Quantum Monte-Carlo Simulations of Atoms and Molecules

by

Roger Kjøde

THESIS
for the degree of
MASTER OF SCIENCE
Master in Computational Physics

Faculty of Mathematics and Natural Sciences
Department of Physics
University of Oslo

June 2016

Preface

I always found the sciences interesting, so the step of enrolling at the University in Oslo was an easy one. I believe I should know something about physics, but there is still a lot I don't know. (and I feel I should know.)

I started early with programming, when we got our first computer. An Amstrad cpc 464. 1 MHz 64 kb ram computer. and I have been programming ever since.

In this thesis we use c++ and python. The main Monte-carlo algorithm is written in c++, and python is used to generate c++ code. Symbolic Python (Sympy) is used extensively in this thesis in order to autogenerate code.

Thanks to my supervisor Professor Morten Hjorth-Jensen for all help with this thesis.

Contents

1 Introduction 9

I Theory 11

2 Physical theory 13
- 2.1 The Schrödinger equation 13
  - 2.1.1 Time-independent Schrödinger equation 14
  - 2.1.2 The Schrödinger equation in 3 dimensions 14
- 2.2 The potential 15
- 2.3 The Born-Oppenheimer approximation 16
- 2.4 Atomic units 16
- 2.5 Scaling 16
- 2.6 The variance. 17
- 2.7 Slater determinant 17
- 2.8 Jastrow factor 18
- 2.9 Full wave function 18
- 2.10 Hartree-Fock 18
- 2.11 Spin 19

3 Quantum Monte-Carlo 21
- 3.1 Monte Carlo integration 21
- 3.2 Importance sampling 21
- 3.3 The Metropolis Algorithm 22
- 3.4 Computing the energy 23
- 3.5 The variational principle 25
- 3.6 The Local energy 26

4 Optimizations 27
- 4.1 Optimizing the Slater determinant 27
- 4.2 Optimizing the Slater ratio 28

4.3 Optimizing the $\nabla_i|S|/|S|$ ratio. . . . 28
4.4 Optimizing the $\nabla_i^2|S|/|S|$ ratio. . . . 29
4.5 Updating the inverse Slater determinant. . . . 29
4.6 Optimizing the Jastrow ratio 29
4.7 Optimizing the $\frac{\nabla_i J}{J}$ Ratio. . . . 30
4.8 Optimizing the $\frac{\nabla_i^2 J}{J}$ Ratio 31

5 Orbitals 33
5.1 Hydrogen orbitals 33
5.2 Gaussian orbitals 35
5.3 The gradient and laplacian of the orbitals. . . . 39

6 Calculation of Molecules, Blocking and Optimized Variational Parameters 41
6.1 Homonuclear Diatomic Molecules 41
6.2 Random numbers 42
6.3 Blocking 43
6.4 Gradient descent 43
6.5 Parallelization 44
6.6 Errors 45

7 How to code VMC 49
7.1 The C++ files 49
7.2 The Python files 50
7.3 C++ code 50
  7.3.1 Main MVC algorithm 51
  7.3.2 Local energy code 53
  7.3.3 Quantum Force 57
  7.3.4 Update inverse Slater determinant. . . . 58
  7.3.5 Oprimized code for Slater ratio. . . . 60
  7.3.6 Optimized code for Jastrow factor. . . . 61
  7.3.7 Cusp condition algorithm. . . . 62

II Results 63

8 Results 65
8.1 The output 65
8.2 About the tables. . . . 66
8.3 Atoms 67
  8.3.1 Using Hydrogenic orbitals 67

- 8.3.2 Using Gaussian orbitals 69
- 8.4 Molecules 71
  - 8.4.1 Using Hydrogenic orbitals 71
  - 8.4.2 Using Gaussian orbitals 73
- 8.5 Time used in VMC. 75
- 8.6 Conclusion 78

9 Conclusions **79**
- 9.1 The point of the thesis 79
- 9.2 C++ and Python 79
- 9.3 Results 79
- 9.4 Effiniency of the code. 80
- 9.5 Further work 80

III Appendix **81**

A Programming languages **83**
- A.1 Computer programs in general 83
- A.2 C++ 83
  - A.2.1 Variables 84
  - A.2.2 Arrays 84
  - A.2.3 Includes 84
  - A.2.4 Namespace 85
  - A.2.5 Loops 85
  - A.2.6 If test 86
  - A.2.7 Case 86
  - A.2.8 Objects 87
  - A.2.9 Functions 88
  - A.2.10 Accessibility levels 88
  - A.2.11 Armadillo 89
  - A.2.12 MPI 89
  - A.2.13 Output 90
- A.3 Python 90
  - A.3.1 Variables 90
  - A.3.2 Loops 91
  - A.3.3 If test 91
  - A.3.4 Objects 91
  - A.3.5 Exec 92
  - A.3.6 Symbolic Python 92
  - A.3.7 C++ code 93

A.3.8 Printing to file. 93

Chapter 1

Introduction

The introduction of computers opens up a new world to the sciences. Problems which before were impossible to solve are now possible to solve.

The aim of this thesis is to develop an *Ab initio approach* to different atomic and diatomic systems. We will use Monte-carlo methods to avoid heavy integrals when solving the Schrödinger equation. The Schrödinger equation cannot be solved analytically, except for some very trivial systems. Our method of choice is Variational Monte-Carlo (VMC) which is used to find the ground state of a quantum system.

We use trial wave functions to build a Slater determinant with is used in the simulation. We use both hydrogenic orbitals and Gaussian orbitals. A correlation function (Jastrow factor) is used to improve the results. The Jastrow factor describes the particle-particle correlation of the system.

The goal is to find the binding energy of the electrons in many different atoms and simple molecules. The calculations have been performed at the local supercomputing facility Abel. With Abel many simulations can be run in parallel. With Monte-Carlo paralellization is very easy.

In this thesis we will look at atoms with even number of electrons. We look at systems like He, Be, Ne, Ar and Kr. We also look at simple diatomic molecules, from the light H_2 to the heavier O_2. In theory we can solve for heavier atoms and molecules.

The code for this thesis can be found at `https://github.com/rogerkj/master`. Here you will find all the c++ code and all Python scripts used to generate the final c++ code.

The prerequisite for reading this thesis is an understanding of basic calculus and statistical methods. The goal is to make it understandable for most people, but an explanation of basic math is not included. An understanding of quantum physics is not required but will make this thesis easier to understand.

In Appendix A we give a basic introduction to c++ and Python programming. In this thesis we use the linear algebra library Armadillo, which makes the code easy to read.

The structure

- In chapter 2 we will develop the physics and methods used in this thesis. The Schrödinger equation is used here.
- In chapter 3 we will then look at the Monte-Carlo method. We will develop the Metropolis algorithm and find an expression for the local energy. We will also introduce importance sampling.
- In chapter 4 we will look at optimizations of the algorithm. Here we use the inverse Slater determinant to optimize the algorithm. The Jastrow factor will also be optimalized.
- In chaper 5 we will look at the orbitals used in this thesis. We use either hydrogenic orbitals or we use Gaussian orbitals.
- In chapter 6 we will look at at various topics as homonuclear molecules, random numbers, blocking, gradient descent, parallelization and a source of errors.
- In chapter 7 we will look at some of the code used in this thesis. Not all code will be discussed but the most important parts are emphasized in the main text.
- In chapter 8 we will look at the results from the VMC program developed for this thesis.
- In chapter 9 we will conclude this thesis with a few words.
- In appendix A we give a short introduction to c++ and python programming.

Part I

Theory

Chapter 2

Physical theory

2.1 The Schrödinger equation

The basis for this thesis is the Schrödinger equation, which represents the law of motion for quantum systems.

Here we give a brief survey of the essentials behind the Schrödinger equation.
The time dependent Schrödinger equation (SE) for one particle moving in a potential V is: [1]

$$\boxed{i\hbar\frac{\partial\Psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\Psi}{\partial x^2} + \hat{V}\Psi,} \tag{2.1}$$

where Ψ represents the wave function. The wavefunction Ψ is a function of time, t and position, x. The product $\Psi * \Psi = |\Psi|^2$ is interpreted as the probability density of a given state. In one dimension the probability for a particle to be between two points a and b is given as

$$\int_a^b |\Psi(x,t)|^2 dx, x \in [a,b]. \tag{2.2}$$

The probability is normalized

$$\int_{-\infty}^{\infty} |\Psi(x,t)|^2 dx = 1. \tag{2.3}$$

We have also the corresponding quantum mechanical representations of the different classical operators: [7]

Quantity	Classical	QM operator
Position	$\mathbf{r}$	$\hat{\mathbf{r}} = \mathbf{r}$
momentum	$\mathbf{p}$	$\hat{\mathbf{p}} = -i\hbar\nabla$
Orbital momentum	$\mathbf{L} = \mathbf{r} \times \mathbf{p}$	$\hat{\mathbf{L}} = \mathbf{r} \times (-\hbar\nabla)$
Kinetic energy	$T = (\mathbf{p})^2/2m$	$\hat{\mathbf{T}} = -(\hbar^2/2m)(\nabla)^2$
Total energy	$H = (p)^2/2m + V(r)$	$\hat{\mathbf{H}} = -(\hbar^2/2m)(\nabla)^2 + V(\mathbf{r})$

2.1.1 Time-independent Schrödinger equation

We use the method of separation of variables, and look for solutions for the product:

$$\Psi(x,t) = \psi(x)\varphi(t). \tag{2.4}$$

We now insert this into the full SE and divide by $\psi\varphi$ we get:

$$i\hbar\frac{1}{\varphi}\frac{d\varphi}{dt} = -\frac{\hbar^2}{2m}\frac{1}{\psi}\frac{d^2\psi}{dx^2} + V(x). \tag{2.5}$$

We now see that the left side depends only on time t and the right side is only dependent on x. This means the expressions are constant. We will call this constant E. This gives us two equations:

$$\frac{d\varphi}{dt} = -\frac{iE}{\hbar}\varphi, \tag{2.6}$$

$$\boxed{-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + V\psi = E\psi} \tag{2.7}$$

Equation (2.7) is called the Time-independent Schrödinger equation.

Equation (2.6) has the solution:

$$\varphi(t) = e^{-iEt/\hbar}. \tag{2.8}$$

This is just a phase and dissapears when we calculate $\Psi^*\Psi$.

2.1.2 The Schrödinger equation in 3 dimensions

We introduce the Laplacian in Cartesian coordinates: [1]

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}. \tag{2.9}$$

The Time-independent Schrödinger equation then becomes

$$-\frac{\hbar^2}{2m}\nabla^2\psi + V\psi = E\psi. \tag{2.10}$$

This is the main equation used in this thesis, and is often written as

$$\hat{H}\psi = E\psi. \tag{2.11}$$

Here $\hat{H}$ is the Hamiltonian.

We can also derive the time independent Schrödinger equation in another way. The classical Hamiltonian reads: [11]

$$H(x,p) = \frac{p^2}{2m} + V(x). \tag{2.12}$$

By substitution $p = (\hbar/i)(d/dx)$ [7] for the quantum mechanical momentum operator. This gives

$$\hat{H} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x). \tag{2.13}$$

2.2 The potential

We now need a complete expression for $\hat{H}$.

The Hamiltonian consists of two parts, kinetic and potential energy

$$\hat{H}(x) = \hat{T}(x) + \hat{V}(x). \tag{2.14}$$

We look at the first term, which is the kinetic energy,

$$T = \frac{P^2}{2M} + \sum_{j=1}^{N}\frac{p_j^2}{2m}. \tag{2.15}$$

We use the quantum operator $p_k \to -i\hbar\partial/\partial x_k$, taken from [7]. This gives us

$$\hat{T}(x) = -\frac{\hbar^2}{2M}\nabla_0^2 - \sum_{j=1}^{N}\frac{\hbar^2}{2m}\nabla_j^2. \tag{2.16}$$

We now look at the potential part of the Hamiltonian [5],

$$\hat{V}(x) = -\sum_{j=1}^{N}\frac{Ze^2}{(4\pi\epsilon_0)r_j} + \sum_{i=1,i<j}^{N}\frac{e^2}{(4\pi\epsilon_0)r_{ij}}. \tag{2.17}$$

2.3 The Born-Oppenheimer approximation

We know the proton-electron mass ratio is around 1/1836 and the neutron-electron mass ratio is about 1/1839. We will use the Born-Oppenheimer approximation [10] and consider only electronic degrees of freedom, The kinetic energy is then only given by the kinetic energy of the various electrons, namely,

$$\hat{T} = -\sum_{j=1}^{N} \frac{\hbar^2}{2m} \nabla_j^2. \tag{2.18}$$

2.4 Atomic units

In this thesis we use atomic units in our calculations. This means we set

$$m = e = \hbar = 4\pi\epsilon_0 = 1. \tag{2.19}$$

This gives us the equation:

$$\boxed{\Big[-\sum_{i=1}^{N} \frac{1}{2}\nabla_i^2 - \sum_{i=1}^{N} \frac{Z}{r_i} + \sum_{i<j}^{N} \frac{1}{r_{ij}} \Big] \psi(x) = E\psi(x).} \tag{2.20}$$

2.5 Scaling

We will work in atomic units. To convert between SI units we need to use the conversion factors of table 2.1 [7]

Table 2.1: Scaling table

Quantity	Si	Atomic Units
Electron mass, m	$9.109 * 10^{-31}$ kg	1
Charge, w	$1.602 * 10^{-19}$ C	1
Planck's reduced constant, $\hbar$	$1.055 * 10^{-34}$ Js	1
Permibility, 4 $\pi\epsilon_0$	$1.113 * 10^{-10} C^2 J^{-1} m^{-1}$	1
Energy, $\frac{e^2}{4\pi\epsilon_o a_0}$	$27.211 eV$	1
Length, $a_0 = \frac{4\pi\epsilon_0\hbar^2}{me^2}$	$0.529 * 10^{-10}$	1

2.6 The variance.

We can now look at the expectation value of the Hamiltonian

$$\langle H\rangle = \int \psi^* \hat{H}\psi dx = E\int |\psi|^2 dx = E. \tag{2.21}$$

We can also look at $\langle H^2\rangle$

$$\langle H^2\rangle = \int \psi^* \hat{H}^2\psi dx = E\int \psi^* \hat{H}\psi dx = E^2\int |\psi|^2 dx = E^2. \tag{2.22}$$

This gives us a variance and standard deviation of zero if we have the exact eigenfunctions, namely

$$\sigma_H^2 = \langle H^2\rangle - \langle H\rangle^2 = E^2 - E^2 = 0. \tag{2.23}$$

If this is to be true the wave function must be the exact correct wavefunction. The variance is therefore a good measure of how good our trial wavefunction is.

