VMC part 2

Stian Roti Svendby

13. April 2015

Abstract:

In this project we have implemented a general function to compute the Slater determinant for different atoms, with focus on computational efficiency, and implemented a more efficient way of including the Jastrow factor. We have also parallelized the code to gain a speedup in the computations. We have then used our new program to compute the ground state and the onebody density for both beryllium and neon with Monte Carlo simulations.

Introduction:

In part 1 of this project, we estimated the ground state energy for helium and beryllium, using our Monte Carlo solver with predefined wavefunctions. In such small atoms, it was higly doable to just write out and implement the wavefunctions as we did, and they were not very CPU-expensive to compute. When moving to bigger atoms, like neon, we have to compute a ten by ten Slater determinant to obtain the wavefunction, which require a lot of CPU-time if we use a brute force approach. We can not tolerate using the brute force method computing the whole Slater determinant each time we move an electron, so we have to do some tricks making us able to update the Slater determinant in an efficient way. How this is done will be explained in method section. The Jastrow factor will also be implementet in a similar way as the Slater determinant for efficient implementation. For further speedup, we will also parallelize the code. When this is done we, will again compute the ground state and the onebody density for beryllium as a benchmark to verify that the implementation work as expected, and also intestigate the speedup of our code due to parallization. We will also do the same calculations for neon.

Methods:

In our calculations of the local energy (kinetic part), we have to calculate the Laplacian of the wavefunction every time we move an electron. If quantum force is applied, we also have to compute the gradient of the wavefunction. This means that we have to differentiate the Slater determinant and the Jastrow factor with respect to all spatial coordinates and particles for every move of an electron, which usually will be the most time-consuming part of our code. To compute the determinant of a $N \times N$ matrix we would brute force use Gaussian elimination of order $O(N^3)$. In our case we have $N = number\,of\,particles \times d = dimensions$ independent coordinates, which means that we have to evaluate the entire Slater determinant $N \times d$ times to calulate the local energy and $N \times d$ times to calulate the quantum force. This leads to an order of $O(d \times N^4)$, that is very much computation for large atoms. What we would like to do, is to find an effisient way of computing the gradient and Laplacian of the Slater determinant, as well as the ratio used in metropolis algorithm, to make the code run faster. As mentioned in the introduction, we will now explain how the Slater determinant can be implemented in a much more efficient way.

We write our Slater determinant matrix as D. A matrix element is defined as

$$d_{ij} = \phi_j(\boldsymbol{r}_i)$$

, with $\phi_j(\boldsymbol{r}_i)$ as a single particle wavefunction. The important thing we have to realize, is that when differentiating the Slater determinant with respect to a given electron (moving only one electron at the time), only one of the the rows in the corresponding Slater determinant is changed, making it unnecessary to recompute the whole determinant. The solutions turns out to be using the inverse of the Slater matrix with elements defined as

$$d^{-1}_{i,j} = \frac{C_{ji}}{|D|}$$

with C_{ji} as the cofactor matrix. This matrix will vanish in our calculations, and will not cause any troubles.

The orbitals we use in the Slater determinant are hydrogenic, and are given by

$$\phi_{1s}(\boldsymbol{r}_i) = e^{-\alpha r_i}$$

$$\phi_{2s}(\boldsymbol{r}_i) = (1 - \alpha r_i/2)e^{-\alpha r_i/2}$$

$$\phi_{2p}(\boldsymbol{r}_i) = \alpha \boldsymbol{r}_i e^{-\alpha r_i/2}$$

, where $\phi_{2p}(\boldsymbol{r}_i)$ can be split up in

$$\phi_{2px}(\boldsymbol{r}_i) = \alpha x_i e^{-\alpha r_i/2}$$

$$\phi_{2py}(\boldsymbol{r}_i) = \alpha y_i e^{-\alpha r_i/2}$$

$$\phi_{2pz}(\boldsymbol{r}_i) = \alpha z_i e^{-\alpha r_i/2}$$

Here we have actually used a mapping from spherical coordinates to cartesian coordinated using *real solid harmonics*, with

$$\psi(r, \theta, \phi) = R(r)Y(\theta, \phi)$$

, where we recognize $Y(\theta, \phi)$ as x, y and z after the mapping with solid harmonics found in tables (lecture notes FYS4411 "vmc" page 60). This will work (not alter the energies) as long an external magnetic field not is applied to the system. So we have five different orbitals, that are enought for doing simulations on the neon atom. For heavier atoms, we have to add more orbitals.

First we want an efficient expression to evaluate the ratio in the metropolis algorithm

$$R_{SD} = \frac{|D(\boldsymbol{r}^{new})|}{|D(\boldsymbol{r}^{old})|} = \frac{\sum_{j=1}^{N} d_{ij}(\boldsymbol{r}^{new})C_{ij}(\boldsymbol{r}^{new})}{\sum_{j=1}^{N} d_{ij}(\boldsymbol{r}^{old})C_{ij}(\boldsymbol{r}^{old})}$$

We have that $C_{ij}(\boldsymbol{r}^{new}) = C_{ij}(\boldsymbol{r}^{old})$, $D_{i,j}D_{ji}^{-1} = \delta_{ij} = 1$, and using the expression we have for $d_{i,j}^{-1}$, we end up with

$$R_{SD} = \frac{\sum_{j=1}^{N} d_{ij}(\boldsymbol{r}^{new})d_{ji}^{-1}(\boldsymbol{r}^{old})}{\sum_{j=1}^{N} d_{ij}(\boldsymbol{r}^{old})d_{ji}^{-1}(\boldsymbol{r}^{old})} = \sum_{j=1}^{N} d_{ij}(\boldsymbol{r}^{new})d_{ji}^{-1}(\boldsymbol{r}^{old})$$

or

$$\boxed{R_{SD} = \sum_{j=1}^{N} \phi_j(\boldsymbol{r}_i^{new})d_{ji}^{-1}(\boldsymbol{r}^{old})}$$

This is the ratio we want to use in Metropolis algorithm with order $O(N)$.

