^n}{n!}\frac{\partial^n}{\partial x^n}\left[W(\mathbf{x} + \xi, \tau|\mathbf{x})W(\mathbf{x}, s|\mathbf{x}_0)\right].$$

Importance sampling, Fokker-Planck and Langevin equations

We can then rewrite the ESKC equation as

$$\frac{\partial W(\mathbf{x}, s|\mathbf{x}_0)}{\partial s}\tau = -W(\mathbf{x}, s|\mathbf{x}_0) + \sum_{n=0}^{\infty} \frac{(-\xi)^n}{n!}\frac{\partial^n}{\partial x^n}\left[W(\mathbf{x}, s|\mathbf{x}_0)\int_{-\infty}^{\infty}\xi^n W(\mathbf{x} + \xi, \tau|\mathbf{x})d\xi\right].$$

We have neglected higher powers of τ and have used that for $n = 0$ we get simply $W(\mathbf{x}, s|\mathbf{x}_0)$ due to normalization.

Importance sampling, Fokker-Planck and Langevin equations

We simplify the above by introducing the moments

$$M_n = \frac{1}{\tau}\int_{-\infty}^{\infty} \xi^n W(\mathbf{x}+\xi, \tau|\mathbf{x})d\xi = \frac{\langle[\Delta x(\tau)]^n\rangle}{\tau},$$

resulting in

$$\frac{\partial W(\mathbf{x}, s|\mathbf{x}_0)}{\partial s} = \sum_{n=1}^{\infty}\frac{(-\xi)^n}{n!}\frac{\partial^n}{\partial x^n}\left[W(\mathbf{x}, s|\mathbf{x}_0)M_n\right].$$

Importance sampling, Fokker-Planck and Langevin equations

When $\tau \to 0$ we assume that $\langle[\Delta x(\tau)]^n\rangle \to 0$ more rapidly than τ itself if $n > 2$. When τ is much larger than the standard correlation time of system then M_n for $n > 2$ can normally be neglected. This means that fluctuations become negligible at large time scales.

If we neglect such terms we can rewrite the ESKC equation as

$$\frac{\partial W(\mathbf{x}, s|\mathbf{x}_0)}{\partial s} = -\frac{\partial M_1 W(\mathbf{x}, s|\mathbf{x}_0)}{\partial x} + \frac{1}{2}\frac{\partial^2 M_2 W(\mathbf{x}, s|\mathbf{x}_0)}{\partial x^2}.$$

Importance sampling, Fokker-Planck and Langevin equations

In a more compact form we have

$$\frac{\partial W}{\partial s} = -\frac{\partial M_1 W}{\partial x} + \frac{1}{2}\frac{\partial^2 M_2 W}{\partial x^2},$$

which is the Fokker-Planck equation! It is trivial to replace position with velocity (momentum).

Importance sampling, Fokker-Planck and Langevin equations

Langevin equation. Consider a particle suspended in a liquid. On its path through the liquid it will continuously collide with the liquid molecules. Because on average the particle will collide more often on the front side than on the back side, it will experience a systematic force proportional with its velocity, and directed opposite to its velocity. Besides this systematic force the particle will experience a stochastic force $\mathbf{F}(t)$. The equations of motion are

- $\frac{d\mathbf{r}}{dt} = \mathbf{v}$ and
- $\frac{d\mathbf{v}}{dt} = -\xi\mathbf{v} + \mathbf{F}$.

Importance sampling, Fokker-Planck and Langevin equations

Langevin equation. From hydrodynamics we know that the friction constant ξ is given by

$$\xi = 6\pi\eta a/m$$

where η is the viscosity of the solvent and a is the radius of the particle .

Solving the second equation in the previous slide we get

$$\mathbf{v}(t) = \mathbf{v}_0 e^{-\xi t} + \int_0^t d\tau e^{-\xi(t-\tau)}\mathbf{F}(\tau).$$

Importance sampling, Fokker-Planck and Langevin equations

Langevin equation. If we want to get some useful information out of this, we have to average over all possible realizations of $\mathbf{F}(t)$, with the initial velocity as a condition. A useful quantity for example is

$$\langle \mathbf{v}(t)\cdot\mathbf{v}(t)\rangle_{\mathbf{v}_0} = v_0^{-\xi 2t} + 2\int_0^t d\tau e^{-\xi(2t-\tau)}\mathbf{v}_0\cdot\langle\mathbf{F}(\tau)\rangle_{\mathbf{v}_0}$$

$$+\int_0^t d\tau' \int_0^t d\tau e^{-\xi(2t-\tau-\tau')}\langle\mathbf{F}(\tau)\cdot\mathbf{F}(\tau')\rangle_{\mathbf{v}_0}.$$