2.7 Slater determinant

As a trial wavefunction we use a Slater determinant. [12] We see that if two rows are equal the determinant becomes zero. This follows from the Pauli principle [11] which states that the total wavefunction of a fermionic system has to be antisymmetric. As a consequence, when using a single-particle basis, no single-particle state can be occupied by more than one fermion. If two rows of the slater determinant is equal the determinant becomes zero. The Slater determinant is antisymmetric under the interchange of any pair of fermions. The Slater determinant is given by

$$\psi = \frac{1}{\sqrt{N}}\begin{vmatrix} \chi_1(x_1) & \chi_2(x_1) & \dots & \chi_N(x_1) \\ \chi_1(x_2) & \chi_2(x_2) & \dots & \chi_N(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \chi_1(x_N) & \chi_2(x_N) & \dots & \chi_N(x_N) \end{vmatrix}, \tag{2.24}$$

where the various functions χ_i represent single-particle states. We will use both hydrogenic and Gaussian orbitals to represent hese. Since the Hamiltonian is spin independent, we can here split the Slater determinant into two parts. [7] This gives us

$$\psi \propto Det\uparrow Det\downarrow = |S\uparrow||S\downarrow|. \tag{2.25}$$

We will use both hydrogenic orbitals and Gaussian orbitals.

2.8 Jastrow factor

The Slater determinant is not the whole story. To include two-body and higher order correlation effects we need a Jastrow factor. [7] This factors cancels also out the divergency in the relative distance when two electrons get close to each other. The ideea is to make the trial wavefunction closer to the real one.

$$\boxed{J = \prod_{i<j}^{N} exp\Big(\frac{a_{ij} r_{ij}}{1+\beta r_{ij}}\Big).} \tag{2.26}$$

Here i and j are indeces that refer to different particles. Here a_{ij} is 0.5 if the spins of i and j are parallel and 0.25 if they are not.
We will later see if this Jastrow factor gives us a better result than without. We will test both.

2.9 Full wave function

We have now the full wave function as the product of spin up and spin down Slater determinants and the jastrow factor

$$\boxed{\psi = |S \uparrow||S \downarrow|J.} \tag{2.27}$$

2.10 Hartree-Fock

The Hartree-Fock method is used to solve the time-independent Schrödinger equation.
This method uses five major simplifications. [13]

- The Born-Oppenheimer approximation is assumed
- Relativistic effects are neglected
- The solution is a linear combination of basis functions
- The ansatz for the ground state is a Slater determinant
- Electron correlations beyond one-particle-one-hole excitations are neglected

We will not develop the Hartree-Fock equations in this thesis as they are not directly used in Variational Monte Carlo, however we will later look at Gaussian orbitals which are a result of Hartree-Fock calculations.

2.11 Spin

Spin is an intrinsic property of every elementary particle. [11] Half integer spin particles are called Fermions, and make up ordinary matter. Particles with integer spin are called Bosons, such as photons. Fermions must obey the Pauli exclusion principle, which states that no two particles can have the same quantum numbers. (They cannot be in the same position with the same spin)
We see this in the Slater determinant. If two rows are equal the determinant becomes zero.

Chapter 3

Quantum Monte-Carlo

3.1 Monte Carlo integration

The Monte Carlo method is a stochastic method for integration using random variable. It can be used for any number of dimensions. Conventional integration gives us:

$$\int_{\Omega} f(x)d\Omega = \sum_{i=1}^{m} w_i f(x_i). \tag{3.1}$$

Here w_i are weights found by the distribution of the random numbers x_i If the evaluation points are equally spaced the weights becomes the length of the interval divided by the number of integration points, m,

$$\int_0^1 f(x)dxdy = \frac{1}{m}\Big(\sum_{i=1}^{m} f(x_i) + O(\frac{1}{m}\Big). \tag{3.2}$$

We can also do this in two dimensions (or any higher dimension)

$$\int_0^1\int_0^1 f(x)dx = \frac{1}{m^2}\Big(\sum_{i=1}^{m}\sum_{j=1}^{m} f(x_i, y_j) + O(\frac{1}{m}\Big). \tag{3.3}$$

Integration over a d-dimensional cube requires m^d integration points to get a $1/m$ order of accuracy. In Variational Monte Carlo which we will develop in this thesis (VMC) we have N particles and each of these have d dimensions so we have m^{dN} integration points to get a $1/m$ order of accuracy.

3.2 Importance sampling

In order to select properly and efficiently the integration points, we will use importance sampling, where the Fokker-Planck equation is used to select new integration points. The Fokker-Planck equation [10] is given as:

$$\frac{\partial P(r,t)}{\partial t} = D\nabla \cdot \left[\Big(\nabla - F(r,t)\Big)P(r,t)\right]. \tag{3.4}$$

The new suggested move part of the algorithm becomes [5]

$$r_{new} = r_{old} + \chi + DF(r_{old}\delta t). \tag{3.5}$$

Here χ is a random number and F is the so-called quantum force given by: [6]

$$F(r,t) = \frac{2}{|\psi(r,t)|}\nabla|\psi(r,t)|. \tag{3.6}$$

The factor used in the Metropolis test is then the Greens function. [3]

$$\boxed{G(i \to j) \propto e^{(x_i - x_j - D\delta F(x_i))^2/4D\delta\tau}.} \tag{3.7}$$

These equations can be found in the files mcintegrator.cpp and QuantunForce.cpp.

3.3 The Metropolis Algorithm

Here we will look at the Metropolis Algorithm [4], which is a Markov chain Monte Carlo (MCMC) method, generally used for sampling from multidimensional distributions.

Here we have a reversible Markov prosess. Using detailed balance we have the following ratio between probabilities

$$P_i W(i \to j) = P_j W(j \to i). \tag{3.8}$$

Here P_i is the probability density configuration of i and $W(i \to j)$ is the transition probability between states i and j.

We can write $W(i \to j)$ as a product between the probability of selection configuration j given i $g(i \to j)$ and the probability of accepting a move. $A(i \to j)$

$$P_i g(i \to j)A(i \to j) = P_j g(j \to i)A(j \to i). \tag{3.9}$$

Inserting the probability distribution as the wavefunction squared and the selection probability as the Green's function (See equation (3.7)) We get

$$|\psi_i|^2 G(i \rightarrow j) A(i \rightarrow j) = |\psi_j|^2 G(j \rightarrow i) A(j \rightarrow i). \tag{3.10}$$

$$\frac{A(j \rightarrow i)}{A(i \rightarrow j)} = \frac{G(i \rightarrow j)|\psi_i|^2}{G(j \rightarrow i)|\psi_j|^2} \equiv R_G(j \rightarrow i) R_\psi(j \rightarrow i)^2. \tag{3.11}$$

A step from a position i to a new position j is automatically accepted so $A(i \rightarrow j) = 1$
This leaves us with

$$A(j \rightarrow i) = R_G(j \rightarrow i) r_\psi(j \rightarrow i)^2. \tag{3.12}$$

This gives us the acceptance/rejection rules:

$$A(i \rightarrow j) = min(R_G(i \rightarrow j) R_\psi(i \rightarrow j)^2, 1). \tag{3.13}$$

In the case of VMC without importance sampling we have

$$A(i \rightarrow j) = min(R_\psi(i \rightarrow j)^2, 1) = min(\frac{|\psi_i|^2}{|\psi_j|^2}, 1). \tag{3.14}$$

With importance sampling we have the Metropolis/Hasting acceptance ratio given as the ratio of the trial wave function at the new and old positions is given by

$$R \equiv \frac{\psi^{new}}{\psi^{old}} = \frac{|S|^{new}_{\uparrow}}{|S|^{old}_{\uparrow}} \frac{|S|^{new}_{\downarrow}}{|S|^{old}_{\downarrow}} \frac{J^{new}}{J^{old}}. \tag{3.15}$$

The acceptance/rejection ratio then becomes

$$\boxed{R_S = R^2 \frac{G(old, new)}{G(new, old)} = \frac{|\psi^{new}|^2}{|\psi^{old}|^2} \frac{G(old, new)}{G(new, old)}} \tag{3.16}$$

We use this in the file: mcintegrator.cpp.

3.4 Computing the energy

The expectation value of the energy is: [7]

$$\langle E \rangle = \frac{\int dR \psi^*(R) \hat{H}(R) \psi(R)}{\int dR \psi^*(R) \psi(R)} \tag{3.17}$$

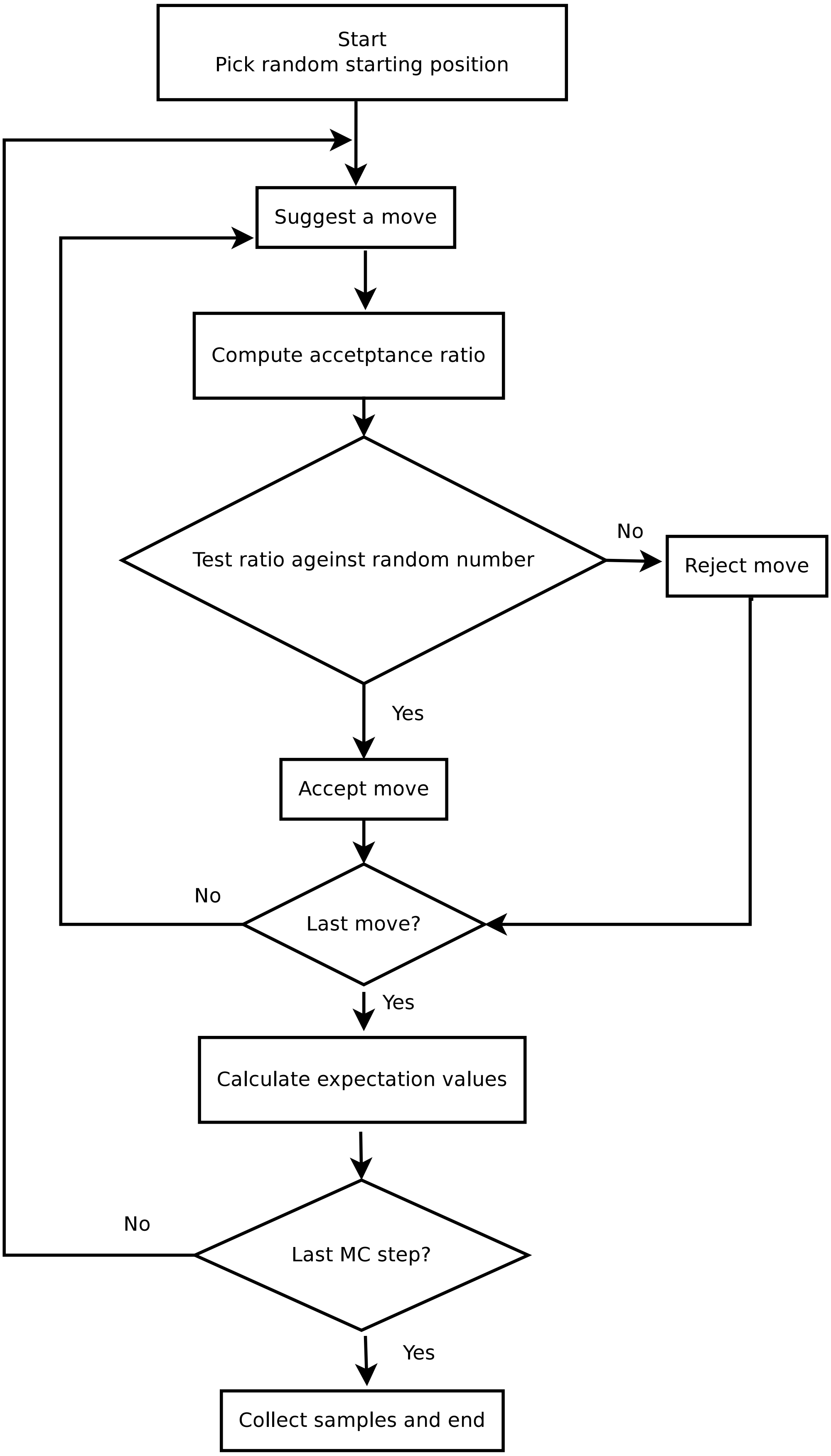

Start
Pick random starting position
Suggest a move
Compute accetptance ratio
Test ratio ageinst random number
No
Reject move
Yes
Accept move
No
Last move?
Yes
Calculate expectation values
No
Last MC step?
Yes
Collect samples and end

Figure 3.1: Flowchart of VMC.

We can rewrite this as

$$\boldsymbol{\langle E\rangle = \frac{\int |\psi(R)|^2 \frac{\hat{H}\psi(R)}{\psi(R)} dR}{\int |\psi(R|^2 dR} = \int \frac{|\psi(R)|^2}{\int |\psi(R)|^2 dR} \Big(\frac{\hat{H}\psi(R)}{\psi(R)}\Big) dR.} \tag{3.18}$$

We now define a probability distribution as: [5]

$$\boxed{\boldsymbol{P(R) = \frac{|\psi_T(R)|^2}{\int |\psi_T(R)|^2 dR}.}} \tag{3.19}$$

We define the local energy as:

$$\boxed{\boldsymbol{E_L(R) = \frac{1}{\psi_T(R)} \hat{H}\psi_T(R).}} \tag{3.20}$$

We will examine this equation in section 3.6.

The expectation value of the energy then becomes

$$\langle E\rangle = \int P(R) E_L(R) dR. \tag{3.21}$$

Since P(R) is a probability distribution we get: [5]

$$\boxed{\langle E\rangle \approx \frac{1}{N} \sum^{N} E_L(x_i).} \tag{3.22}$$

This is the equation used to find the energy of the system, and is listed in tables in chapter 8.