Implementation is done with the following code:

```
// compute the R_sd ratio
void VMCSolver::compute_R_sd(int i){
    R_sd = 0.0;
    if(i < nParticles/2){
        for(int j=0; j<nParticles/2; j++){
            R_sd += SlaterPsi(rNew, i, j)*D_up_old(j, i);
        }
    }
    else{
        for(int j=0; j<nParticles/2; j++){
            R_sd += SlaterPsi(rNew, i, j)*D_down_old(j, i-nParticles/2);
        }
    }
}
```

Then we want efficient expressions for

$$\frac{\nabla\Psi}{\Psi}$$

$$\frac{\nabla^2\Psi}{\Psi}$$

used in the calculation of quantum force and local energy. We have when moving only one electron at a time that

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

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

Computer implementation of these expressions is given by:

```
// compute slater first derivative
void VMCSolver::SlaterGradient(int i){
    compute_R_sd(i);
    if(i < nParticles/2){
        for(int k=0; k<nDimensions; k++){
            double derivative_up = 0.0;
            for(int j=0; j<nParticles/2; j++){
                derivative_up += Psi_first_derivative(i, j, k)*D_up_old(j, i);
            }
            SlaterGradientNew(i, k) = (1.0/R_sd)*derivative_up;
        }
    }
    else{
        for(int k=0; k<nDimensions; k++){
            double derivative_down = 0.0;
                for(int j=0; j<nParticles/2; j++){
                    derivative_down += Psi_first_derivative(i, j, k)*D_down_old(j, i-
                        nParticles/2);
                }
                SlaterGradientNew(i, k) = (1.0/R_sd)*derivative_down;
            }
        }
    }

// compute slater second derivative
double VMCSolver::SlaterLaplacian(){
    double derivative_up = 0.0;
    double derivative_down = 0.0;
    for(int i=0; i<nParticles/2; i++){
        for(int j=0; j<nParticles/2; j++){
            derivative_up += Psi_second_derivative(i, j)*D_up_new(j, i);
                        derivative_down += Psi_second_derivative(i+nParticles/2, j)*
                D_down_new(j, i);
        }
    }
    double derivative_sum = derivative_up + derivative_down;
    return derivative_sum;
}
```

The update of the inverse matrix are done by

$$d_{kj}^{-1}(\boldsymbol{r}^{new}) = \begin{cases} d_{kj}^{-1}(\boldsymbol{r}^{old}) - \frac{d_{ki}^{-1}(\boldsymbol{r}^{old})}{R_{SD}} \sum_{l=1}^{N} d_{il}(\boldsymbol{r}^{new}) d_{lj}^{-1}(\boldsymbol{r}^{old}) & if\ j \neq i \\ \frac{d_{ki}^{-1}(\boldsymbol{r}^{old})}{R_{SD}} \sum_{l=1}^{N} d_{il}(\boldsymbol{r}^{old}) d_{lj}^{-1}(\boldsymbol{r}^{old}) & if\ j = i \end{cases}$$

With the property $D_{i,j} D_{ji}^{-1} = 1$, this can be written:

$$d_{kj}^{-1}(\boldsymbol{r}^{new}) = \begin{cases} d_{kj}^{-1}(\boldsymbol{r}^{old}) - \frac{d_{ki}^{-1}(\boldsymbol{r}^{old})}{R_{SD}} \sum_{l=1}^{N} d_{il}(\boldsymbol{r}^{new}) d_{lj}^{-1}(\boldsymbol{r}^{old}) & if\ j \neq i \\ \frac{d_{ki}^{-1}(\boldsymbol{r}^{old})}{R_{SD}} & if\ j = i \end{cases}$$

The updating algorithm is implemented in our code by:

```
// update Slater matrix
void VMCSolver::update_D(mat& D_new, const mat& D_old, int i, int selector){
    i = i - nParticles/2*selector;
    for(int k=0; k<nParticles/2; k++){
        for(int j=0; j<nParticles/2; j++){
            if(j!=i){
                double sum = 0;
                for(int l=0; l<nParticles/2; l++){
                    sum += SlaterPsi(rNew, i+nParticles/2*selector, l)*D_old(l,j);
                }
                D_new(k,j) = D_old(k,j) - D_old(k, i)*sum/R_sd;
            }
            else{
                D_new(k,j) = D_old(k, i)/R_sd;
            }
        }
    }
}
```

With the updating algorithm we only need to invert the Slater matrix once, using LU decomposition ($O(N^3)$), or just using the $inv()$ function in Armadillo. A single derivative is of order $O(N)$ and we have to take $d \times N$ derivatives, which leads to an order of $O(d \times N^2)$. We also have $O(N^2)$ for updating the inverse matrix. This means that we get a far better scaling than the brute force solver of order $O(d \times N^4)$.

We also want to split up the Slater determinant and write it as a product of a spin up part and a spin down part. In our case we give the first half of electrons spin up, and the second half spinn down. When moving one electron at a time, we only need to update one of the matrizes at each move. The Slater determinant can then be written as

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

This is correct as long as the Hamiltonian is independent of spin. This ansatz is not antisymmetric under the exchange of electrons with opposite spinn, but gives the same expectation value as the full Slater determinant.

For the correlation part, or the Jastrow factor we have

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

as in part 1 on this project.