Importance sampling, Fokker-Planck and Langevin equations

Langevin equation. In order to continue we have to make some assumptions about the conditional averages of the stochastic forces. In view of the chaotic character of the stochastic forces the following assumptions seem to be appropriate

$$\langle\mathbf{F}(t)\rangle = 0,$$

and

$$\langle\mathbf{F}(t)\cdot\mathbf{F}(t')\rangle_{\mathbf{v}_0} = C_{\mathbf{v}_0}\delta(t-t').$$

We omit the subscript $\mathbf{v}_0$, when the quantity of interest turns out to be independent of $\mathbf{v}_0$. Using the last three equations we get

$$\langle\mathbf{v}(t)\cdot\mathbf{v}(t)\rangle_{\mathbf{v}_0} = v_0^2 e^{-2\xi t} + \frac{C_{\mathbf{v}_0}}{2\xi}(1-e^{-2\xi t}).$$

For large t this should be equal to 3kT/m, from which it follows that

$$\langle \mathbf{F}(t) \cdot \mathbf{F}(t') \rangle = 6\frac{kT}{m}\xi\delta(t-t').$$

This result is called the fluctuation-dissipation theorem .

Importance sampling, Fokker-Planck and Langevin equations

Langevin equation. Integrating

$$\mathbf{v}(t) = \mathbf{v}_0 e^{-\xi t} + \int_0^t d\tau e^{-\xi(t-\tau)}\mathbf{F}(\tau),$$

we get

$$\mathbf{r}(t) = \mathbf{r}_0 + \mathbf{v}_0\frac{1}{\xi}(1-e^{-\xi t}) + \int_0^t d\tau \int_0^\tau \tau' e^{-\xi(\tau-\tau')}\mathbf{F}(\tau'),$$

from which we calculate the mean square displacement

$$\langle (\mathbf{r}(t)-\mathbf{r}_0)^2\rangle_{\mathbf{v}_0} = \frac{v_0^2}{\xi}(1-e^{-\xi t})^2 + \frac{3kT}{m\xi^2}(2\xi t - 3 + 4e^{-\xi t} - e^{-2\xi t}).$$

Importance sampling, Fokker-Planck and Langevin equations

Langevin equation. For very large t this becomes

$$\langle (\mathbf{r}(t)-\mathbf{r}_0)^2\rangle = \frac{6kT}{m\xi}t$$

from which we get the Einstein relation

$$D = \frac{kT}{m\xi}$$

where we have used $\langle (\mathbf{r}(t)-\mathbf{r}_0)^2\rangle = 6Dt$.