3.5 The variational principle

The variational principle states that the expectation value of the Hamiltonian is an upper bond to the true ground state energy E_0

$$E_0 \leq \langle E\rangle. \tag{3.23}$$

We want to show this by setting the wave function as a sum of eigenstates:

$$\psi_T(R) = \sum \psi_i(R). \tag{3.24}$$

We insert this into equation (3.17)

$$\langle E\rangle = \frac{\sum_{mn} a_m^* a_n \int dR\psi_m^*(R)H(R)\psi_n(R)}{\sum_{mn} a_m^* a_n \int dR\psi_m^*(R)\psi_n(R),} \tag{3.25}$$

$$= \frac{\sum_{mn} a_m^* a_n \int dR\psi_m^*(R)E_n(R)\psi_n(R)}{\sum_n a_n^2}. \tag{3.26}$$

This can be rewritten as [7]

$$\boxed{\frac{\sum_n a_n^2 E_n}{\sum_n a_n^2} \geq E_0.} \tag{3.27}$$

This shows that the value for $\langle E\rangle$ is always greater than the true value E_0

3.6 The Local energy

We must find an analytic expression for the local energy found in equation (3.20)

$$E_L = \frac{1}{\psi_T}H\psi_T. \tag{3.28}$$

$$E_L = \frac{1}{\psi_T}\Big(-\frac{1}{2}\sum_{i=1}^{N}\nabla_i^2 - \sum_{i=1}^{N}\sum_{n=1}^{K}\frac{Z_n}{|R_n - r_i|} + \frac{1}{2}\sum_{i\neq j}^{n}\frac{1}{|r_i - r_j|}\Big)\psi_T. \tag{3.29}$$

$$E_L = \frac{1}{\psi_T}\Big(-\frac{1}{2}\sum_{i=1}^{N}\nabla_i^2\Big)\psi_T - \sum_{i=1}^{N}\sum_{n=1}^{K}\frac{Z_n}{|R_n - r_i|} + \frac{1}{2}\sum_{i\neq j}^{n}\frac{1}{|r_i - r_j|}. \tag{3.30}$$

We fist examine the kinetic energy part. The other parts are trivial. We split the wavefunction into spin up, spin down and a Jastrow factor

$$\frac{\nabla^2\psi_T}{\psi_T} = \frac{1}{S_\uparrow S_\downarrow J}\nabla^2(S_\uparrow S_\downarrow J) = \nabla(\nabla S_\uparrow S_\downarrow J + S_\uparrow \nabla S_\downarrow J + S_\uparrow s_\downarrow \nabla J), \tag{3.31}$$

$$\boxed{= \frac{1}{S_\uparrow}\nabla^2 S_\uparrow + \frac{1}{S_\downarrow}\nabla^2 S\downarrow + \frac{1}{J}\nabla^2 J + 2(\frac{1}{S_\uparrow S_\downarrow}\nabla S_\uparrow \nabla S_\downarrow + \frac{1}{S_\uparrow J}\nabla S_\uparrow \nabla J + \frac{1}{S_\downarrow J}\nabla S_\downarrow \nabla J).} \tag{3.32}$$

We will look deeper into these equations in chapter 4, where we will find optimized equations for each term. The c++ implementation of these equations is found in chapter 7.

The implementation for this local energy is found in: LocalEnergyOpt.cpp.

Chapter 4

Optimizations

We here wish to optimize the program, and by using the inverse matrix of the Slater determinant we can save a lot of floating point operations.

Updating the invese for each cycle runs as $\mathcal{O}(N^3)$ but the optimized algorithm explained below runs as $\mathcal{O}(N^3/4)$. [7].

We will also look at optimizing the equations containing the Jastrow factor.

4.1 Optimizing the Slater determinant

We define the Slater matrix S as in equation (2.24) as:

$$\mathbf{S_{ij}} \equiv \chi_\mathbf{j}(\mathbf{r_i}), \tag{4.1}$$

where $\chi_j(r_j)$ is the j^{th} wavefuncion of the particle r_i.
Only the i^{th} particle is moved in each cycle.
For the inverse of the Slater determinant we have from [2]

$$\mathbf{S^{-1}} = \frac{\mathbf{1}}{|\mathbf{S}|}\mathbf{adjS}, \tag{4.2}$$

where $adj\mathbf{S}$ is the transpose of the cofactor matrix.
We can rewrite this as the transpose of the cofactor matrix

$$\mathbf{S^{-1}} = \frac{\mathbf{C^T}}{|\mathbf{S}|}, \tag{4.3}$$

$$\mathbf{S^{-1}_{ji}} = \frac{\mathbf{C_{ij}}}{|\mathbf{S}|}. \tag{4.4}$$

Now the determinant can be expressed as a cofactor expansion. (Cramer's rule) [7]

$$|\mathbf{S}| = \sum_{\mathbf{i}} \mathbf{S_{ij} C_{ji}}. \tag{4.5}$$

Then we have:

$$C_{ji}^{new} = C_{ji}^{old} = (S_{ij}^{old})^{-1}|S^{old}|. \tag{4.6}$$

We use this as a basis for optimizing the code.

4.2 Optimizing the Slater ratio

We need to find the Slater ratio

$$R_s = \frac{|S|^{new}}{|S|^{old}}, \tag{4.7}$$

$$R_S = \frac{|S^{old}| \sum_i S_{ji}^{new} (S_{ij}^{old})^{-1}}{|S^{old}| \sum_i S_{ji}^{old} (S_{ij}^{old})^{-1}}, \tag{4.8}$$

$$= \frac{\sum_i S_{ji}^{new} (S_{ij}^{old})^{-1}}{I_{jj}}. \tag{4.9}$$

We can then use this as an optimization.

$$\boxed{R_S = \sum_i \chi_i(r_j^{new})(S_{ij}^{old})^{-1}.} \tag{4.10}$$

The implementation of this is found in WaveFunction.cpp.

4.3 Optimizing the $\nabla_i |S|/|S|$ ratio.

We will now derive the $\nabla_i |S|/|S|$ ratio using the inverse Slater matrix,

$$\frac{\nabla_i |S|}{|S|} == \frac{\nabla_i \sum_k S_{ji} C_{ji}}{\sum_k S_{ji} C_{ji}}, \tag{4.11}$$

$$= \frac{\nabla_i \sum_k S_{ji} (S_{ij})^{-1} |S|}{\sum_k S_{ji} (S_{ij})^{-1} |S|}, \tag{4.12}$$

$$\boxed{= \sum_k \nabla_i \chi_k (r_i^{new})(S_{ki}^{new})^{-1}} \tag{4.13}$$

.

The implementation for this is found in: QuantumForce.cpp and LocalEnergyOpt.cpp.

4.4 Optimizing the $\nabla_i^2|S|/|S|$ ratio.

We will now derive the $\nabla_i^2|S|/|S|$ ratio using the inverse Slater matrix,

$$\frac{\nabla_i^2|S|}{|S|} = \frac{\nabla_i^2 \sum_k S_{ji}C_{ji}}{\sum_k S_{ji}C_{ji}}, \tag{4.14}$$

$$= \frac{\nabla_i^2 \sum_k S_{ji}(S_{ij})^{-1}|S|}{\sum_k S_{ji}(S_{ij})^{-1}|S|}, \tag{4.15}$$

$$\boxed{= \sum_k \nabla_i^2 \chi_k(r_i^{new})(S_{ki}^{new})^{-1}.} \tag{4.16}$$

These expressions speeds up the algorithm. The implementaion of this is found in LocalEnergyOpt.cpp.

4.5 Updating the inverse Slater determinant.

We now want to find the inverse of the Slater determinant without using the CPU intensive inversion algorithm. For updating the inverse we have [7] to first we calculate a matrix $\tilde{I}_{ij}$ used in the calculation.

$$\tilde{I}_{ij} = \sum_l S_{il}^{new}(S_{lj}^{old})^{-1}. \tag{4.17}$$

Now we find all values where $j \neq i$

$$\boxed{(S_{kj}^{new})^{-1} = (S_{kj}^{old})^{-1} - \frac{1}{R_S}(S_{ji}^{old})^{-1}\tilde{I}_{ij} j \neq i.} \tag{4.18}$$

Then we find all values where $i = j$

$$\boxed{(S_{ki}^{new})^{-1} = \frac{1}{R_S}(S_{ki}^{old})^{-1} else.} \tag{4.19}$$

Here R_S is the Slater ratio found earlier.
The implementation of this is found in WaveFunction.cpp.

4.6 Optimizing the Jastrow ratio

We set

$$g_{kj} = \frac{a_{kj}r_{kj}}{1+\beta r_{kj}}. \tag{4.20}$$

We can express the Jastrow ratio as.

$$log\frac{J^{new}}{J^{old}} = \sum_{k<j=1}^{N}\Big[\frac{a_{kj}r_{kj}^{new}}{1+\beta r_{kj}^{new}} - \frac{a_{kj}r_{kj}^{old}}{1+\beta r_{kj}^{old}}\Big], \tag{4.21}$$

$$\equiv \sum_{k<j=1}^{N}[g_{kj}^{new} - g_{kj}^{old}]. \tag{4.22}$$

Changing r_i only changes r_{ij}

$$r_{kj}^{new} = r_{kj}^{old} k \neq i, \tag{4.23}$$

$$log\frac{J^{new}}{J^{old}} = \sum_{k<j=1}^{N}[g_{kj}^{old} - g_{kj}^{old}] + \sum_{j=1}^{N}[g_{ij}^{new} - g_{ij}^{old}], \tag{4.24}$$

$$log\frac{J^{new}}{J^{old}} = \sum_{j=1}^{N}\Big[\frac{a_{ij}r_{ij}^{new}}{1+\beta r_{ij}^{new}} - \frac{a_{ij}r_{ij}^{old}}{1+\beta r_{ij}^{old}}\Big]. \tag{4.25}$$

Taking the exponent on both sides gives

$$\boxed{\frac{J^{new}}{J^{old}} = exp\Big(\sum_{j=1}^{N} a_{ij}\Big[\frac{r_{ij}^{new}}{1+\beta r_{ij}^{new}} - \frac{r_{ij}^{old}}{1+\beta r_{ij}^{old}}\Big]\Big).} \tag{4.26}$$

Here i is the moved fermion.
The implementation of this is found in WaveFunction.cpp.

4.7 Optimizing the $\frac{\nabla_i J}{J}$ Ratio.

The x component of the Jastrow gradient ratio is then:

$$\frac{1}{J}\frac{\partial J}{\partial x_i} = \frac{1}{\prod_{i<j} exp(g_{ij})}\frac{\partial}{\partial x_i}\prod_{i<j} exp(g_{ij}). \tag{4.27}$$

Using the product rule we transform the expression to a sum

$$\frac{1}{J}\frac{\partial J}{\partial x_i} = \sum_{i\neq j}\frac{1}{exp(g_{ij})}\frac{\partial}{\partial x_i}exp(g_{ij}), \tag{4.28}$$

$$= \sum_{i\neq j}\frac{1}{exp(g_{ij})}\frac{\partial g_{ij}}{\partial x_i}exp(g_{ij}), \tag{4.29}$$

$$= \sum_{i\neq j} \frac{\partial g_{ij}}{\partial x_i}, \tag{4.30}$$

$$= \sum_{i\neq j} \frac{\partial g_{ij}}{\partial r_{ij}} \frac{\partial r_{ij}}{\partial x_i}. \tag{4.31}$$

We look at each part separately. First

$$\frac{\partial g_{ij}}{\partial r_{ij}} = \frac{\partial}{\partial r_{ij}} \Big(\frac{a_{ij} r_{ij}}{1+\beta r_{ij}} \Big), \tag{4.32}$$

$$= \frac{a_{ij}}{(1+\beta r_{ij})^2}. \tag{4.33}$$

then

$$\frac{\partial r_{ij}}{\partial x_i} = \frac{\partial}{\partial x_i} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}, \tag{4.34}$$

$$= \frac{1}{2} 2(x_i - x_j) / \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}, \tag{4.35}$$

$$= \frac{x_i - x_j}{r_{ij}}. \tag{4.36}$$

This combines to

$$\frac{1}{J} \frac{\partial J}{\partial x_i} = \sum_{i\neq j} \frac{a_{ij}}{r_{ij}} \frac{x_i - x_j}{(1+\beta r_{ij})^2}. \tag{4.37}$$

The expression for the gradient becomes

$$\boxed{\frac{\nabla_i J}{J} = \sum_{i<j} \frac{a_{ij}}{r_{ij}} \frac{r_i - r_j}{(1+\beta r_{ij})^2}.} \tag{4.38}$$

The implementation of this is found in QuantumForce.cpp and LocalEnergyOpt.cpp.

4.8 Optimizing the $\frac{\nabla_i^2 J}{J}$ Ratio

We can use similar techniques to find the Laplaican. This is done in [7]

$$\frac{\nabla_i^2 J}{J} = \Big| \frac{\nabla_i J}{J} \Big|^2 + \sum_{i\neq j} \Big(\frac{2}{r_{ij}} \frac{\partial g_{ij}}{\partial r_{ij}} + \frac{\partial^2 g_{ij}}{\partial r_{ij}^2} \Big). \tag{4.39}$$

We find

$$\frac{\partial^2 g_{ij}}{\partial r_{ij}^2} = -\frac{2a_{ij}\beta}{(1+\beta r_{ij})^3}. \tag{4.40}$$

Put together this gives

$$\frac{\nabla_i^2 J}{J} = \Big|\frac{\nabla_i J}{J}\Big|^2 + \sum_{i\neq j}\Big(\frac{2}{r_{ij}}\frac{a_{ij}}{(1+\beta r_{ij})^2} - \frac{2a_{ji}\beta}{(1+\beta r_{ij})^3}\Big), \tag{4.41}$$

$$\boxed{= \Big|\frac{\nabla_i J}{J}\Big|^2 + \sum_{i\neq j} a_{ij}\frac{2}{r_{ij}(1+\beta r_{ij})^3}.} \tag{4.42}$$

The implementation of this is found in LocalEnergyOpt.cpp.

Chapter 5

Orbitals

We will here look at two different types of orbitals, Hydrogenic ones and Gaussian orbitals derived by Hartree-Fock calculations.

The hydrogenic have a $exp(-\alpha r)$ dependency while the Gaussian orbitals have a $exp(-\alpha r^2)$ dependency.