We need to find the ratio used in metropolis algorith, R_C. This is given by:

$$R_C = \frac{\Psi_C^{new}}{\Psi_C^{new}} = \frac{e^{U_{new}}}{e^{U_{old}}} = e^{\Delta U}$$

, with

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

, with

$$f_{ki} = \frac{a_{ki}r_{ki}}{1+\beta r_{ki}}$$

R_C can be implemented by:

```
// compute C matrix
void VMCSolver::fillJastrowMatrix(mat &CorrelationMatrix){
    for(int k=0; k < nParticles; k++){
        for(int i=k+1; i < nParticles; i++){
            CorrelationMatrix(k,i) = a_matrix(k,i)*r_distance(k,i)/(1.0 + beta*
                r_distance(k,i));
        }
    }
}

// compute R_c ratio
```

```
void VMCSolver::compute_R_c(){
    double deltaU = 0.0;
    for(int k=0; k < nParticles; k++){
        for(int i=0; i < k; i++){
            deltaU += C_new(i, k) - C_old(i, k);
        }
        for(int i=k+1; i < nParticles; i++){
            deltaU += C_new(k, i) - C_old(k, i);
        }
    }
    R_c = exp(deltaU);
}
```

When R_{SD} and R_C is computed after a move, we have $R = R_{SD}R_C$, and the rule for acceptance of a move is given by the expression:

$$if(rand\,num <= GreensFunction * R^2)$$

When the correlation functions only depends on the relative distance between particles, we can use linear $Padé - Jastrow$ form to compute the gradient needed for the quantum force, and also local energy. The gradient becomes:

$$\frac{1}{\Psi_{PJ}}\frac{\partial \Psi_{PJ}}{\partial x_k} = \sum_{i=1}^{k-1}\frac{\boldsymbol{r}_{ik}}{r_{ik}}\frac{\partial f_{ik}}{\partial r_{ik}} - \sum_{i=k+1}^{N}\frac{\boldsymbol{r}_{ki}}{r_{ki}}\frac{\partial f_{ki}}{\partial r_{ki}}$$

The Laplacian needed for the local energy is given by

$$\frac{\nabla_k^2\Psi_{PJ}}{\Psi_{PJ}} = \left(\frac{\nabla_k\Psi_{PJ}}{\Psi_{PJ}}\right)^2 + \sum_{i=1}^{k-1}\left[\left(\frac{d-1}{r_{ik}}\right)\frac{\partial f_{ik}}{\partial r_{ik}} + \frac{\partial^2 f_{ik}}{\partial^2 r_{ik}}\right] - \sum_{i=k+1}^{N}\left[\left(\frac{d-1}{r_{ki}}\right)\frac{\partial f_{ki}}{\partial r_{ki}} + \frac{\partial^2 f_{ki}}{\partial^2 r_{ki}}\right]$$

, with

$$\frac{\partial f_{ik}}{\partial r_{ik}} = \frac{a_{ik}}{(1+\beta r_{ik})^2}$$

$$\frac{\partial^2 f_{ik}}{\partial^2 r_{ik}} = -\frac{2a_{ik}\beta}{(1+\beta r_{ik})^3}$$

, where we have to sum over all particles when computing the local energy. The first term (we call it $GradientSquared$) can be computed by summing up squares of $\frac{1}{\Psi_{PJ}}\frac{\partial \Psi_{PJ}}{\partial x_k}$ and can effectively be computed in the same function as quantum force:

```
// compute the quantum force used in importance sampling
void VMCSolver::QuantumForce(const mat &r, mat &F){
    GradientSquared = 0.0;
    for(int k=0; k<nParticles; k++){
        for(int j=0; j<nDimensions; j++){
            if(activate_JastrowFactor){
                double GradientSum = 0.0;
                for(int i=0; i<k; i++){
                    GradientSum += (r(k,j)-r(i,j))/r_distance(i,k)*JastrowDerivative(i,k
                        );
                }
                for(int i=k+1; i<nParticles; i++){
                    GradientSum -= (r(i,j)-r(k,j))/r_distance(k,i)*JastrowDerivative(k,i
                        );
                }
                F(k,j) =  2.0*(SlaterGradientNew(k,j) + GradientSum);
                GradientSquared += GradientSum*GradientSum;
                JastrowGradient(k, j) = GradientSum;
            }
            else{
                F(k,j) =  2.0*SlaterGradientNew(k,j);
            }
```

```
        }
    }
}
```

In the code above, we notice the calculation of *JastrowGradient*. This gradient has to be multiplied by the *SlaterGradient* as an energy crossterm. All calculations for the energy related to the including of Jastrow factor can be implemented with the folling code:

```
// compute derivatives used in gradient
void VMCSolver::computeJastrowDerivative(int k){
    for(int i=0; i<k; i++){
        double divisor = 1.0 + beta*r_distance(i, k);
        JastrowDerivative(i, k) = a_matrix(i, k)/(divisor*divisor);
    }
    for(int i=k+1; i<nParticles; i++){
        double divisor = 1.0 + beta*r_distance(k, i);
        JastrowDerivative(k, i) = a_matrix(k, i)/(divisor*divisor);
    }
}

// compute double derivatives used in Jastrow laplacian
void VMCSolver::computeJastrowLaplacian(int k){
    for(int i=0; i<k; i++){
        double divisor = 1.0 + beta*r_distance(i, k);
        JastrowLaplacianNew(i, k) = -2.0*a_matrix(i, k)*beta/(divisor*divisor*divisor);
    }
    for(int i=k+1; i<nParticles; i++){
        double divisor = 1.0 + beta*r_distance(k, i);
        JastrowLaplacianNew(k, i) = -2.0*a_matrix(k, i)*beta/(divisor*divisor*divisor);
    }
}

// compute Jasstrow energy and energy crossterm
double VMCSolver::computeJastrowEnergy(){
    double sum = 0.0;
    energytermSlaterJastrow = 0.0;
    sum += GradientSquared;
    for(int k=0; k<nParticles; k++){
        for(int i=0; i<k; i++) {
            sum += (nDimensions - 1)/r_distance(i, k)*JastrowDerivative(i, k) +
                    JastrowLaplacianNew(i, k);
        }
        for(int i=k+1; i<nParticles; i++) {
            sum += (nDimensions - 1)/r_distance(k, i)*JastrowDerivative(k, i) +
                JastrowLaplacianNew(k, i);
        }
    }
    for(int k=0; k<nDimensions; k++){
        energytermSlaterJastrow += dot(SlaterGradientNew.col(k), JastrowGradient.col(k))
            ;
    }
    return sum;
}
```