Code example for two electrons in a quantum dots

```
# 2-electron VMC code for 2dim quantum dot with importance sampling
# Using gaussian rng for new positions and Metropolis- Hastings
# No energy minimization
from math import exp, sqrt
from random import random, seed, normalvariate
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import sys
```

```
from numba import jit,njit


#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')
outfile = open(outfilename,'w')

# Trial wave function for the 2-electron quantum dot in two dims
def WaveFunction(r,alpha,beta):
    r1 = r[0,0]**2 + r[0,1]**2
    r2 = r[1,0]**2 + r[1,1]**2
    r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2)
    deno = r12/(1+beta*r12)
    return exp(-0.5*alpha*(r1+r2)+deno)

# Local energy  for the 2-electron quantum dot in two dims, using analytical local energy
def LocalEnergy(r,alpha,beta):

    r1 = (r[0,0]**2 + r[0,1]**2)
    r2 = (r[1,0]**2 + r[1,1]**2)
    r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2)
    deno = 1.0/(1+beta*r12)
    deno2 = deno*deno
    return 0.5*(1-alpha*alpha)*(r1 + r2) +2.0*alpha + 1.0/r12+deno2*(alpha*r12-deno2+2*beta*deno-1

# Setting up the quantum force for the two-electron quantum dot, recall that it is a vector
def QuantumForce(r,alpha,beta):

    qforce = np.zeros((NumberParticles,Dimension), np.double)
    r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2)
    deno = 1.0/(1+beta*r12)
    qforce[0,:] = -2*r[0,:]*alpha*(r[0,:]-r[1,:])*deno*deno/r12
    qforce[1,:] = -2*r[1,:]*alpha*(r[1,:]-r[0,:])*deno*deno/r12
    return qforce

# The Monte Carlo sampling with the Metropolis algo
# jit decorator tells Numba to compile this function.
# The argument types will be inferred by Numba when function is called.
@jit()
def MonteCarloSampling():

    NumberMCcycles= 100000
    # Parameters in the Fokker-Planck simulation of the quantum force
    D = 0.5
    TimeStep = 0.05
    # positions
    PositionOld = np.zeros((NumberParticles,Dimension), np.double)
    PositionNew = np.zeros((NumberParticles,Dimension), np.double)
    # Quantum force
    QuantumForceOld = np.zeros((NumberParticles,Dimension), np.double)
    QuantumForceNew = np.zeros((NumberParticles,Dimension), np.double)

    # seed for rng generator
    seed()
    # start variational parameter  loops, two parameters here
    alpha = 0.9
    for ia in range(MaxVariations):
        alpha += .025
```

```
            AlphaValues[ia] = alpha
            beta = 0.2
            for jb in range(MaxVariations):
                beta += .01
                BetaValues[jb] = beta
                energy = energy2 = 0.0
                DeltaE = 0.0
                #Initial position
                for i in range(NumberParticles):
                    for j in range(Dimension):
                        PositionOld[i,j] = normalvariate(0.0,1.0)*sqrt(TimeStep)
                wfold = WaveFunction(PositionOld,alpha,beta)
                QuantumForceOld = QuantumForce(PositionOld,alpha, beta)

                #Loop over MC MCcycles
                for MCcycle in range(NumberMCcycles):
                    #Trial position moving one particle at the time
                    for i in range(NumberParticles):
                        for j in range(Dimension):
                            PositionNew[i,j] = PositionOld[i,j]+normalvariate(0.0,1.0)*sqrt(TimeStep)+
                                               QuantumForceOld[i,j]*TimeStep*D
                        wfnew = WaveFunction(PositionNew,alpha,beta)
                        QuantumForceNew = QuantumForce(PositionNew,alpha, beta)
                        GreensFunction = 0.0
                        for j in range(Dimension):
                            GreensFunction += 0.5*(QuantumForceOld[i,j]+QuantumForceNew[i,j])*\
                                          (D*TimeStep*0.5*(QuantumForceOld[i,j]-QuantumForceNew[i,j])-
                                          PositionNew[i,j]+PositionOld[i,j])

                        GreensFunction = exp(GreensFunction)
                        ProbabilityRatio = GreensFunction*wfnew**2/wfold**2
                        #Metropolis-Hastings test to see whether we accept the move
                        if random() <= ProbabilityRatio:
                            for j in range(Dimension):
                                PositionOld[i,j] = PositionNew[i,j]
                                QuantumForceOld[i,j] = QuantumForceNew[i,j]
                            wfold = wfnew
                        DeltaE = LocalEnergy(PositionOld,alpha,beta)
                        energy += DeltaE
                        energy2 += DeltaE**2
                # We calculate mean, variance and error (no blocking applied)
                energy /= NumberMCcycles
                energy2 /= NumberMCcycles
                variance = energy2 - energy**2
                error = sqrt(variance/NumberMCcycles)
                Energies[ia,jb] = energy
                outfile.write('%f %f %f %f %f\n' %(alpha,beta,energy,variance,error))
        return Energies, AlphaValues, BetaValues


    #Here starts the main program with variable declarations
    NumberParticles = 2
    Dimension = 2
    MaxVariations = 10
    Energies = np.zeros((MaxVariations,MaxVariations))
    AlphaValues = np.zeros(MaxVariations)
    BetaValues = np.zeros(MaxVariations)
    (Energies, AlphaValues, BetaValues) = MonteCarloSampling()
    outfile.close()
    # Prepare for plots
    fig = plt.figure()
```

```
ax = fig.gca(projection='3d')
# Plot the surface.
X, Y = np.meshgrid(AlphaValues, BetaValues)
surf = ax.plot_surface(X, Y, Energies,cmap=cm.coolwarm,linewidth=0, antialiased=False)
# Customize the z axis.
zmin = np.matrix(Energies).min()
zmax = np.matrix(Energies).max()
ax.set_zlim(zmin, zmax)
ax.set_xlabel(r'$\alpha$')
ax.set_ylabel(r'$\beta$')
ax.set_zlabel(r'$\langle E \rangle$')
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))
# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.show()
```

Bringing the gradient optmization

The simple one-particle case in a harmonic oscillator trap

```
# Gradient descent stepping with analytical derivative
import numpy as np
from scipy.optimize import minimize
def DerivativeE(x):
    return x-1.0/(4*x*x*x);

def Energy(x):
   return x*x*0.5+1.0/(8*x*x);
x0 = 1.0
eta = 0.1
Niterations = 100

for iter in range(Niterations):
    gradients = DerivativeE(x0)
    x0 -= eta*gradients

print(x0)
```

