

# Slides from FYS-KJM4411/9411

## Variational Monte Carlo methods

Morten Hjorth-Jensen<sup>1,2</sup>

<sup>1</sup>Department of Physics, University of Oslo, Oslo, Norway

<sup>2</sup>Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University, East Lansing, MI 48824, USA

Spring 2015

### Quantum Monte Carlo Motivation

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

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

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.,

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

and assuming the set of eigenfunctions to be normalized one obtains

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

where we used that  $H(\mathbf{R}) \Psi_n(\mathbf{R}) = E_n \Psi_n(\mathbf{R})$ . 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 homogeneously 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 Schrodinger 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  $\psi_T(\mathbf{R}, \alpha)$ , for a many-body system consisting of  $N$  particles located at positions

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

- Then we evaluate the expectation value of the hamiltonian  $H$

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

- Thereafter we vary  $\alpha$  according to some minimization algorithm and return to the first step.

## Quantum Monte Carlo Motivation

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

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

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

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

## Quantum Monte Carlo Motivation

Define a new quantity

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

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

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

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  $\mathbf{R}$  and variational parameters  $\alpha$  and calculate  $|\psi_T^\alpha(\mathbf{R})|^2$ .
- Initialise the energy and the variance and start the Monte Carlo calculation.
  - Calculate a trial position  $\mathbf{R}_p = \mathbf{R} + r * \text{step}$  where  $r$  is a random variable  $r \in [0, 1]$ .
  - Metropolis algorithm to accept or reject this move  $w = P(\mathbf{R}_p)/P(\mathbf{R})$ .
  - If the step is accepted, then we set  $\mathbf{R} = \mathbf{R}_p$ .
  - 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  $\alpha$  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

$\alpha$	$\langle H \rangle$	$\sigma^2$	$\sigma/\sqrt{N}$
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 hamiltonian 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

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

**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: 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  $r_{12} = |\mathbf{r}_1 - \mathbf{r}_2|$ .

## 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

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

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  $r_1 \rightarrow 0$ .

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

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

For small values of  $r_1$ , the terms which dominate are

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

since the second derivative does not diverge due to the finiteness of  $\Psi$  at the origin.

## Quantum Monte Carlo: the helium atom

This results in

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

and

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

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

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

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

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

The last equation can be generalized to

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

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

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

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  $\Psi_C$  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

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

with  $\alpha$  and  $\beta$  as variational parameters.

The local energy is for this case

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

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 URL: '<http://docs.sympy.org/latest/index.html>'. 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

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

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).s
# Factorize also the r**2 terms
(diff(diff(phi, x), x) + diff(diff(phi, y), y) + diff(diff(phi, z), z)).factor().collect(Z).
print printing.ccode((diff(diff(phi, x), x) + diff(diff(phi, y), y) + diff(diff(phi, z), z))
```

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) / (h*h);
        }
    }
    return kineticEnergy + 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, all in one file.

## 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;

```



```

    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 first attempt at solving the Helium atom

Exercises for first lab session.

- If you have never used git, Qt, armadillo etc, get familiar with them

- Study the simple program in the folder URL: '<https://github.com/CompPhysics/ComputationalPhysics/pages/doc/pub/vmc/programs/>'
- Implement the closed form expression for the local energy and the so-called quantum force
- Convince yourself that the closed form expressions are correct. Check both wave functions
- Implement the closed form expression for the local energy and compare with a code where the second derivatives are computed numerically.