The total kinetic energy with Jastrow activated is now given by:

$$-\frac{1}{2}\frac{\nabla^2\Psi}{\Psi} = -\frac{1}{2}\left(\frac{\nabla^2|D|_\uparrow}{|D|_\uparrow} + \frac{\nabla^2|D|_\downarrow}{|D|_\downarrow} + \frac{\nabla^2\Psi_{PJ}}{\Psi_{PJ}} + 2\left[\frac{\nabla|D|_\uparrow}{|D|_\uparrow} + \frac{\nabla|D|_\downarrow}{|D|_\downarrow}\right]\cdot\frac{\nabla\Psi_{PJ}}{\Psi_{PJ}}\right)$$

Then the total energy is given by this energy, pluss the potential energy.

To parallelize the code we use MPI. This is simply done by running multiple MC simulations simultaneously, collect the estimated energies, and divide by the number of processes:

```
int my_rank, world_size;
```

```
double energy, sum_energy;

// Initialize the parallel environment
MPI_Init (&nargs, &args);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
MPI_Comm_size (MPI_COMM_WORLD, &world_size);

//VMCSolver *solver = new VMCSolver();
//energy = solver->runMonteCarloIntegration(nCycles, my_rank, world_size);

// Collect energy estimates
MPI_Reduce(&energy, &sum_energy, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

// Final energy estimate with multiple processors
if(my_rank==0){
    cout << "Totalt cycles: " << nCycles << endl;
    cout << endl << "Ground state estimate: " << sum_energy/world_size << endl;
}

// Clean up and close the MPI environment
MPI_Finalize();
```

Results:

With our new optimized solver, using α and β obtained from project 1 and including importance sampling and Jastrow factor, we compute again the ground state energy for Beryllium. With $nCycles = 10^7$, we find $E_{min} = -14.4967$ with mean $r_{ij} = 2.1209$. Using blocking, we obtain:

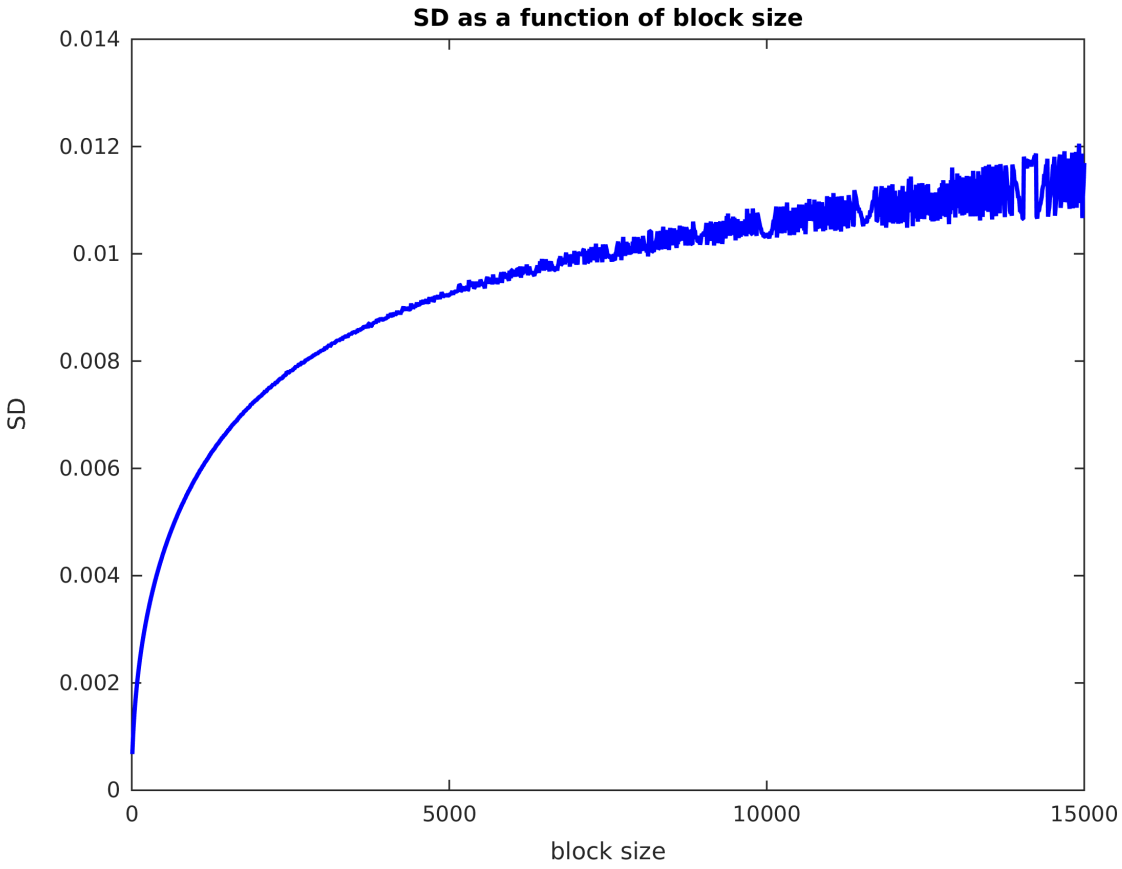

Figure 1: Blocking performed on the Beryllium atom with $N = 10^7$. We see a "plateu" where $\sigma_{block} \approx 0.011$ and $N_{blocksize} \approx 10000$.