And then for the non-interacting two-particle case

```
# 2-electron VMC code for 2dim quantum dot with importance sampling
# Using gaussian rng for new positions and Metropolis- Hastings
from math import exp, sqrt
from random import random, seed, normalvariate
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import sys
from numba import jit


# Trial wave function for the 2-electron quantum dot in two dims
def WaveFunction(r,alpha):
    r1 = r[0,0]**2 + r[0,1]**2
    r2 = r[1,0]**2 + r[1,1]**2
    return exp(-0.5*alpha*(r1+r2))
```

```
# Local energy  for the 2-electron quantum dot in two dims, using analytical local energy
def LocalEnergy(r,alpha):

    r1 = (r[0,0]**2 + r[0,1]**2)
    r2 = (r[1,0]**2 + r[1,1]**2)
    return 0.5*(1-alpha*alpha)*(r1 + r2) +2.0*alpha

# Derivate of wave function ansatz as function of variational parameters
def DerivativeWFansatz(r,alpha):

    r1 = (r[0,0]**2 + r[0,1]**2)
    r2 = (r[1,0]**2 + r[1,1]**2)
    WfDer = -(r1+r2)
    return  WfDer

# Setting up the quantum force for the two-electron quantum dot, recall that it is a vector
def QuantumForce(r,alpha):

    qforce = np.zeros((NumberParticles,Dimension), np.double)
    qforce[0,:] = -2*r[0,:]*alpha
    qforce[1,:] = -2*r[1,:]*alpha
    return qforce

# Computing the derivative of the energy and the energy
# jit decorator tells Numba to compile this function.
# The argument types will be inferred by Numba when function is called.
@jit
def EnergyMinimization(alpha):

    NumberMCcycles= 1000
    # Parameters in the Fokker-Planck simulation of the quantum force
    D = 0.5
    TimeStep = 0.05
    # positions
    PositionOld = np.zeros((NumberParticles,Dimension), np.double)
    PositionNew = np.zeros((NumberParticles,Dimension), np.double)
    # Quantum force
    QuantumForceOld = np.zeros((NumberParticles,Dimension), np.double)
    QuantumForceNew = np.zeros((NumberParticles,Dimension), np.double)

    # seed for rng generator
    seed()
    energy = 0.0
    DeltaE = 0.0
    EnergyDer = 0.0
    DeltaPsi = 0.0
    DerivativePsiE = 0.0
    #Initial position
    for i in range(NumberParticles):
        for j in range(Dimension):
            PositionOld[i,j] = normalvariate(0.0,1.0)*sqrt(TimeStep)
    wfold = WaveFunction(PositionOld,alpha)
    QuantumForceOld = QuantumForce(PositionOld,alpha)

    #Loop over MC MCcycles
    for MCcycle in range(NumberMCcycles):
        #Trial position moving one particle at the time
        for i in range(NumberParticles):
            for j in range(Dimension):
                PositionNew[i,j] = PositionOld[i,j]+normalvariate(0.0,1.0)*sqrt(TimeStep)+\
```

```
                                    QuantumForceOld[i,j]*TimeStep*D
            wfnew = WaveFunction(PositionNew,alpha)
            QuantumForceNew = QuantumForce(PositionNew,alpha)
            GreensFunction = 0.0
            for j in range(Dimension):
                GreensFunction += 0.5*(QuantumForceOld[i,j]+QuantumForceNew[i,j])*\
                                  (D*TimeStep*0.5*(QuantumForceOld[i,j]-QuantumForceNew[i,j])-
                                  PositionNew[i,j]+PositionOld[i,j])

            GreensFunction = exp(GreensFunction)
            ProbabilityRatio = GreensFunction*wfnew**2/wfold**2
            #Metropolis-Hastings test to see whether we accept the move
            if random() <= ProbabilityRatio:
                for j in range(Dimension):
                    PositionOld[i,j] = PositionNew[i,j]
                    QuantumForceOld[i,j] = QuantumForceNew[i,j]
                wfold = wfnew
        DeltaE = LocalEnergy(PositionOld,alpha)
        DeltaPsi = DerivativeWFansatz(PositionOld,alpha)
        energy += DeltaE
        DerivativePsiE += DeltaPsi*DeltaE

    # We calculate mean, variance and error (no blocking applied)
    energy /= NumberMCcycles
    DerivativePsiE /= NumberMCcycles
    DeltaPsi /= NumberMCcycles
    EnergyDer  = 2*(DerivativePsiE-DeltaPsi*energy)
    return energy, EnergyDer


#Here starts the main program with variable declarations
NumberParticles = 2
Dimension = 2
# guess for variational parameters
x0 = 1.5
# Set up iteration using stochastic gradient method
Energy =0 ; EnergyDer = 0
Energy, EnergyDer = EnergyMinimization(x0)
print(Energy, EnergyDer)

eta = 0.01
Niterations = 100

for iter in range(Niterations):
    gradients = EnergyDer
    x0 -= eta*gradients
    Energy, EnergyDer = EnergyMinimization(x0)

print(x0)
```