So with the gaussian we do not need a square root operation, which is a slow operation. We will later see how this saves some computational time.

5.1 Hydrogen orbitals

We will first use hydrogen orbitals derived from: [11]

$$E\psi = -\frac{1}{2}\nabla^2\psi + \frac{1}{r}\psi. \tag{5.1}$$

We use here the quantum numbers n, l, m.

- n is the principal quantum number. It can have values $n = 1, 2, 3, ...$
- l is the orbital momentum. It can have values $l = 0, 1, 2, \ldots$
- m is the projection of the orbital momentum. It can have values $-l, -l+1, ..., 0, 1, 2, .., l-1, l$.

We have in spherical coordinates the wavefunction

$$\boxed{\psi(r, \theta, \phi) = R(r)P(\theta)F(\phi).} \tag{5.2}$$

For the $R(r)$ function we have a tool in python: Sympy.physics.hydrogen.R_nl.

Listing 5.1: Radial part of the hydrogen equation.

```
R_nl(n, l, r, Z=1):
```

- n is the main quantum number.
- l is the orbital momentum quantum number.
- r is the radial coordinate.
- Z is the atomic number.

This gives the radial part of the wavefunction with quantum number n and l.

We have for the spherical harmonics: [5]

$$\boxed{P(\theta)F(\phi) = \sqrt{\frac{(2l+1)(l-m)!}{4\pi(l+m)!}}P_l^m(cos(\theta))exp(im\phi).} \tag{5.3}$$

Here P_l^m is the Legendre polynomial, and is given by the Python function sympy.assoc_legendre. It takes as input quantum numbers n and m.

Listing 5.2: Associated Legendre polynomial.

```
assoc_legendre(n,m, x)
```

- n is the main quantum number.
- m is the projection of the orbital momentum quantum number.
- x is the coordinate used.

We then convert it all to Cartesian coordinates. The first few hydrogen orbitals are

$$exp(-\alpha r), \tag{5.4}$$

$$(\alpha r - 2)exp(-\alpha r/2), \tag{5.5}$$

$$x * exp(-\alpha r/2), \tag{5.6}$$

$$y * exp(-\alpha r/2), \tag{5.7}$$

$$z * exp(-\alpha r/2). \tag{5.8}$$

The algorithm used here can be found in: orbitalGenerator.py.

5.2 Gaussian orbitals

A Gaussian orbital is given as: [8]

$$\boxed{\chi_n(x, y, z) = x^i y^j z^k c_n e^{-\alpha_n r^2}.} \tag{5.9}$$

Here x, y and z are Cartesian coordinates and $r^2 = x^2 + y^2 + z^2$ Where we have:

$$i + j + k = l. \tag{5.10}$$

Here l is the total angular momentum

We use a sum of these orbitals as in equation (5.15). Here we have the constants c_n and α_n which we get from EMSL Basis set exchange.

One of the simplest form of GTO (there are many) is STO-3G, (Slater type orbitals) which we will use in this thesis.

In EMSL we first select the orbital type (here STO-G3), then the format (here Turbomole format), and then the type of atom from the periodic table. Then we just press the "Get Basis Set" button, and we get a table as in figure 5.2. See the interface for Basis Set Exchange in figure 5.1.

Letter	Ang mom	Nr Orb
S	0	1
P	1	3
D	2	6
F	3	10
G	4	15
H	5	21

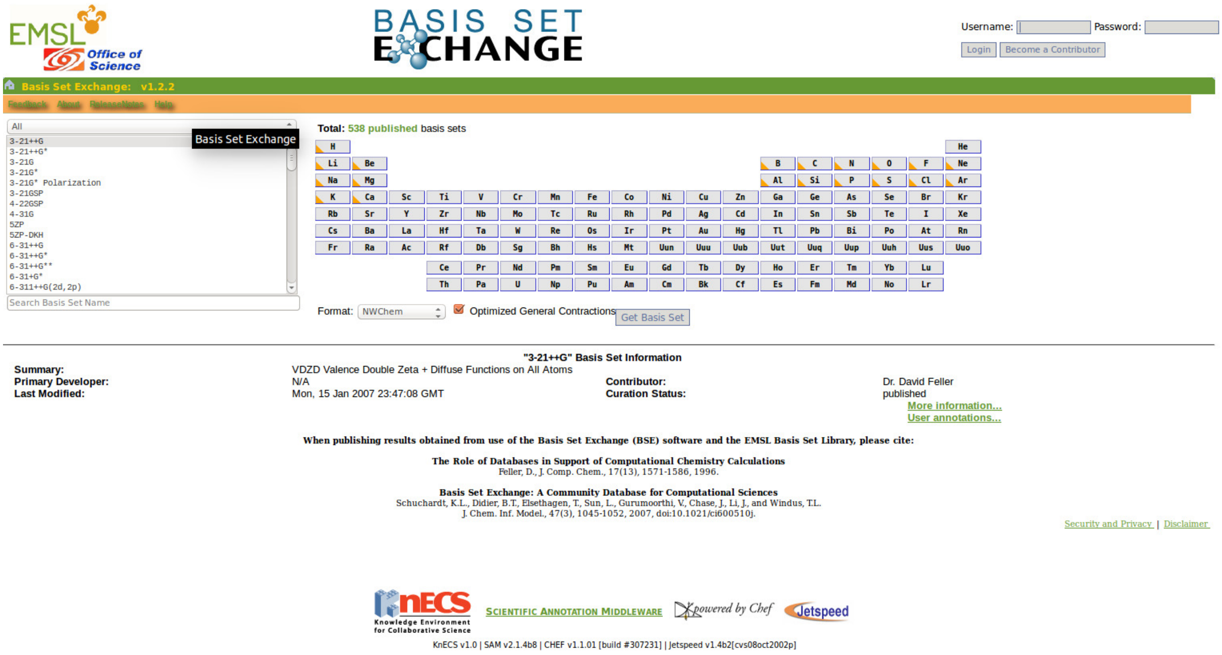
Table 5.1: This table gives us the name of and the number of orbitals for a given angular momentum used in Gaussian orbitals.

The general formula for number of orbitals is seen in equation (5.11).

$$NrOrbitlas = \frac{(l+1)(l+2)}{2}. \tag{5.11}$$

i	j	k
2	0	0
0	2	0
0	0	2
1	1	0
1	0	1
0	1	1

Table 5.2: Here the angular momentum l = 2. This table gives us the different values of i,j and k for a D orbital.


EMSL
Office of Science
BASIS SET EXCHANGE
Username:
Password:
Login
Become a Contributor
Basis Set Exchange: v1.2.2
Feedback
About
ReleaseNotes
Help
All
3-21++G
3-21++G*
3-21G
3-21G*
3-21G* Polarization
3-21GSP
4-22GSP
4-31G
5ZP
5ZP-DKH
6-31++G
6-31++G*
6-31++G**
6-31+G*
6-311++G(2d,2p)
Basis Set Exchange
Search Basis Set Name
Get Basis Set
More information...
User annotations...
Security and Privacy | Disclaimer
KnECS Knowledge Environment for Collaborative Science
Scientific Annotation Middleware
powered by Chef
Jetspeed

Total: 538 published basis sets

Format: NWChem

☑ Optimized General Contractions

"3-21++G" Basis Set Information

Summary: VDZD Valence Double Zeta + Diffuse Functions on All Atoms
Primary Developer: N/A
Last Modified: Mon, 15 Jan 2007 23:47:08 GMT
Contributor: Dr. David Feller
Curation Status: published

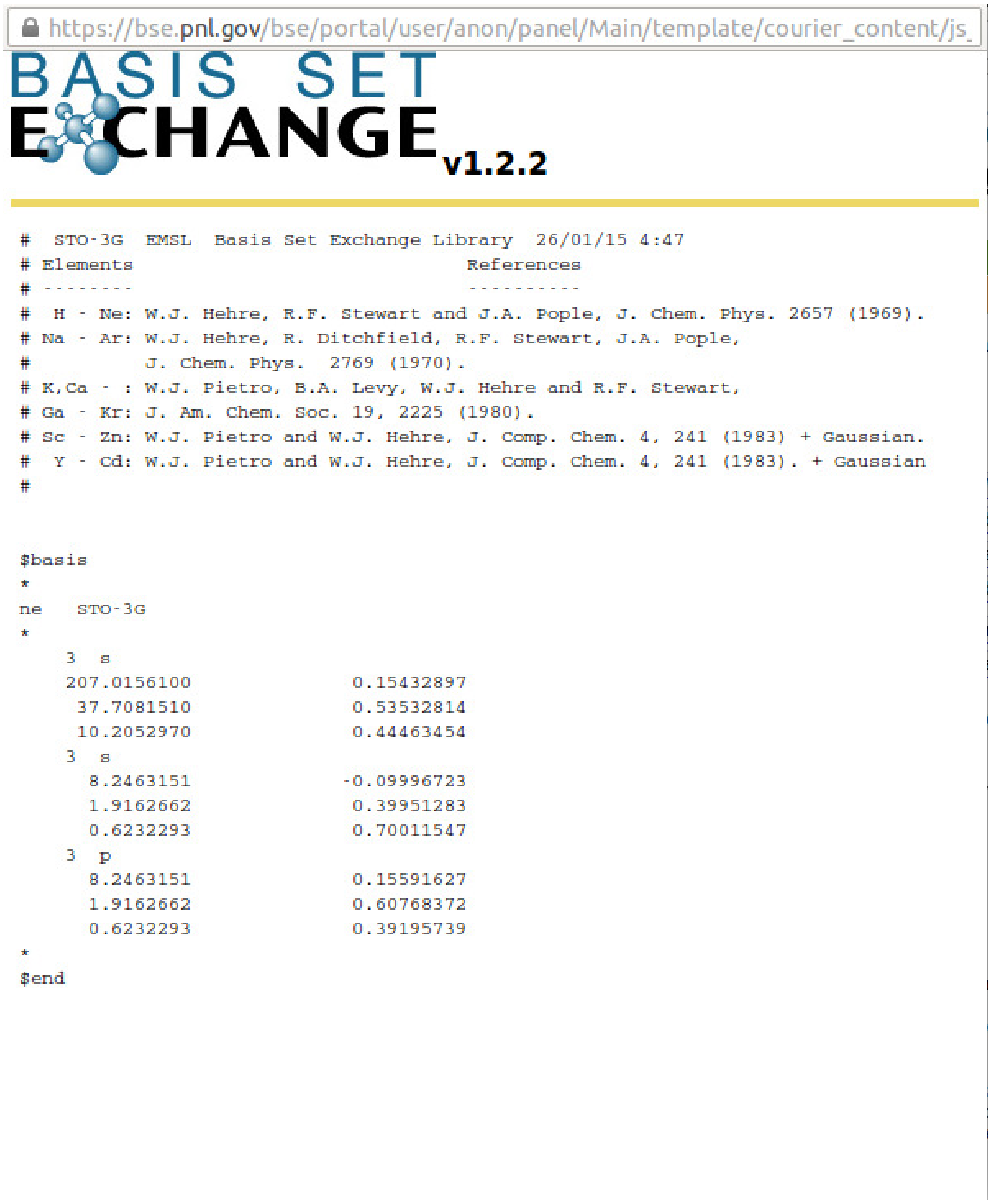
When publishing results obtained from use of the Basis Set Exchange (BSE) software and the EMSL Basis Set Library, please cite:

The Role of Databases in Support of Computational Chemistry Calculations
Feller, D., J. Comp. Chem., 17(13), 1571-1586, 1996.

Basis Set Exchange: A Community Database for Computational Sciences
Schuchardt, K.L., Didier, B.T., Elsethagen, T., Sun, L., Gurumoorthi, V., Chase, J., Li, J., and Windus, T.L.
J. Chem. Inf. Model., 47(3), 1045-1052, 2007, doi:10.1021/ci600510j.

KnECS v1.0 | SAM v2.1.4b8 | CHEF v1.1.01 [build #307231] | Jetspeed v1.4b2[cvs08oct2002p]

Figure 5.1: The interface to Basis set exchange. One can choose which atom one wants, the type of orbital and the format wanted. We use here STO (Slater type orbitals) in Turbomole format.


https://bse.pnl.gov/bse/portal/user/anon/panel/Main/template/courier_content/js_
BASIS SET
EXCHANGE v1.2.2

```
#  STO-3G  EMSL  Basis Set Exchange Library  26/01/15 4:47
# Elements                             References
# --------                             ----------
#  H - Ne: W.J. Hehre, R.F. Stewart and J.A. Pople, J. Chem. Phys. 2657 (1969).
# Na - Ar: W.J. Hehre, R. Ditchfield, R.F. Stewart, J.A. Pople,
#          J. Chem. Phys.  2769 (1970).
# K,Ca - : W.J. Pietro, B.A. Levy, W.J. Hehre and R.F. Stewart,
# Ga - Kr: J. Am. Chem. Soc. 19, 2225 (1980).
# Sc - Zn: W.J. Pietro and W.J. Hehre, J. Comp. Chem. 4, 241 (1983) + Gaussian.
#  Y - Cd: W.J. Pietro and W.J. Hehre, J. Comp. Chem. 4, 241 (1983). + Gaussian
#


$basis
*
ne   STO-3G
*
    3  s
    207.0156100              0.15432897
     37.7081510              0.53532814
     10.2052970              0.44463454
    3  s
      8.2463151             -0.09996723
      1.9162662              0.39951283
      0.6232293              0.70011547
    3  p
      8.2463151              0.15591627
      1.9162662              0.60768372
      0.6232293              0.39195739
*
$end
```

Figure 5.2: We see here the output for a Ne atom. We see we get two s orbitals and one p orbial. The first number is the α_n value and the second is the c_n value.