Using value $\sigma_{block} \approx 0.011$ and $N_{blocksize} \approx 10000$, we find the standard deviation:

$$\sigma = \frac{\sigma_{block}}{\sqrt{N/N_{blocksize}}} = \frac{0.011}{\sqrt{10^7/10000}} = 3.48 \times 10^{-4}$$

Doing the same simulation with importance samling and Jastrowfactor on neon, with $\alpha = 10.2$ and $\beta = 0.09$, we find (with $nCycles = 10^7$), $E_{min} = -127.985$ with mean $r_{ij} = 1.16195$. Using blocking, we obtain:

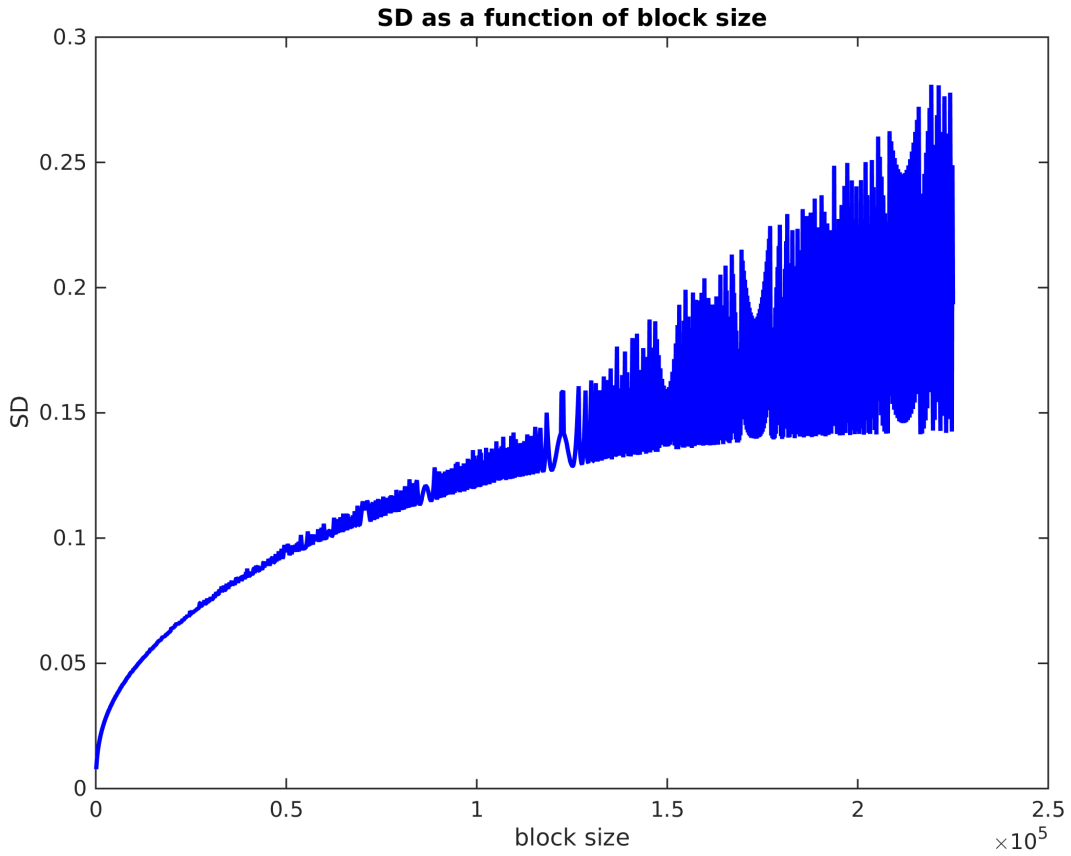

Figure 2: Blocking performed on the Neon atom with $N = 10^7$. We see a "plateu" where $\sigma_{block} \approx 0.14$ and $N_{blocksize} \approx 200000$.

Using value $\sigma_{block} \approx 0.14$ and $N_{blocksize} \approx 200000$, we find the standard deviation:

$$\sigma = \frac{\sigma_{block}}{\sqrt{N/N_{blocksize}}} = \frac{0.14}{\sqrt{10^7/200000}} = 1.98 \times 10^{-2}$$

If we now go over to the parallelization of the code, we can investigate the speedup using more processors. The computer used in this project have a quad Core (4 CPUs) with support of hyperthreading (additional 4 virtual CPUs). If we try with 1, 2, 4 and 8 processors, we can se how long time the program use to run a given number of MC cycles, and compare the time to the time spend when using only one processor. If we divide the time with one CPU by the time used with more CPUs, we get how many times faster the program run. The results are given below:

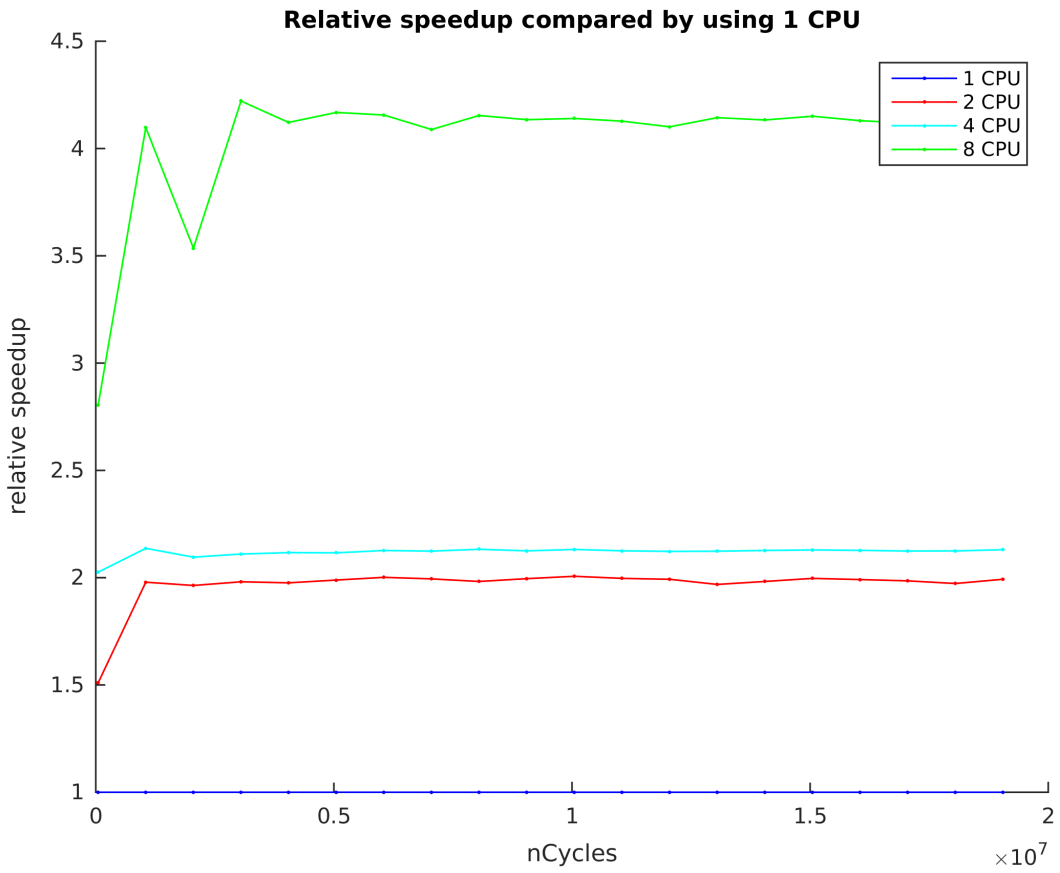

Figure 3: Relative speedup between equal simulations with different number of processors.

Theoretically, the optimal speedup scales with number of cores used in the simualation. But there are many factor that can slow down the speedup. In our case, we are lucky, because there are very little overhead due to communication between processors. There are still limitations due to memory speed, and when using 8 CPUs, we are actually using 4 CPUs with multithreading. This schould slow slow down our program a bit. In the figure below, the deviation in percentage from theoretically optimal speedup is shown:

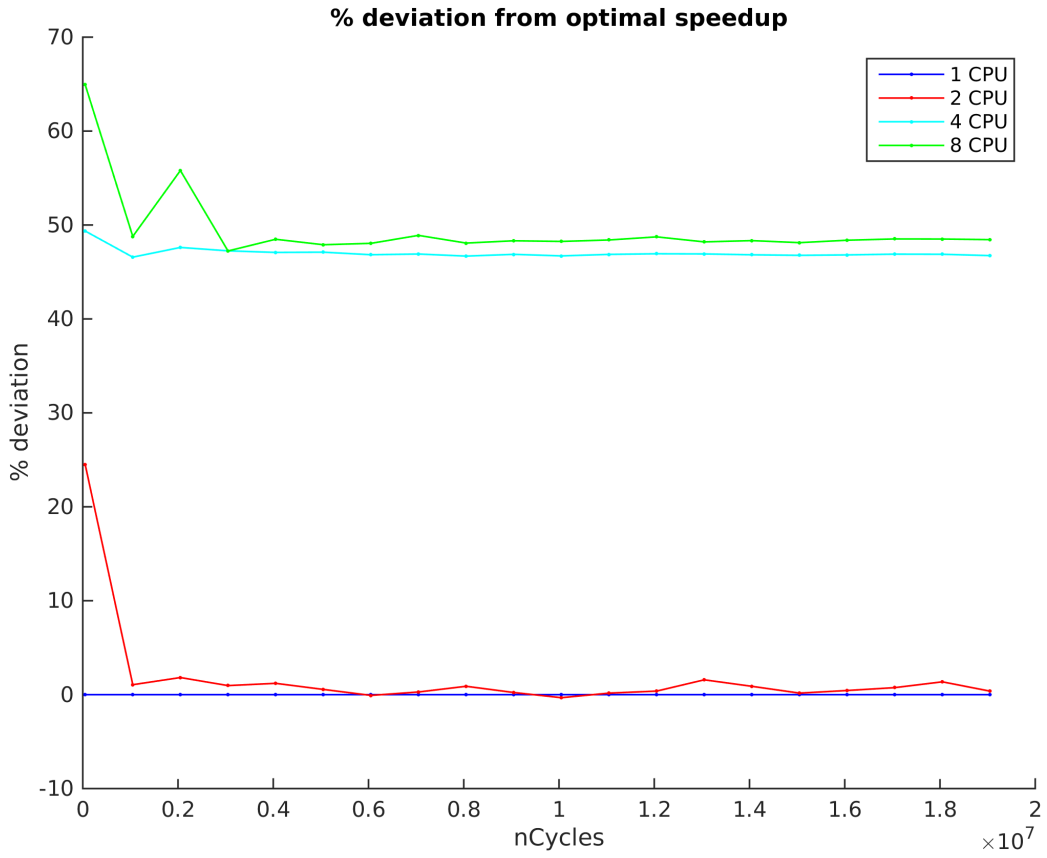

Figure 4: Deviation from theoretical optimal speedup using different number of processors.