Project 2, VMC for fermions: 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) $|\hat{D}|$:

$$d_{ij}^{-1} = \frac{C_{ji}}{|\hat{D}|} \tag{7}$$

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 $\hat{D}^{-1}$:

$$\sum_{k=1}^{N} d_{ik} d_{kj}^{-1} = \delta_{ij} \tag{8}$$

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

$$R \equiv \frac{|\hat{D}(\mathbf{r}^{\text{new}})|}{|\hat{D}(\mathbf{r}^{\text{old}})|} = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})\, C_{ij}(\mathbf{r}^{\text{new}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}})\, C_{ij}(\mathbf{r}^{\text{old}})} \tag{9}$$

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:

$$C_{ij}(\mathbf{r}^{\text{new}}) = C_{ij}(\mathbf{r}^{\text{old}}) \quad \forall\ j \in \{1, \ldots, N\} \tag{10}$$

Efficient calculation of Slater determinants

Inserting this into the numerator of eq. (9) and using eq. (7) to substitute the cofactors with the elements of the inverse matrix, we get:

$$R = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})\, C_{ij}(\mathbf{r}^{\text{old}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}})\, C_{ij}(\mathbf{r}^{\text{old}})} = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}})} \tag{11}$$

Efficient calculation of Slater determinants

Now by eq. (8) the denominator of the rightmost expression must be unity, so that we finally arrive at:

$$R = \sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}}) = \sum_{j=1}^{N} \phi_j(\mathbf{r}_i^{\text{new}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}}) \tag{12}$$

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 $(\phi_1(\mathbf{r}_i^{\text{new}}), \ldots, \phi_N(\mathbf{r}_i^{\text{new}}))$ 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 $\hat{D}^{-1}(\mathbf{x}^{\text{old}})$.

Efficient calculation of Slater determinants

If the new position $\mathbf{r}^{\text{new}}$ 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:

$$S_j = (\hat{D}(\mathbf{r}^{\text{new}}) \times \hat{D}^{-1}(\mathbf{r}^{\text{old}}))_{ij} = \sum_{l=1}^{N} d_{il}(\mathbf{r}^{\text{new}})\, d_{lj}^{-1}(\mathbf{r}^{\text{old}}) \tag{13}$$

Efficient calculation of Slater determinants

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

$$d_{kj}^{-1}(\mathbf{r}^{\text{new}}) = d_{kj}^{-1}(\mathbf{r}^{\text{old}}) - \frac{S_j}{R}\, d_{ki}^{-1}(\mathbf{r}^{\text{old}}) \qquad \begin{array}{l} \forall \;\; k \in \{1, \ldots, N\} \\ j \neq i \end{array} \tag{14}$$

Efficient calculation of Slater determinants

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

$$d_{ki}^{-1}(\mathbf{r}^{\text{new}}) = \frac{1}{R}\, d_{ki}^{-1}(\mathbf{r}^{\text{old}}) \quad \forall \;\; k \in \{1, \ldots, N\} \tag{15}$$

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

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

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

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

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:

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

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:

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

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

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

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

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

$$K_i = -\frac{1}{2}\frac{\nabla_i^2 \Psi}{\Psi}. \tag{16}$$

$$\begin{aligned} \frac{\nabla^2\Psi}{\Psi} &= \frac{\nabla^2(\Psi_D\,\Psi_C)}{\Psi_D\,\Psi_C} = \frac{\nabla\cdot[\nabla(\Psi_D\,\Psi_C)]}{\Psi_D\,\Psi_C} = \frac{\nabla\cdot[\Psi_C\nabla\Psi_D + \Psi_D\nabla\Psi_C]}{\Psi_D\,\Psi_C} \\ &= \frac{\nabla\Psi_C\cdot\nabla\Psi_D + \Psi_C\nabla^2\Psi_D + \nabla\Psi_D\cdot\nabla\Psi_C + \Psi_D\nabla^2\Psi_C}{\Psi_D\,\Psi_C} \end{aligned} \tag{17}$$

$$\frac{\nabla^2\Psi}{\Psi} = \frac{\nabla^2\Psi_D}{\Psi_D} + \frac{\nabla^2\Psi_C}{\Psi_C} + 2\frac{\nabla\Psi_D}{\Psi_D}\cdot\frac{\nabla\Psi_C}{\Psi_C} \tag{18}$$