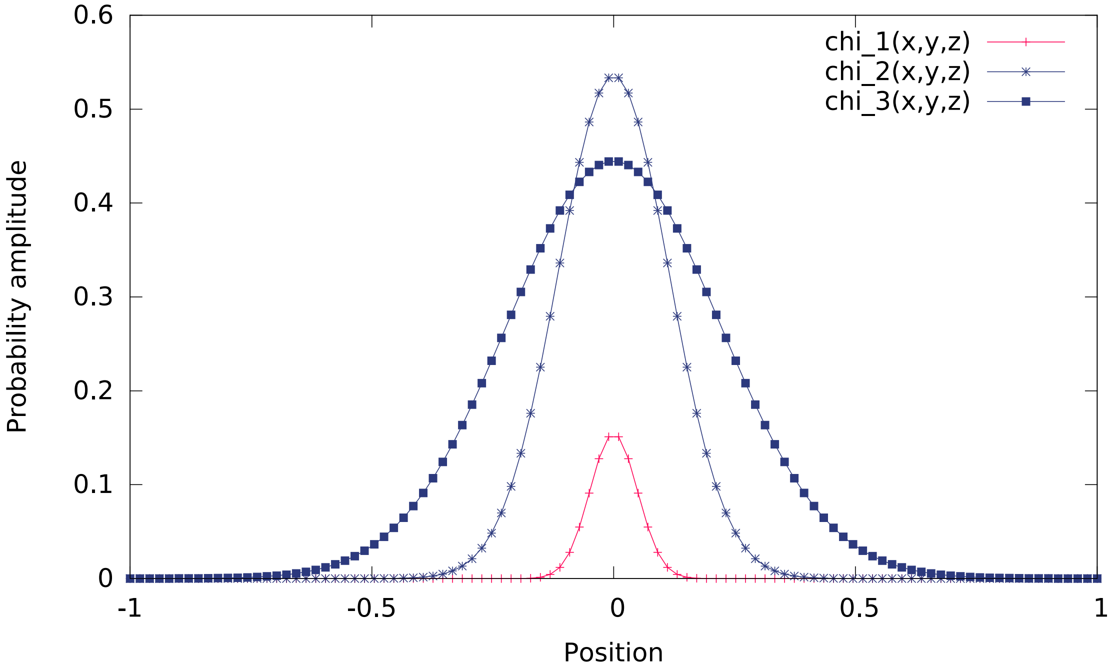


Figure 5.3: Here is a plot of equations (5.12), (5.13) and (5.14).The position axis is in hartree length units and the other axis is a probability amplitude

As an example we look at a S orbital where we have 3 contracted GTOs. since l = 0 we have $x^0y^0z^0 = 1$. We have a plot of these functions in figure 5.3.

$$\chi_1(x,y,z) = 0.15432897 * exp(-207.0156100 * r^2), \tag{5.12}$$

$$\chi_2(x,y,z) = 0.53532814 * exp(-37.7081510 * r^2), \tag{5.13}$$

and

$$\chi_3(x,y,z) = 0.44463454 * exp(-10.2052970 * r^2). \tag{5.14}$$

The orbital function becomes:

$$\boxed{\phi(x,y,z) = \sum_n N_n \chi_n(x,y,z).} \tag{5.15}$$

This function must be normalized

$$|\langle\phi_n|\phi_n\rangle|^2 = 1. \tag{5.16}$$

From [9] we have the expression for the normalization constant N_n

$$\boxed{N_n = \left(\frac{2\alpha_n}{\pi}\right)^{3/4}\left[\frac{(8\alpha_n)^{i+j+k} i! j! k!}{(2i)!(2j)!(2k)!}\right]^{1/2}.} \tag{5.17}$$

The code to do this is found in: parsebasisset.py

5.3 The gradient and laplacian of the orbitals.

We need to find the gradient function and the laplacian of the orbital functions. To do this we use a python script which does all derivations for us. In python this is very simple. The command *diff* in Sympy symbolically differentiates an expression for a given variable. We then generate a c++ file with functions for the wavefunction, the gradient and the Laplacian for the different orbitals. We use basically the same algorithm for hydrogenic and Gaussian orbitals.

The code that does this for hydrogenic is found in the file: generateDerivatesHydrogenic.py The code that does this for Gaussains is found in the file: generateDerivatesGaussians.py

Chapter 6

Calculation of Molecules, Blocking and Optimized Variational Parameters

In this chapter we present some additional theoretical ingredients needed in our calculations. We start with molecules, and move on to Blocking as a technique to compute the standard deviation and finally how to find the optimal variational parameters.

6.1 Homonuclear Diatomic Molecules

We will now look at the Hamiltonian of Homonuclear Diatomic molecules

$$\boxed{\hat{H} = -\frac{1}{2}\sum_{i=1}^{N}\nabla_i^2 + \sum_{i=1}^{N}\Big(\frac{Z}{|R/2+r_i|} + \frac{Z}{|R/2-r_i|}\Big) + \frac{Z^2}{R} + \frac{1}{2}\sum_{i\neq j}^{n}\frac{1}{|r_i-r_j|}.} \tag{6.1}$$

The wave functions are here a super position of two wavefunctions:

$$\psi^{+}(r_i, R) = \psi(r+R/2) + \psi(r_i - R/2), \tag{6.2}$$

$$\psi^{-}(r_i, R) = \psi(r+R/2) - \psi(r_i - R/2). \tag{6.3}$$

For the gradient we use a similar expression:

$$\nabla\psi^{+}(r_i, R) = \nabla\psi(r+R/2) + \nabla\psi(r_i - R/2), \tag{6.4}$$

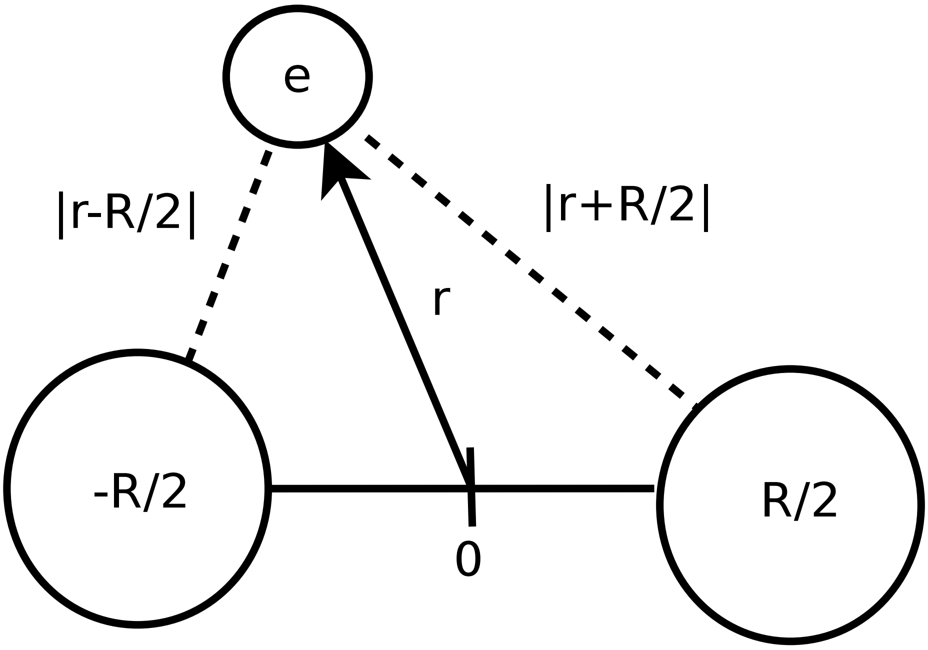


Figure 6.1: A diatomic molecule with one of the electrons. Here R and r are vectors.

$$\nabla\psi^-(r_i,R) = \nabla\psi(r+R/2) - \nabla.\psi(r_i-R/2). \tag{6.5}$$

For the Laplacian we have

$$\nabla^2\psi^+(r_i,R) = \nabla^2\psi(r+R/2) + \nabla^2\psi(r_i-R/2), \tag{6.6}$$

$$\nabla^2\psi^-(r_i,R) = \nabla^2\psi(r+R/2) - \nabla^2\psi(r_i-R/2). \tag{6.7}$$

The code that does this can be found in: Diatomic.cpp.

6.2 Random numbers

Random numbers are a bit tricky on a computer since all algorithms for random numbers are deterministic, but we can make a pretty good approximation with pseudo-random numbers. The random function used in the code delivers values between 0 and 1 derived from a starting seed. The numbers should have little or no correlation between them, meaning a number should not affect the probability for the next number. Also the algorithm should be fast. The most common type is Linear the congruential algorithm

$$N_i = (aN_{i-1} + c)MOD(M) \tag{6.8}$$

Here a, c and M are constants. The variable M should be as large as possible. This gives a random number between 0 and 1 with the equation

$$x_i = N_i/M \tag{6.9}$$

The code with random functions are found in: lib.cpp

6.3 Blocking

We wish here to estimate the covariance in order to find a proper error estimate for ours results.
The problem here is that a brute force method will produce N^2 calculation, and when N is big this will take too long time on an ordinary computer. To do this we use the method of blocking where we save all local energies from a single Monte Carlo simulation and divide it in blocks of size $n_b = N/n$

$$\overline{m_n^r} \equiv \frac{1}{n_b}\sum_{k=1}^{n_b} m_k^r \tag{6.10}$$

,

$$\boxed{\sigma^2(m) = \left\langle m^2\right\rangle - \left\langle m\right\rangle^2 \simeq \overline{m_n^2} - \left(\overline{m_n}\right)^2.} \tag{6.11}$$

We make this calculation with increasing block sizes and plot it in a graph. We can from this graph find approximately the covariance, as the graph will go towards the real value. Example of plot is seen in Figure 6.2.

The code that does this is found in: blockingmain.cpp.

6.4 Gradient descent

We will here develop a method for finding the minimum value. If the minimum is at $x = x_m$ we have:

$$\nabla f(x_m) = 0, \tag{6.12}$$

$$\nabla f(x_m - dx) < 0, \tag{6.13}$$

Table 6.1: delta values

• Brute force I	$\delta_i = \delta \frac{1}{\lvert\nabla f(x_i)\rvert}$
• Brute force II	$\delta_i = \delta$
• Monotone Decreasing	$\delta_i = \delta / i^N$
• Newton's Method	$\delta_i = \frac{1}{\nabla^2 f(x_i)}$

$$\nabla f(x_m + dx) > 0. \tag{6.14}$$

Here dx is an infinite decimal displacement. The method used is

$$x_{i+1} = x_i - \delta_i \nabla f(x_i). \tag{6.15}$$

Newtons methods is an iterative way of finding the zero point of a function f. We choose a random point x_i and draw a straight line to the x axis. The formula for this is:

$$0 = f'(x_n)(x_{n+1} - x_n) + f(x_n). \tag{6.16}$$

We rewrite this as

$$x_{i+1} = x_i - f(x_i)/\nabla f(x_i). \tag{6.17}$$

We use in this thesis Newton's method. We must use numeric approximations for the derivatives

$$\nabla f(x_m) = ((f(x_{m+h}) - f(x_{m-h}))/(2h). \tag{6.18}$$

$$\nabla^2 f(x_m) = ((f(x_{m+h}) - 2f(x_m) + f(x_{m-h}))/(h^2). \tag{6.19}$$

The values for h must here be picked carefully. Finding the minimum value for alpha beta we must find the zero point for the derivative

$$\boxed{x_{i+1} = x_i - \nabla f(x_i)/\nabla^2 f(x_i).} \tag{6.20}$$

The code that does this (with MPI) is found in mpimain.cpp.

6.5 Parallelization

VMC is extremely easy to parallelize with MPI. One has to make sure all the processes start with different seed to the random number generator. So one just runs VMC in parallel and sum the results and divide by the number of nodes.

The Mpi function MPI_Reduce does the job nicely. This function collects and sums or multiplies all results from the different nodes. In this case we sum

Listing 6.1: MPI_Reduce

```
int MPI_Reduce(const void *sendbuf, void *recvbuf,
     int count,MPI_Datatype datatype,
    MPI_Op op, int root, MPI_Comm comm)
```

We look at the inputs and outputs of *MPI_Reduce*:

- *sendbuf* is a pointer to an array we wish to reduce.
- *recvbuf* is where the result is stored after calculation.
- *count* is the number of elements in the *sendbuf* array.
- *MPI_Datatype* is the type of variable used. Here *MPI_Double*.
- *MPI_Op* is the type of operation used. Here we sum and therefore use *MPI_SUM*.
- *root* is the node where the result is sent. (Into *recvbuf*.)
- *MPI_Comm* is here just *MPI_COMM_WORLD*.
- The returned integer is an errorcode.

There was some trouble making Armadillo work on the computer cluster Abel, but after some tweaking it now compiles and runs there.

6.6 Errors

Some few times the start positions give errors.
The simple solution is to run the first timestep of the simulation and check for Nan (Not a number) and reselect if this show up.

Listing 6.2: Check for Nan

```


bool not_done = true;

//Loop until working values are found
while (not_done) {

     not_done = false;

```

```
    //Pick new random points
    for(int i = 0; i < nParticles; i++) {
      for(int j = 0; j < nDimensions; j++) {
         rOld(i,j) = gaussianDeviate(&idum)*sqrt(timestep);
       }
    }

        //Remember the starting point
    rCurr = rOld;
    rNew = rOld;

    oldidum = idum;

    //Get the inverse Slater matrix
    wf->set_inverse(rOld);

    //Find the quatumforce
    qforce_old = qf->quantumforceOpt(rOld);


    //Loop trough all particles
    for (int i = 0; i < nParticles; i++) {


        //Check for errors
        if(!Cycle(0,i))
           not_done = true;

    }
}
```

Finds new starter points which does not give singular inverse.