We see an almost optimal speedup when using 2 CPUs. When using 4 CPUs, we actually get a very poor speedup compared to using 2 CPUs, and almost 50% from optimal theoretical speedup. This can be due tue how the operating system choose processors. It is possible that we are using 2 physical cores and 2 virtual cores with hyperthreading, with a limitation of shared memory. We have the same effect with 8 CPUs, with a speedup almost 50% from optimal theoretical speedup. Therefore, the hyperthreading seems to give poor improvment on the computational efficiency, but the overhead due to communication between prosessors have very little cost. We can only observe slowdown for very few MC cycles, as shown in the two plots above. MC simulations are very easy to parallelize with high performace gain.

Next we look at the onebody density for beryllium and neon:

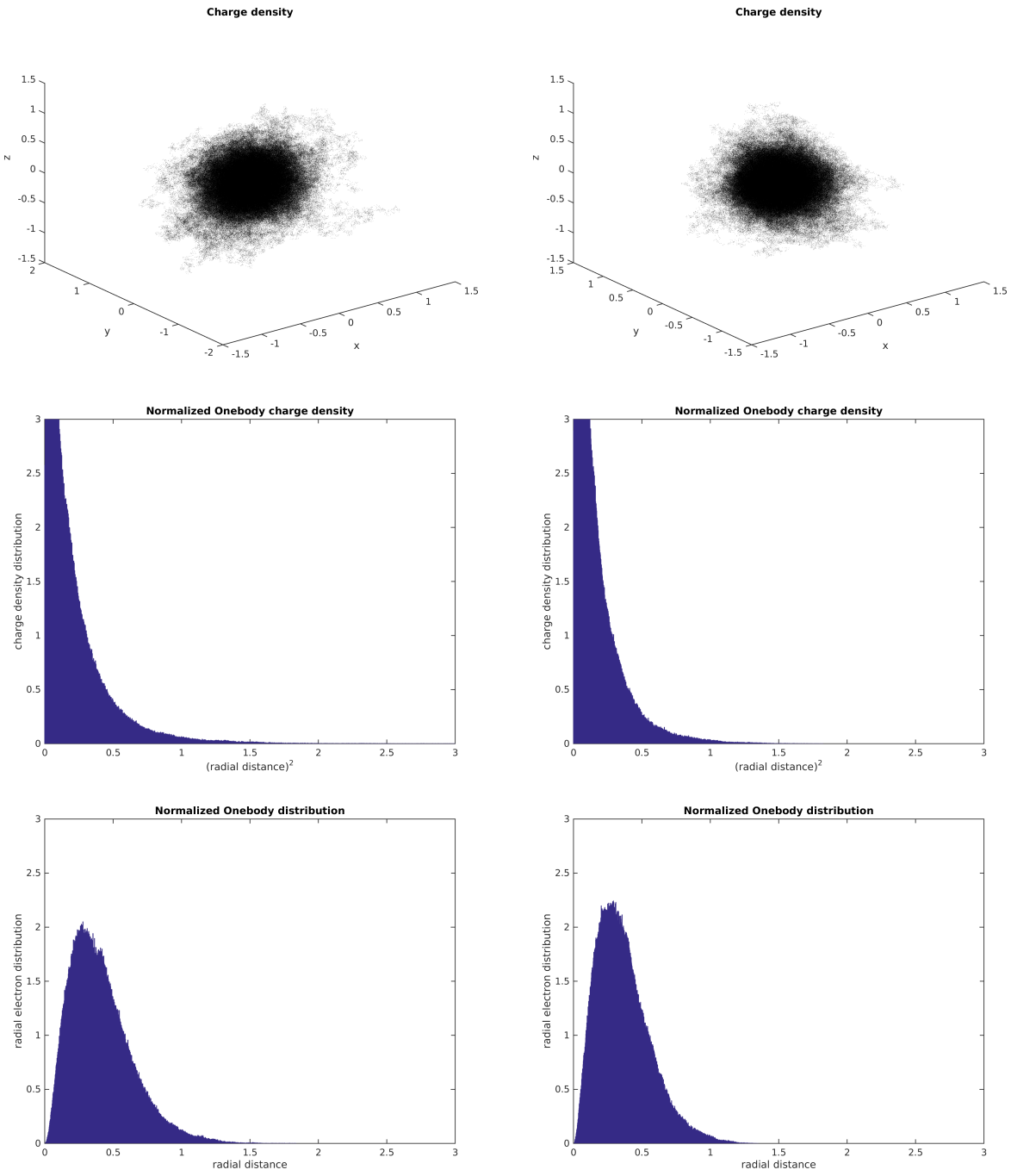

Figure 5: Onebody density beryllium. With Jastrow factor in the left coloumn, pure hydrogenic wavefunction in the right coloumn.