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

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

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

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

and $g'(r_{kj}) = dg(r_{kj})/dr_{kj}$ and $g''(r_{kj}) = d^2 g(r_{kj})/dr_{kj}^2$ 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 $\hat{D}^{-1}(\mathbf{x}^{\mathrm{old}})$.

Single-particle states

The $1s$ hydrogen like wave function

$$R_{10}(r) = 2\left(\frac{Z}{a_0}\right)^{3/2} \exp\left(-Zr/a_0\right) = u_{10}/r$$

The total energy for helium (not the Hartree or Fock terms) from the direct and the exchange term should give $5Z/8$.

The single-particle energy with no interactions should give $-Z^2/2n^2$.

Single-particle states

The $2s$ hydrogen-like wave function is

$$R_{20}(r) = 2\left(\frac{Z}{2a_0}\right)^{3/2} \left(1-\frac{Zr}{2a_0}\right) \exp\left(-Zr/2a_0\right) = u_{20}/r$$

and the $2p$ hydrogen -like wave function is

$$R_{21}(r) = \frac{1}{\sqrt{3}}\left(\frac{Z}{2a_0}\right)^{3/2}\frac{Zr}{a_0}\exp\left(-Zr/2a_0\right) = u_{21}/r$$

We use $a_0 = 1$.

Problems with neon states for VMC

In the standard textbook case one uses spherical coordinates in order to get the hydrogen-like wave functions

$$x = rsin\theta cos\phi,$$

$$y = rsin\theta sin\phi,$$

and

$$z = rcos\theta.$$

Problems with neon states for VMC

The reason we introduce spherical coordinates is the spherical symmetry of the Coulomb potential

$$\frac{e^2}{4\pi\epsilon_0 r} = \frac{e^2}{4\pi\epsilon_0\sqrt{x^2+y^2+z^2}},$$

where we have used $r = \sqrt{x^2+y^2+z^2}$. It is not possible to find a separable solution of the type

$$\psi(x,y,z) = \psi(x)\psi(y)\psi(z).$$

However, with spherical coordinates we can find a solution of the form

$$\psi(r,\theta,\phi) = R(r)P(\theta)F(\phi).$$

Spherical harmonics

The angle-dependent differential equations result in the spherical harmonic functions as solutions, with quantum numbers l and m_l. These functions are given by

$$Y_{lm_l}(\theta,\phi) = P(\theta)F(\phi) = \sqrt{\frac{(2l+1)(l-m_l)!}{4\pi(l+m_l)!}}P_l^{m_l}(cos(\theta))\exp\left(im_l\phi\right),$$

with $P_l^{m_l}$ being the associated Legendre polynomials They can be rewritten as

$$Y_{lm_l}(\theta,\phi) = sin^{|m_l|}(\theta)\times(\text{polynom}(cos\theta))\exp\left(im_l\phi\right),$$

Examples of spherical harmonics

We have the following selected examples

$$Y_{00} = \sqrt{\frac{1}{4\pi}},$$

for $l = m_l = 0$,

$$Y_{10} = \sqrt{\frac{3}{4\pi}} cos(\theta),$$

for $l = 1$ og $m_l = 0$,

$$Y_{1\pm1} = \sqrt{\frac{3}{8\pi}} sin(\theta) exp(\pm i\phi),$$

for $l = 1$ og $m_l = \pm 1$.