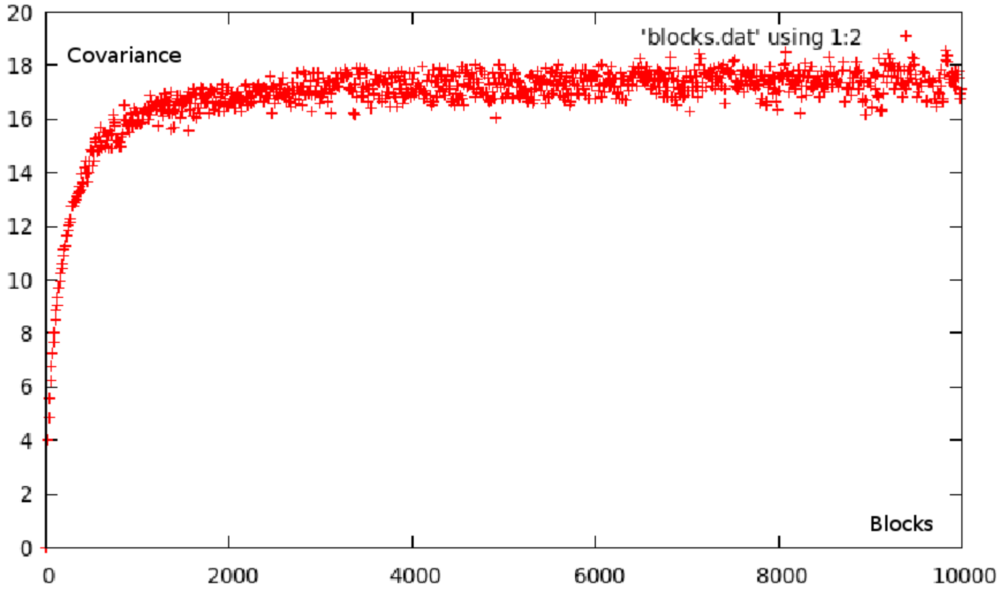


Figure 6.2: Example of blocking plot. The blocks axis shows number of blocks used in calculating the covariance which is plotted here in dimensionless units.

Chapter 7

How to code VMC

We will here examine some of the code used in this thesis. Not all code will be listed, but the most important is listed and explained. The code for this thesis can be found at https://github.com/rogerkj/master

Included are codes for the generation of orbitals. Hydrogenic and Gaussian.

7.1 The C++ files

Let us look at the c++ files in this project:

- main.cpp contains the mpi stuff and calls the main algorithm.
- mcintegrator.cpp contains the main VMC algorithm.
- LocalEnergyOpt.cpp contains the functions for finding the local energy.
- WaveFunction.cpp contains different tools as the algorithm for finding the inverse slater determinant and slater tools. Cause of legacy the file is called WaveFunction tough the wavefunctions are not in this file.
- Hydrogenic.cpp contains the hydrogenic wavefunctions, the gradient of that and the Laplacian.
- Gaussians.cpp contains the Gaussian wavefuncions, the gradient of that and the Laplacian. This file must be swapped manually according to the type of atom used.
- Diatomic.cpp contains functions for the wavefunctions, gradient and laplacian for diatomic molecules.

- QuantumForce.cpp contains the algorithm for finding the Quatum force.
- lib.cpp contains the random function used.
- blockingmain.cpp contains code for the blocking algorithm.
- mpimain.cpp contains the minimum-finding MPI algorithm.

7.2 The Python files

We have some Python scripts for generating hydrogenic and Gaussian orbitals.

- generateHydrogenic.py contains code that runs the main algorithm for hydrogenic orbitals
- generateDerivatesHydrogenic.py contains code for finding the hydrogenic gradient and Laplacian and printing to file.
- orbitalGenerator.py contains code for the generation of hydrogenic orbitals.
- generateGaussians.py contains code that runs the main algorithm for Gaussian orbitals
- generateDerivatesGaussians.py contains code for finding the Gaussian gradient and Laplacian and printing to file.
- orbitalGenerator.py contains code for the generation of Gaussian orbitals.

7.3 C++ code

We will here look at some of the code used in this thesis.

Here we have:

- $dr->waveFunction$ is the used wavefunction.
- $dr->gradient$ is the used gradient
- $dr->laplacian$ is the used Laplacian.
- inv_up is the inverse up Slater derterminant.

- *inv_down* is the inverse down Slater determinant.
- nDimensions is the number of dimensions (here 3)
- nParticles is the number of particles.
- beta is the β value used in the Jastrow factor.
- timestep is a set value who seems to work fine:(0.005).
- idum is the random seed.

7.3.1 Main MVC algorithm

Following is the main VMC algorithm inner loop. Here i is the current particle, $rNew$ and $rOld$ are new and old positions of the particles, respectively. The function gaussianDeviate returns a random Gaussian distributed value.

This code is clipped from the file mcintegrator.cpp

Listing 7.1: Main VMC algorithm

```

    //Make next move
    for(int j = 0; j < nDimensions; j++) {
          rNew(i,j) = rOld(i,j) + gaussianDeviate(&idum
             ) * sqrt(timestep) + qforce_old(i, j) *
             timestep *D;
    }

    //Find Slater determinant
    double Rs = wf->slaterOpt(rNew,rOld,i);


    //Store current slaterdeterminant
    old_inv_up = wf->inv_up;
    old_inv_down = wf->inv_down;

    wf->update_inverse(Rs,rNew,i);

    //Find quantum force
    qforce_new = qf->quantumforceOpt( rNew );

    //get Greensfunction
    double greensfunction = getGreensFunctionRatio(rNew,rOld,qforce_new,qforce_old);
The following code is the quantum force. See section 17


//Check for Jastrow factor
#ifdef JASTROW

    //Find Jastrow factor
    double jastrow = wf->jastrowOpt(rNew,rOld,i);

#else

    //No jastrow factor
    double jastrow = 1.0;

#endif


    // Metropolis test
    if (ran2(&idum) <= greensfunction*Rs*Rs*jastrow) {
    {

        //Step accepted,..   store result

        for(int j = 0; j < nDimensions; j++) {
            rOld(i,j) = rNew(i,j);

        }

        qforce_old = qforce_new;

        if (cycle > nThermalize)
            accetpted++;

    } else {

        //Step not accepted...

        for(int j = 0; j < nDimensions; j++) {
            rNew(i,j) = rOld(i,j);
            }

            qforce_new = qforce_old;

            wf->inv_up = old_inv_up;
            wf->inv_down = old_inv_down;

        }




}
```

7.3.2 Local energy code

We now look at the Local energy code.

We wish to calculate equation (3.30) We first look at the kinetic energy in equation (3.32).

Local energy. Jastrow part.

We fist calculate the two equations:

$$\frac{1}{J}\nabla^2 J \tag{7.1}$$

$$\frac{1}{J}\nabla J \tag{7.2}$$

The optimized formulas for this we find in (4.38) and (4.42)

Here r is here the current position of the particles.

This code is clipped from LocalEnergyOpt.cpp

Listing 7.2: Jastrow factor

```

  double jlaplacian = 0.0;

  rowvec3 rijVec;

  mat jgradient  = zeros(nParticles,dimention);

  for(int i = 0; i < nParticles; i++) {
    for(int j = 0; j < nParticles; j++) {
      if(i != j) {
    rijVec = r.row(i) - r.row(j);
    double rij = norm(rijVec,2);
    jgradient.row(i) += rijVec*dfdr(rij,i,j)/rij;
    jlaplacian += 2*dfdr(rij,i,j)/rij + d2fdr2(rij,i,j);
      }
    }
  }

  double gradient2 = 0.0;
  for (int i = 0; i < nParticles; i++) {
    gradient2 += dot(jgradient.row(i),jgradient.row(i));
  }

  jlaplacian += gradient2;



  //Derivate of the Jastrow function
double LocalEnergyOpt::dfdr(const double &r12, const int &particleNum1, const int &particleNum2){
  return wf->amat(particleNum1, particleNum2)/((1 + beta*r12)*(1 + beta*r12));
}


//Second derivate of the Jastrow function
double LocalEnergyOpt::d2fdr2(const double &r12, const int &particleNum1, const int &particleNum2){
  return -2*wf->amat(particleNum1, particleNum2)*beta/((1 + beta*r12)*(1 + beta*r12)*(1 + beta*r12));
}
```

Local energy. Slater part

We now find the gradient and Laplacian for the Slater determinant

$$\frac{1}{S_\uparrow}\nabla^2 S_\uparrow + \frac{1}{S_\downarrow}\nabla^2 S\downarrow, \tag{7.3}$$

$$\nabla S_\downarrow \tag{7.4}$$

The optimized formulas for this we find in (4.13) and (4.16)

This code is clipped from LocalEnergyOpt.cpp

Listing 7.3: LocalEneergy

```

  mat udgradient  = zeros(nParticles,dimention);

  for (int i = 0; i < nParticles/2; i++){
    for (int j = 0; j < nParticles/2; j++){
      udgradient.row(i) += dr->gradient(r, i, j)*wf->inv_up(j, i);
      udgradient.row(i + nParticles/2) += dr->gradient(r, i + nParticles/2, j)*wf->inv_down(j, i);
    }
  }


  double udlaplacian = 0;

  for (int i = 0; i < nParticles/2; i++){
    for (int j = 0; j < nParticles/2; j++){
      udlaplacian += dr->laplacian(r,i,j)*wf->inv_up(j,i) + dr->laplacian(r, i + nParticles/2 , j)*wf->inv_down(j,i);

    }
  }
```

Core-Electron Electron-Electron equations.

We will now look at the core - electon and electron - electron contribution to the local energy as given in equation (3.30).

$$\sum_{i=1}^{N}\sum_{n=1}^{K}\frac{Z_n}{|R_n - r_i|} + \frac{1}{2}\sum_{i\neq j}^{n}\frac{1}{|r_i - r_j|}. \tag{7.5}$$

This code is clipped from LocalEnergyOpt.cpp

Listing 7.4: Core-Electron Electron-Electron equations.

```

  double coreEl = 0.0;
  for (int i = 0; i < nParticles; i++) {
    double rn = norm(r.row(i));
    coreEl -= charge/rn;
  }

  double electronEl = 0.0;
  for(int i = 0; i < nParticles; i++) {
    for(int j = i+1; j < nParticles; j++) {
      double r12 = norm(r.row(i) - r.row(j));
      electronEl += 1/r12;          }    }
```

Slater-Jastrow dot product

We wish now to find the dot product of the Slater and jastrow factor. Here the matrices udgradient and jgradient are found above

$$\frac{1}{S_{\uparrow}J}\nabla S_{\uparrow}\nabla J + \frac{1}{S_{\downarrow}J}\nabla S_{\downarrow}\nabla J. \tag{7.6}$$

This code is clipped from LocalEnergyOpt.cpp

Listing 7.5: Slater-Jastrow dot product.

```

  for(int i = 0 ; i < nParticles; i++) {
    gradient += dot(udgradient.row(i),jgradient.row(i))
       ;
  }
}
```

7.3.3 Quantum Force

We wish now to make a code for the quantum force as given in (3.6)

$$F = \frac{2}{S_{\uparrow}S_{\downarrow}J}\nabla(S_{\uparrow}S_{\downarrow}J) = \nabla S_{\uparrow}S_{\downarrow}J + S_{\uparrow}\nabla S_{\downarrow}J + S_{\uparrow}s_{\downarrow}\nabla J, \qquad (7.7)$$

Quantum Force: Jastrow part.

We first look at the Jastrow gradient part, and here we use the optimized equation found in equation (4.38).

This code is clipped from QuantumForce.cpp

Listing 7.6: Quantum Force. Jastrow part.

```

  mat jgradient = zeros(nParticles, nDimensions);

  for (int k = 0; k < nParticles; k++){
    for (int i = 0; i < k; i++){
      rowvec3 r12Vec = r.row(k) - r.row(i);
      double r12 = sqrt(r12Vec(0)*r12Vec(0) + r12Vec(1)*r12Vec(1) + r12Vec(2)*r12Vec(2));

          jgradient.row(k) += r12Vec*dfdr(r12, k, i)/r12;
    }
    for (int i = k + 1; i < nParticles; i++){
      rowvec3 r12Vec = r.row(i) - r.row(k);
      double r12 = sqrt(r12Vec(0)*r12Vec(0) + r12Vec(1)*r12Vec(1) + r12Vec(2)*r12Vec(2));

      jgradient.row(k) -= r12Vec*dfdr(r12, k, i)/r12;
    }
  }


//The Jastrow derivate
double QuantumForce::dfdr(double r12, int particleNum1, int particleNum2){
  return wf->amat(particleNum1, particleNum2)/((1 + beta*r12)*(1 + beta*r12));
}
```

Quantum Force: Slater part.

We now look at the Slater part, using the optimized equation (4.13)

This code is clipped from QuantumForce.cpp

Listing 7.7: Finding Quantum Force, Slater part.

```

  //Slater part. See section 18.3
  mat sgradient = zeros(nParticles, nDimensions);


  for (int i = 0; i < nParticles/2; i++){
    for (int j = 0; j < nParticles/2; j++){
      sgradient.row(i) += dr->gradient(r,i, j)*wf->inv_up(j, i);
      sgradient.row(i + nParticles/2) += dr->gradient(r,i + nParticles/2, j)*wf->inv_down(j, i);
    }
  }
```

7.3.4 Update inverse Slater determinant.

We will now look at the algoritm for updating the inverse Slater determinant. *inv_up* and *inv_down*. We find the formulas for this in (4.5).

This code is clipped from WaveFunction.cpp

Listing 7.8: Update inverse Slater determinant.

```

void WaveFunction::update_inverse(double Rs,const mat &r,int i) {

  vec I(nParticles);

  //Update up spin particles
  if(i < nParticles/2) {

    mat new_inv_up  = zeros<mat>( nParticles/2 , nParticles/2 );

    for(int j = 0; j < nParticles/2; j++) {
      I(j) = 0;
      if(i!=j) {
    for(int l = 0; l < nParticles/2; l++)
      I(j) +=  dr->waveFunction(r,i,l)*inv_up(l,j);
      }
    }
    for(int j = 0; j < nParticles/2; j++) {
      if (i!=j) {
    for(int k = 0; k < nParticles/2; k++)
      new_inv_up(k,j) = inv_up(k,j) - I(j)*inv_up(k,i)/Rs;
      }
    }

    for (int k = 0; k < nParticles/2; k++)
      new_inv_up(k,i) = inv_up(k,i)/Rs;

    inv_up = new_inv_up;

    //Update spin down particles
  } else {

    i = i - nParticles/2;

    mat new_inv_down  = zeros<mat>( nParticles/2 , nParticles/2 );

   for(int j = 0; j < nParticles/2; j++) {
      I(j) = 0;
      if(i!=j) {
    for(int l = 0; l < nParticles/2; l++)
      I(j) +=  dr->waveFunction(r,i+nParticles/2,l)*inv_down(l,j);
      }
    }
    for(int j = 0; j < nParticles/2; j++) {
      if (i!=j) {
        for(int k = 0; k < nParticles/2; k++)
          new_inv_down(k,j) = inv_down(k,j) - I(j)*inv_down(k,i)/Rs;
          }
        }

        for (int k = 0; k < nParticles/2; k++)
          new_inv_down(k,i) = inv_down(k,i)/Rs;

        inv_down = new_inv_down;

    }

}
```

7.3.5 Oprimized code for Slater ratio.

We will now look at the optimized code for finding the Slater ratio, as found in equation (4.10)

i is here the current particle

This code is clipped from WaveFunction.cpp

Listing 7.9: Finding the optimized Slater ratio.

```

double WaveFunction::slaterOpt(const mat &rNew,const mat &rOld,int i) {

  double Rs = 0;

  //Process up particles
  if (i < nParticles/2) {
    for (int j = 0 ; j < nParticles/2; j++) {
      Rs += dr->waveFunction(rNew,i,j)*inv_up(j,i);
    }
    //Process down particles
  } else {
    for (int j = 0 ; j < nParticles/2; j++) {
      Rs += dr->waveFunction(rNew,i,j)*inv_down(j,i-nParticles/2);
    }
  }

  return Rs;
}
```

7.3.6 Optimized code for Jastrow factor.

We now look at the code for the updated Jastrow ratio as given in equation (4.26).