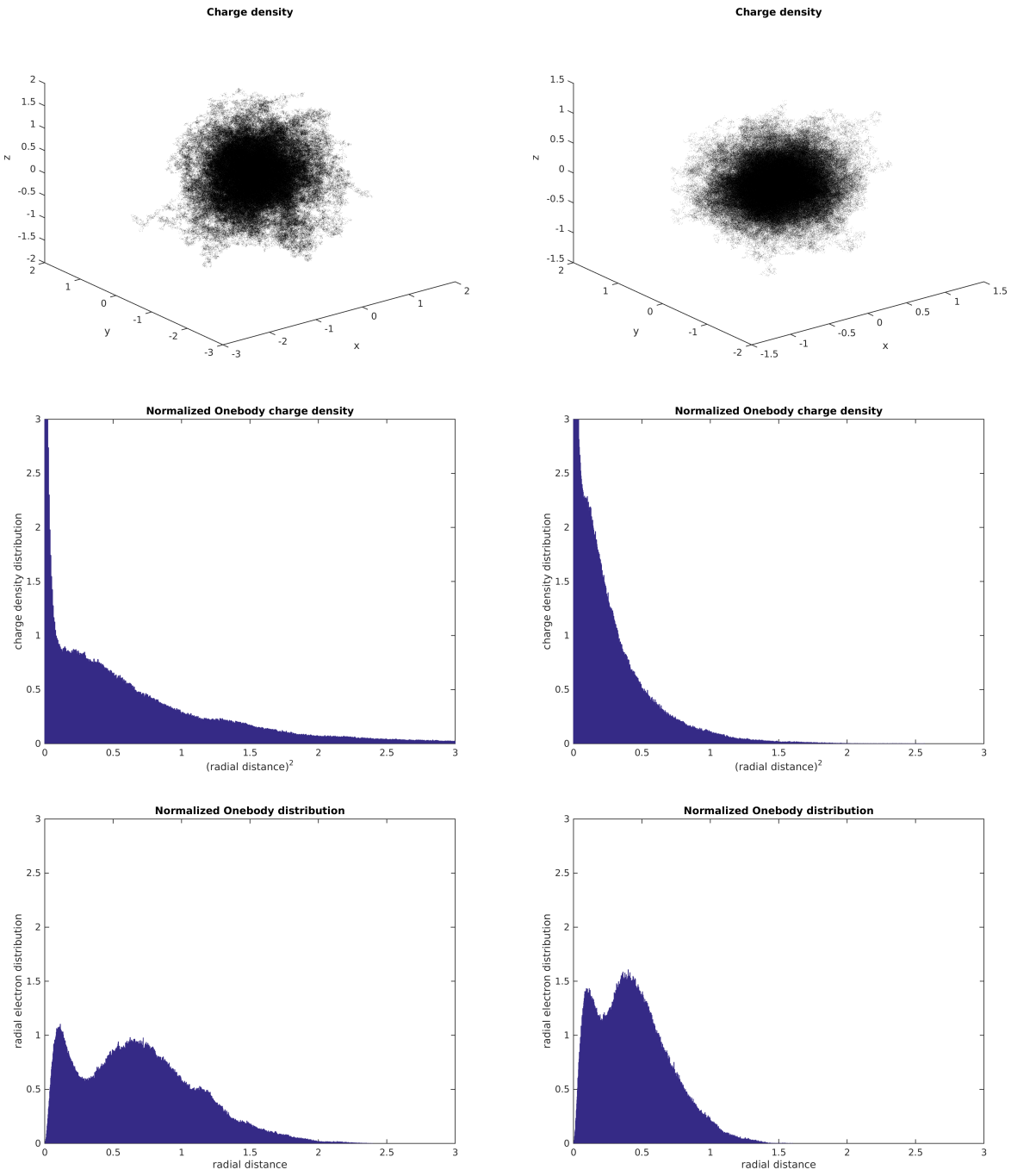

Figure 6: Onebody density neon. With Jastrow factor in the left coloumn, pure hydrogenic wavefunction in the right coloumn.

In *Figure* 5 we have plotted results for beryllium including Jastrow factor to the left and pure hydrogenic wavefunctions to the right. In the first row we have just plotted the position to one electron at all times. In the second row we have plotted a histogram of radial distance squared from the nucleous, since the density decrease with the square of radial distance. In the last row we have plotted a histogram of radial distance from the nucleous.

The first row give us an idea of what the electron desity looks like, but it is a lot interesting information we cannot see from this angle, and we hardly see any difference when including Jastrow factor. In the next figure we cannot either see any big difference, but we see how almost all the charge are distributed close to the nucleous. In the last row, we are able to spot some more spread of electrons when we include Jastrow factor. Since we have filled up two orbitals in beryllium, we could expect to see a distribution with two peaks, but the two orbitals are truly so close that we cannot distinguish them in the figures. The wider spread with Jastrow give some support of that theory.

If we now go over to neon in *Figure* 6 we get some more interesting plots. In the first row we see indications of more spread of charge when Jastrow is included. In the second row, we see a much wider spread in charge with Jastrow. But the last figure is the most interesting. Here we clearly see two peaks in the distribution in both figures, but also here with more spread with Jastrow. We are also able to see an indication of another peak to the right in the figure with Jastrow, which could be the third of three orbitals we would expect to observe for neon.

We see that the Jastrow factor is of great importance in our experiments. It doesn't make a huge difference in the plots, but the small difference makes us able to (truly) spot different orbitals in the atoms.

Conclusion:

In this project we have implemented a general Slater determinant, and included Jastrow factor, making us able investigate bigger atoms (such as Neon) in an efficient way compared to brute force numerical computations. We have also seen how parallelization speed up computations, but also seen that the given number of processors not nescessary correspond to the gain in speedup. In the end we saw how the Jastrow factor brings details to the plots of charge around the nucleous, making us able to spot different electron orbitals. Next we want to develop a better way of finding variational parameters, since the method used by now is very time consuming and unaccurate. We will also use some result from the Hartree-Fock theory to improve our solver.