Problems with spherical harmonics

A problem with the spherical harmonics is that they are complex. The introduction of *solid harmonics* allows the use of real orbital wave-functions for a wide range of applications. The complex solid harmonics $\mathbf{Y}_{lm_l}(\mathbf{r})$ are related to the spherical harmonics $Y_{lm_l}(\mathbf{r})$ through

$$\mathbf{Y}_{lm_l}(\mathbf{r}) = r^l Y_{lm_l}(\mathbf{r}).$$

By factoring out the leading r-dependency of the radial-function

$$\mathbf{R}_{nl}(\mathbf{r}) = r^{-l} R_{nl}(\mathbf{r}),$$

we obtain

$$\Psi_{nlm_l}(r, \theta, \phi) = \mathbf{R}_{nl}(\mathbf{r}) \cdot \mathbf{Y}_{lm_l}(\mathbf{r}).$$

Real solid harmonics

For the theoretical development of the *real solid harmonics* we first express the complex solid harmonics, C_{lm_l}, by (complex) Cartesian coordinates, and arrive at the real solid harmonics, S_{lm_l}, through the unitary transformation

$$\begin{pmatrix} S_{lm_l} \\ S_{l,-m_l} \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} (-1)^m_l & 1 \\ -(-1)^m_l i & i \end{pmatrix} \begin{pmatrix} C_{lm_l} \\ C_{l,-m_l} \end{pmatrix}.$$

Solid harmonics

This transformation will not alter any physical quantities that are degenerate in the subspace consisting of opposite magnetic quantum numbers (the angular momentum l is equal for both these cases). This means for example that the above transformation does not alter the energies, unless an external magnetic field is applied to the system. Henceforth, we will use the solid harmonics, and note that changing the spherical potential beyond the Coulomb potential will not alter the solid harmonics.

Relation between solid harmonics and spherical harmonics

We have defined

$$\mathbf{Y}_{lm_l}(\mathbf{r}) = r^l Y_{lm_l}(\mathbf{r}).$$

The real-valued spherical harmonics are defined as

$$S_{l0} = \sqrt{\frac{4\pi}{2l+1}}\mathbf{Y}_{l0}(\mathbf{r}),$$

$$S_{lm_l} = (-1)^{m_l}\sqrt{\frac{8\pi}{2l+1}}\mathrm{Re}\mathbf{Y}_{l0}(\mathbf{r}),$$

$$S_{lm_l} = (-1)^{m_l}\sqrt{\frac{8\pi}{2l+1}}\mathrm{Im}\mathbf{Y}_{l0}(\mathbf{r}),$$

for $m_l > 0$.

The lowest-order real solid harmonics

$m_l \backslash l$	0	1	2	3
+3				$\frac{1}{2}\sqrt{\frac{5}{2}}(x^2-3y^2)x$
+2			$\frac{1}{2}\sqrt{3}(x^2-y^2)$	$\frac{1}{2}\sqrt{15}(x^2-y^2)z$
+1		x	$\sqrt{3}xz$	$\frac{1}{2}\sqrt{\frac{3}{2}}(5z^2-r^2)x$
0		y	$\frac{1}{2}(3z^2-r^2)$	$\frac{1}{2}(5z^2-3r^2)x$
-1		z	$\sqrt{3}yz$	$\frac{1}{2}\sqrt{\frac{3}{2}}(5z^2-r^2)y$
-2			$\sqrt{3}xy$	$\sqrt{15}xyz$
-3				$\frac{1}{2}\sqrt{\frac{5}{2}}(3x^2-y^2)y$

Proof for updating algorithm for Slater determinant

As a starting point we may consider that each time a new position is suggested in the Metropolis algorithm, a row of the current Slater matrix experiences some kind of perturbation. Hence, the Slater matrix with its orbitals evaluated at the new position equals the old Slater matrix plus a perturbation matrix,

$$d_{jk}(\mathbf{x}^{\mathbf{new}}) = d_{jk}(\mathbf{x}^{\mathbf{old}}) + \Delta_{jk}, \tag{19}$$

where

$$\Delta_{jk} = \delta_{ik}[\phi_j(\mathbf{x}_\mathbf{i}^{\mathbf{new}}) - \phi_j(\mathbf{x}_\mathbf{i}^{\mathbf{old}})] = \delta_{ik}(\Delta\phi)_j. \tag{20}$$

Proof for updating algorithm for Slater determinant

Computing the inverse of the transposed matrix we arrive at

$$d_{kj}(\mathbf{x}^{\mathbf{new}})^{-1} = [d_{kj}(\mathbf{x}^{\mathbf{old}}) + \Delta_{kj}]^{-1}. \tag{21}$$