This function takes the matrix rNew and rOld as an input. Theese are the new and old particle positions, respectively and i is here the current particle.

This code is clipped from WaveFunction.cpp

Listing 7.10: Finding the optimized Jastrow factor.

```

double WaveFunction::jastrowOpt(const mat &rNew, const mat &rOld, int i) {


  double sum = 0;

  for (int j = 0; j < nParticles; j++) {

    rowvec r12_new = rNew.row(i) - rNew.row(j);
    double r12norm_new = norm(r12_new, 2);

    rowvec r12_old = rOld.row(i) - rOld.row(j);
    double r12norm_old = norm(r12_old, 2);

    sum += amat(i,j)*(r12norm_new / (1 + beta * r12norm_new) - r12norm_old / (1 + beta * r12norm_old));

  }

  double e = exp(sum);

  return e*e;

}
```

7.3.7 Cusp condition algorithm.

We here use a cusp condition. This function returns 0.25 if the spins are paralell and 0.5 if the are not. Here spin[i] is *true* if: i < nParticles/2. else it is false.

This code is clipped from WaveFunction.cpp

Listing 7.11: Cups condition.

```


//Constant a. See section 14.
double WaveFunction::amat(int i, int j) {

  if (spin[i] == spin[j])
    return 0.25;
  else
    return 0.5;

}
```

Part II

Results

Chapter 8

Results

We will here look at the output of the VMC program developed for this thesis. The simulation was run on the supercomputer Abel at the University of Oslo. We ran 100 nodes in each simulation and used from 100 million cycles for the lighter atoms and molecules to approximately one million for the heavier ones.

The code for this minimization (with MPI) is found in mpimain.cpp.

Results for hydrogenic and Gaussian orbitals will be presented, and we wish to see if one works better than the other.

Also we will look at the effect of the Jastrow factor.
First we will look at atoms, and then we will look at diatomic molecules.

8.1 The output

We here use a minimization algorithm found in equation (6.20) for finding the α and β values which give us a minimum according to the variational principle. We move the α and β values simultaneously. The program writes to file and we analyse this after a number of cycles, when the results are converged.

As an example of the output of the program we look at the energy as function of α for the Be atom in figure 8.1, and the β dependence of the energy for Be is shown in figure 8.2.

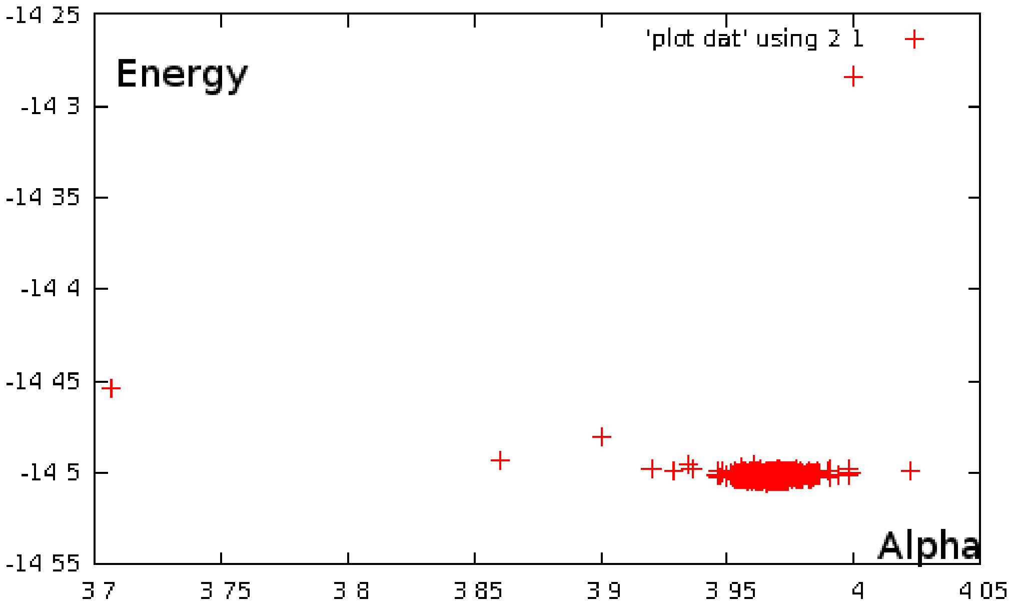


Figure 8.1: This is the plot for the energy of the Be Atom as function of the variational parameter α. The value α is in dimensionless values and the energy is in units of Hartrees.

8.2 About the tables.

In the tables below we have

- α is the dimensionless value used by the hydrogenic wave function.
- β is the dimensionless value in the Jastrow factor.
- VMC are the results from our VMC simulations. These are in units of Hartrees.
- Real are the known correct value found in [6]. The energies are in units of Hartrees.
- rel are in percentage the accuracy of VMC. |real - VMC|/|real|*100.
- The quantity σ is the standard deviation.

This goes for the tables: 8.1, 8.2 8.3 8.4, 8.5, 8.6, 8.7 and 8.8.

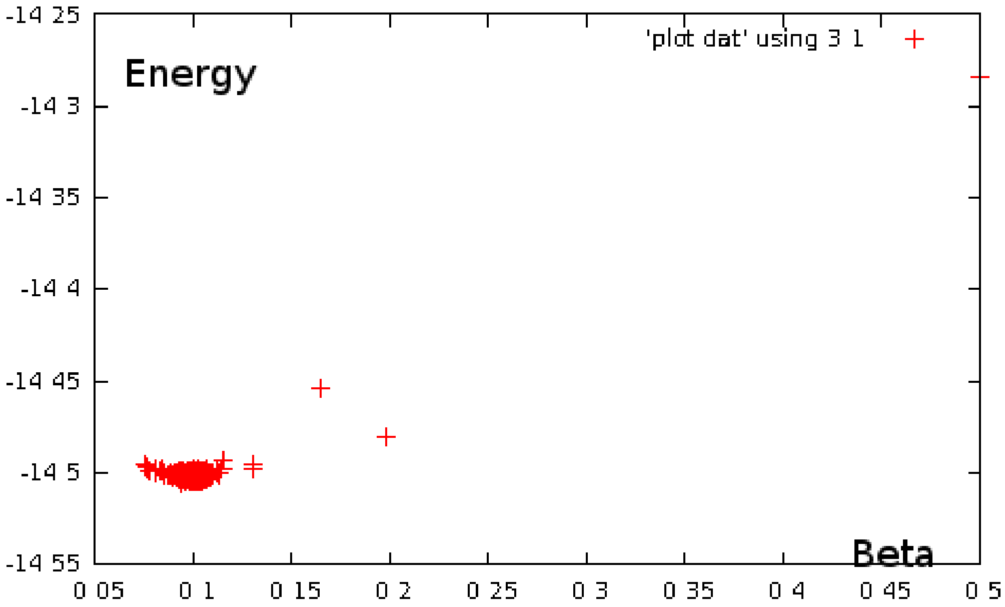


Figure 8.2: This is the plot for the energy of the Be Atom as function of the variational parameter β. The value β is in dimensionless values and the energy is in units of Hartrees.

8.3 Atoms

We will first look at the results and values for some atoms.

Look at tables: 8.1, 8.2, 8.3 and 8.4.

8.3.1 Using Hydrogenic orbitals

First we examine hydrogenic orbitals, with and without Jastrow factor.

Type	α	β	VMC	real	rel	σ
He	1.811	0.50	-2.890	-2.9037	0.47%	1.409e-3
Be	3.97	0.10	-14.50	-14.6674	1.14%	4.282e-3
Ne	10.28	0.083	-127.86	-128.9383	0.84%	1.143e-2
Ar	18.55	0.168	-524.2	-527.544	0.63%	5.828e-2
Kr	35.3	0.40	-2745	-2753.055	0.26%	7.358e-1

Table 8.1: Results of VMC of a single atom using hydrogenic orbitals with Jastrow factor. We see that the error in the *rel* coulomb first increases then decreases, while σ coulomb increases. β first incereases then it decreases. α lies close to the charge.

Type	α	VMC	real	rel	σ
He	1.687	-2.848	-2.9037	1.92%	3.301e-3
Be	3.378	-14.22	-14.667	3.05%	8.535e-3
Ne	7.850	-122.0	-128.938	5.38%	3.426e-2
Ar	14.56	-502.9	-527.544	4.67%	2.291e-1
Kr	28.47	-2674	-2753.055	2.84%	3.081

Table 8.2: Results for VMC of a single atom using hydrogenic orbitals and no Jastrow factor. We see that the rel coulomb first increases then decreases, while σ coulomb increases. α is smaller than in table 8.1 but is still close to the charge. The results here is poorer than with hydrogenic orbitals with jastorw factor.

8.3.2 Using Gaussian orbitals

Now we examine Gaussian orbitals, with and without Jastrow factor.

Type	β	VMC	real	rel	σ
He	0.65	-2.84	-2.9037	2.19%	2.569e-3
Be	0.80	-14.44	-14.667	1.55%	7.897e-3
Ne	2.20	-126.5	-128.938	1.89%	5.696e-2
Ar	2.70	-520.8	-527.544	1.28%	4.164e-1
Kr	2.90	-2718.2	-2753.055	1.23%	30.50

Table 8.3: Results of VMC of a single atom using Gaussian orbitals with a Jastrow factor. We see that the error (rel) coulomb do not follow any pattern, while σ coulomb increases. We see that the β coulomb increases. The result (rel) is poorer than with hydrogenic with Jastrow. We observe a large jump in σ for Kr, but the result (rel) is not very large.

Type	VMC	real	rel	σ
He	-2.81	-2.9037	3.23%	3.546e-3
Be	-14.35	-14.667	2.16%	1.048e-2
Ne	-126.64	-128.938	1.78%	2.282e-1
Ar	-522.7	-527.544	0.92%	7.816e-1
Kr	-2715.1	-2753.055	1.34%	42.97

Table 8.4: Results of VMC of a single atom using Gaussian orbitals with no Jastrow factor. Here we do not use any variables to variate. The result comes therefore from a singe run. We see that the error (rel coulomb) do not follow any pattern, while σ coulomb increases. Also here the σ is quite large for Kr. bur also here the result (rel) is quite good. The result (rel) is not very far from the simulations with a Jastrow factor, in table 8.3. The result (rel) is poorer than with hydrogenic with Jastrow.

We see here reasonable results. From the smallest error (rel) of 0.47% to a larger error of 5.38%
Also we can conclude that here the Jastrow factor has a posetive effect on the results. this means that the trial wavefunction with Jastrow factor is closer to the real wavefunction.

Notice the α value is close to but less than the charge of the atom core. The reason for this is elctron shielding, and keep in mind that the orbitals here uses hydrogenic orbitals who has only one proton in its core. Therefore a good starting point for the α value is the charge.

In tables 8.1 and 8.2 the *rel* coulomb first increases then decreases. If this is a real effect with a reason or it is completely random is unknown. While in table 8.3 and 8.4 the error in coulomb *rel* seems completely random.

8.4 Molecules

We will here look at the results for simple diatomic molecules. The results are listed as functions of the distance R between atoms in Hartree length units found in [6].

See table: 8.5, 8.6, 8.7 and 8.8

8.4.1 Using Hydrogenic orbitals

Here we look at molecules with hydrogen orbitals. With and without Jastrow factor.

Type	R	VMC	α	β	real	rel	σ
H_2	1.4	-1.158	1.296	14.5	-1.1746	1.413%	1.78e-3
Li_2	5.051	-14.745	2.782	0.41	-14.995	1.670%	4.492e-3
Be_2	4.63	-28.820	3.801	0.32	-29.339	1.767%	5.552e-3
B_2	3.005	-48.233	4.805	0.264	-49.418	2.399%	8.595e-3
C_2	2.3481	-73.164	5.662	0.4741	-75.923	3.6339%	1.394e-2
N_2	2.068	-105.63	6.578	0.5044	-109.54	3.571%	2.141e-2
O_2	2.282	-145.3	7.526	0.519	-150.33	3.343%	3.641e-2

Table 8.5: Results of VMC of simple diatomic molecules using hydrogenic orbitals with a Jastrow factor. Here the error (rel) seems to increase, and σ also increases but is relatively low. We notice the α value starts above the charge of a single atom, but decreases to below the charge at C_2. Also we see β first decreasing then increasing.

Type	R	VMC	α	real	rel	σ
H_2	1.4	-1.128	1.190	1.1746	3.967%	2.456e-2
Li_2	5.051	-14.587	2.547	-14.995	2.724%	6.268e-3
Be_2	4.63	-28.423	3.381	-29.339	3.121%	1.380e-2
B_2	3.005	-47.225	4.146	-49.4184	4.438%	2.704e-2
C_2	2.3481	-71.820	4.936	-75.923	5.404%	3.948e-2
N_2	2.068	-103.37	5.671	-109.54	5.635%	4.923e-2
O_2	2.282	-142.07	6.416	-150.321	5.493%	5.939e-2

Table 8.6: Results of VMC of simple diatomic molecules using hydrogenic orbitals with no Jastrow factor. The error (rel) does not follow a pattern, while σ increases but is relatively low. The result is poorer than with hydrogenic orbitals with a Jastrow factor. Also here the alpha value starts above the charge and decreases to below the charge.