Proof for updating algorithm for Slater determinant

The evaluation of the right hand side (rhs) term above is carried out by applying the identity $(A+B)^{-1} = A^{-1} - (A+B)^{-1}BA^{-1}$. In compact notation it yields

$$\begin{aligned}
[\mathbf{D}^T(\mathbf{x^{new}})]^{-1} &= [\mathbf{D}^T(\mathbf{x^{old}}) + \Delta^T]^{-1} \\
&= [\mathbf{D}^T(\mathbf{x^{old}})]^{-1} - [\mathbf{D}^T(\mathbf{x^{old}}) + \Delta^T]^{-1}\Delta^T[\mathbf{D}^T(\mathbf{x^{old}})]^{-1} \\
&= [\mathbf{D}^T(\mathbf{x^{old}})]^{-1} - \underbrace{[\mathbf{D}^T(\mathbf{x^{new}})]^{-1}}_{\text{By Eq.21}}\Delta^T[\mathbf{D}^T(\mathbf{x^{old}})]^{-1}.
\end{aligned}$$

Proof for updating algorithm for Slater determinant

Using index notation, the last result may be expanded by

$$\begin{aligned}
d_{kj}^{-1}(\mathbf{x^{new}}) &= d_{kj}^{-1}(\mathbf{x^{old}}) - \sum_l \sum_m d_{km}^{-1}(\mathbf{x^{new}})\Delta_{ml}^T d_{lj}^{-1}(\mathbf{x^{old}}) \\
&= d_{kj}^{-1}(\mathbf{x^{old}}) - \sum_l \sum_m d_{km}^{-1}(\mathbf{x^{new}})\Delta_{lm} d_{lj}^{-1}(\mathbf{x^{cur}}) \\
&= d_{kj}^{-1}(\mathbf{x^{old}}) - \sum_l \sum_m d_{km}^{-1}(\mathbf{x^{new}})\delta_{im}(\Delta\phi)_l d_{lj}^{-1}(\mathbf{x^{old}}) \\
&= d_{kj}^{-1}(\mathbf{x^{old}}) - d_{ki}^{-1}(\mathbf{x^{new}})\sum_{l=1}^{N}(\Delta\phi)_l d_{lj}^{-1}(\mathbf{x^{old}}) \\
&= d_{kj}^{-1}(\mathbf{x^{old}}) - d_{ki}^{-1}(\mathbf{x^{new}})\sum_{l=1}^{N}[\phi_l(\mathbf{r_i^{new}}) - \phi_l(\mathbf{r_i^{old}})]D_{lj}^{-1}(\mathbf{x^{old}}).
\end{aligned}$$

Proof for updating algorithm for Slater determinant

Using

$$\mathbf{D}^{-1}(\mathbf{x^{old}}) = \frac{adj\mathbf{D}}{|\mathbf{D}(\mathbf{x^{old}})|} \quad \text{and} \quad \mathbf{D}^{-1}(\mathbf{x^{new}}) = \frac{adj\mathbf{D}}{|\mathbf{D}(\mathbf{x^{new}})|},$$

and dividing these two equations we get

$$\frac{\mathbf{D}^{-1}(\mathbf{x^{old}})}{\mathbf{D}^{-1}(\mathbf{x^{new}})} = \frac{|\mathbf{D}(\mathbf{x^{new}})|}{|\mathbf{D}(\mathbf{x^{old}})|} = R \Rightarrow d_{ki}^{-1}(\mathbf{x^{new}}) = \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R}.$$

Proof for updating algorithm for Slater determinant

We have

$$d_{kj}^{-1}(\mathbf{x^{new}}) = d_{kj}^{-1}(\mathbf{x^{old}}) - \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R}\sum_{l=1}^{N}[\phi_l(\mathbf{r_i^{new}}) - \phi_l(\mathbf{r_i^{old}})]d_{lj}^{-1}(\mathbf{x^{old}}),$$

or

$$
\begin{aligned}
d_{kj}^{-1}(\mathbf{x^{new}}) = d_{kj}^{-1}(\mathbf{x^{old}}) \quad &- \quad \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R} \sum_{l=1}^{N} \phi_l(\mathbf{r_i^{new}}) d_{lj}^{-1}(\mathbf{x^{old}}) \\
&+ \quad \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R} \sum_{l=1}^{N} \phi_l(\mathbf{r_i^{old}}) d_{lj}^{-1}(\mathbf{x^{old}}) \\
= d_{kj}^{-1}(\mathbf{x^{old}}) \quad &- \quad \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R} \sum_{l=1}^{N} d_{il}(\mathbf{x^{new}}) d_{lj}^{-1}(\mathbf{x^{old}}) \\
&+ \quad \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R} \sum_{l=1}^{N} d_{il}(\mathbf{x^{old}}) d_{lj}^{-1}(\mathbf{x^{old}}).
\end{aligned}
$$

Proof for updating algorithm for Slater determinant

In this equation, the first line becomes zero for $j = i$ and the second for $j \neq i$. Therefore, the update of the inverse for the new Slater matrix is given by

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