8.4.2 Using Gaussian orbitals

Here we look at molecules with Gaussian orbitals. With and without Jastrow factor.

Type	R	VMC	β	real	rel	σ
H_2	1.4	-1.148	0.457	-1.1746	2.264%	1.572e-3
Li_2	5.051	-14.724	1.012	-14.995	1.810%	5.462e-3
Be_2	4.63	-28.686	38.25	-29.339	2.224%	1.258e-2
B_2	3.005	-48.230	1.041	-49.4184	2.405%	2.119e-2
C_2	2.3481	-72.515	6.409	-75.923	4.489%	3.330e-2
N_2	2.068	-105.9	1.627	-109.54	3.325%	4.797e-2
O_2	2.282	-146.4	6.641	-150.321	2.612%	7.520e-2

Table 8.7: Results of VMC of simple diatomic molecules using Gaussian orbitals with a Jastrow factor.The error (rel) does not seem to follow any pattern, while σ increases but is relatively low. The β value seems random. The result (rel) can be compared to the results from the hydrogenic table. 8.5

Type	R	VMC	real	rel	σ
H_2	1.4	-1.117	-1.1746	4.904%	2.399e-3
Li_2	5.051	-14.626	-14.995	2.464%	6.016e-3
Be_2	4.63	-28.686	-29.339	2.224%	1.241e-2
B_2	3.005	-46.912	-49.4184	5.0718%	2.749e-2
C_2	2.3481	-72.432	-75.923	4.5980%	3.186e-2
N_2	2.068	-105.86	-109.54	3.3615%	4.522e-2
O_2	2.282	-146.2	-150.321	2.7452%	8.565e-2

Table 8.8: Results of VMC of simple diatomic molecules using Gaussian orbitals with no Jastrow factor.The error (rel) does not seem to follow any pattern, while σ increases but is relatively low. We see the results (rel) is a bit poorer than with hydrogenic orbitals. (table 8.5) Here we do not have any variables to variate, so the table is from a single run.

We see the results are reasonable, and the error (rel) varies a bit, from 5.635% to 1.413%.We get better results with a Jastrow factor then without. Gaussian orbitals works fine here, and can be compared to the results from hydrogenic orbitals. We note that the standard deviation always gets larger with increasing number of particles.

We notice the α value starts above the charge of a single atom, but decreases to below the charge at C_2. This is seen in table 8.5 and 8.6. The reason for the strange behavior of the α value is not known, but it could be of randomness or how correlations are absorbed. This could also be a effect not taken into account with our trial wavefunction, and the real wavefunction is different or more complex.

8.5 Time used in VMC.

We now measure the time for each cycle. The numbers are given in milliseconds per cycle. For atoms we get table 8.9 and for molecules we get table 8.10.

Type	Hydro Jast	Hydro NoJast	Gauss Jast	Gauss NoJast
He	0.005	0.003	0.008	0.007
Be	0.027	0.018	0.050	0.040
Ne	0.420	0.270	0.900	0.700
Ar	2.500	1.700	5.000	4.200
Kr	22.00	15.00	42.00	36.00

Table 8.9: Milliseconds per simulation: Atoms

Type	Hydro Jast	Hydro NoJast	Gauss Jast	Gauss NoJast
H_2	0.01	0.01	0.02	0.012
Li_2	0.19	0.16	0.37	0.344
Be_2	0.46	0.40	0.90	0.79
B_2	1.00	0.80	2.00	1.60
C_2	1.60	2.00	3.00	3.00
N_2	2.50	3.00	5.00	4.80
O_2	4.00	4.00	8.00	7.60

Table 8.10: Milliseconds per simulation: Molecules

Here we use curve fitting in python to find a curve which matches these points closest possible. So we fit it ageist a $a * n^b$ curve. We get table 8.11.

A value to notice here is the b value. It lies around three and this means that doubling the number of particles causes the simulation to use eight times more time. This is as expected.
A plot, based on the a and b in table 8.11, of this function for Atoms is seen in figure 8.3 and a plot for the same function but for Molecules is found in figure 8.4.

Type	a	b
Mol Hydro Jast	2.92e-4	3.133
Mol Hydro No Jast	1.941e-4	3.141
Mol Gauss Jast	7.15e-4	3.064
Mol Gauss No Jast	5.439e-4	3.097
Dia Hydro Jast	6.01e-3	3.119
Dia Hydro No Jast	9.18e-3	2.939
Dia Gauss Jast	1.013e-2	3.202
Dia Gauss No Jast	8.209e-3	3.283

Table 8.11: Time power function. $t = a * n^b$ milliseconds. Mol: Molecules, Dia: Diatomic molecules, Hydro: Hydrogenic orbitals, Gauss: Gaussain orbitals, Jast: With or without Jastrow factor.

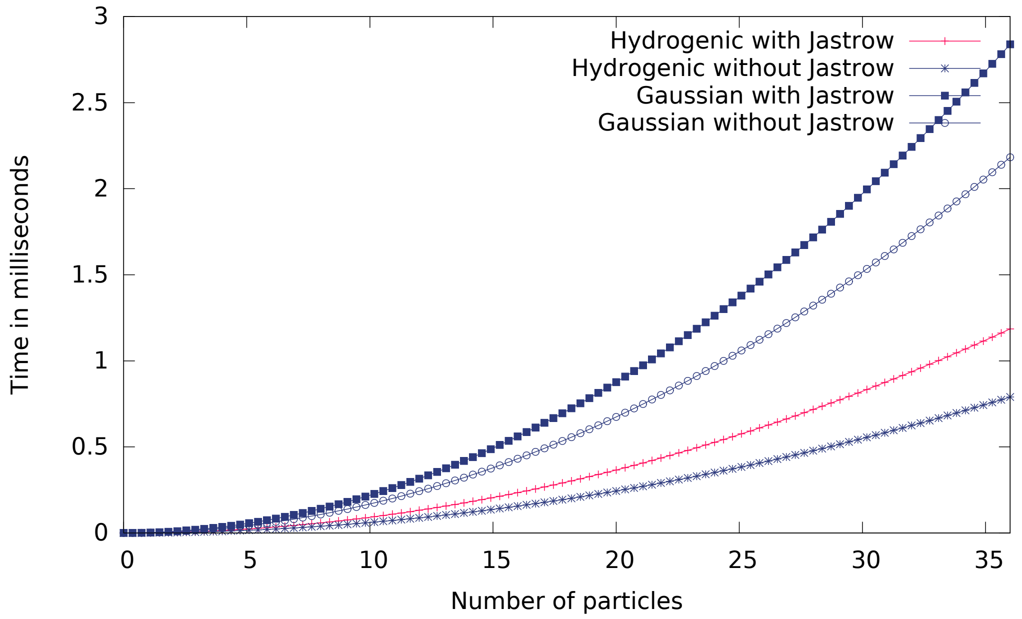


Figure 8.3: In this plot we see time (in milliseconds) per cycle per number of particles for Atoms. The plot is based on the values found in 8.11 a. This makes it easier to compare the efficeincy of the code. It shows here that the fastest is hydrogenic orbitlas without Jastrow factor. Gaussian orbitals with Jastrow is much slower.

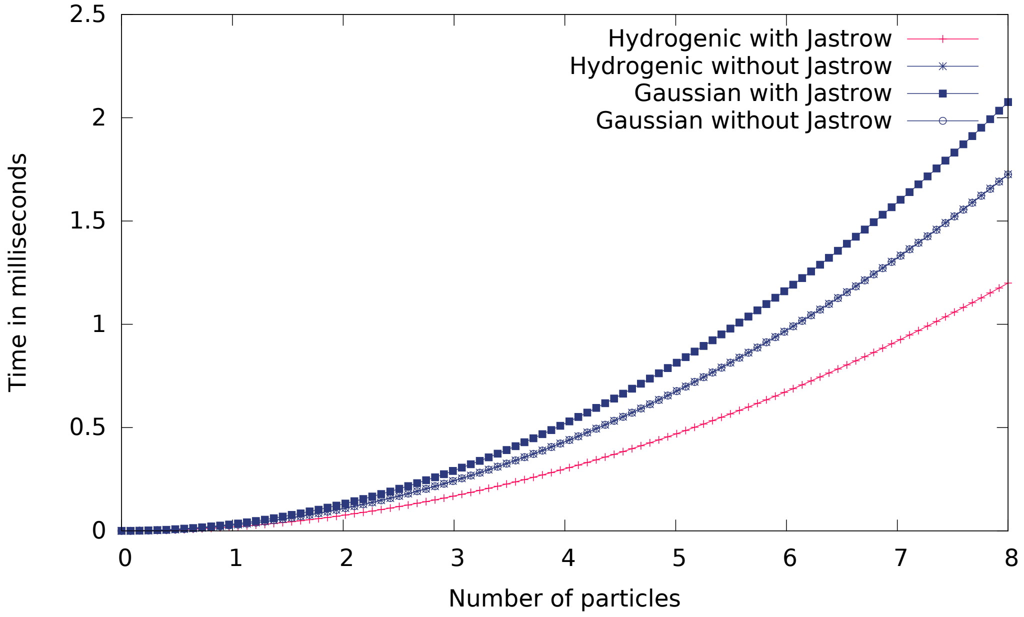


Figure 8.4: In this plot we see time (in milliseconds) per cycle per number of particles for Molecules. The plot is based on the values found in 8.11 a. This makes it easier to compare the efficeincy of the code. Pecurlaly here the hydrogenic orbitals with Jastrow is fastest. Gaussian with Jastrow is also here the slowest.

In the code for this thesis hydrogenic orbitals are faster than Gaussian orbitals. This comes from a slower wavefunction and slower gradients and Laplacians in the Gaussain code, cause that is the only difference in the code, the rest is identical. This time difference outweighs the lack of a square root function in the Gaussain orbitals.

8.6 Conclusion

We can here conclude that both Hydrogenic orbitals and Gaussian orbitals works fine in VMC. At worst we get about %95 of the real value which is not very accurate, but it shows that the equations used are reasonable. For Atoms the run with hydrogenic orbitals with a Jastrow factor the most accurate, but for Molecules there was about no difference.

The β factor behaves strangely. in table 8.1 and 8.5 the value first decreases then increases. In table 8.3 it just increase, while in table 8.7 it seems totally random. This can mean the real wavefunction is more complex than our simple ansatz.

We see in tables 8.1 to 8.8 that the result (see the VMC coulomb) improve better with the introduction of a Jastrow factor.

We see also that the standard deviation increases with number of particles. This is a measure of the errors in the VMC simulation.

Relativistic effects are not taken in consideration in this thesis, and will be a source for error. How large error is difficult to determine as we also have a source of error in equation (3.22). Neither do we know we use a optimal trial wave function.

Chapter 9

Conclusions

9.1 The point of the thesis

One of the aims of this thesis was to develop a general quantum Monte Carlo solver which can simulate a system with a large number of particles, and give us the binding energy of the electrons. Another point was to test the use of Gaussain orbitals instead of the ordinary hydrogen orbitals. Both theese goals has been done sucsessfully in this thesis.

9.2 C++ and Python

The choice of using c++ in this thesis is sound, because python is a scripted language and is therefore slower than c++ and QVMC is a very cpu intensive algorithm. Symbolic python (Sympy) makes Python invaluable in this thesis, since we can make Python do all the mathematicaly heavy tasks (on paper) as finding the derivative and second derivative of a function.

The Armadillo library makes the code easy to read and easy to program, can be slower than if we used ordinary double arrays for the matrices.

The entire project consist of approximately 3000 line of c++ and python code.

9.3 Results

The output of the VMC program made for this thesis is not completley wrong. At most 5.635% of the energy is lost, at best only 0.47% is lost. The usefulness of this thesis is mostly theoretical, but it shows that the theory behind it all, as the Schrödinger equation, the potential used and the VMC method works fine here.

9.4 Effiniency of the code.

We have learned that the hydrogenic orbitals are faster to calculate than the Gaussain orbitals. As a rule we can say that doubeling the number of particles we must multiply the runtime by approximatley 8.

9.5 Further work

The program developed for this thesis only handles light atoms and light diatomic molecules, but can easily be modified to handle heavier atoms and more complex molecules. With the use of more powerful computers we can simulate heavier atoms and molecules.

One optimization one should consider is to store the Slater determinants, the gradient and the Laplacian in tables, so we don't have to calculate them several times. This is not done in this thesis.

Better results could be achieved using other basis sets from the Basis set exchange webpage. In this thesis only STO-3G orbitals are used. Also other potensial could be used which is closer to the real wavefuncion.

In the code we have a variable called timestep that is set to 0.005. This is a adjustable quantity and we might get better results by tweaking it. It is used when making a new move of a particle and in the Greensfunction.

Also the proses of making new Gaussian wavefunction objects could be automatized. In this thesis the work of making new source files for different atoms is done manually.

Part III

Appendix

Appendix A

Programming languages

A.1 Computer programs in general

A computer program is a series of instructions run from top to bottom.

Usually this code is written in plain ascii (standard keyset) but there are some languages which uses graphical methods. C++ and python used in this thesis are written in plain ascii.

Common in all languages are the way we write formulas, with basic instructions as addition, subtraction, multiplication. More advanced functions as square root may vary.

Listing A.1: A simple formula

```

x = y * 2 + 1
```

Here the expression right of the equality sign is calculated and stored in the variable to the left. Only one variable can be at the left side. The compiler does not solve equations in this way.

A.2 C++

The main program is written in c++, which is a compiled general-purpose low-level programming language. C++ is quick and flexible, but lacs things like symbolic computing. We use the library Armadillo which is a linear algebra library. It gives us access to arrays and matrices. We use for example the inverse matrix function in the main program.

A.2.1 Variables

The main type of variables used in this thesis are integers and floating point: int and double. Typically an int is 32 bits while a double is 64 bits.

A 32 bits int can have values: -2147483648 to 2147483647. A 64 bits double can have values from $\pm 10^{-322}$ to $\pm 10^{308}$ We also have booleans which are true or false. All variables must be declared before use. This is done as:

Listing A.2: Initializing variables

```

int i = 0;
double d;
bool b = true;
```

A.2.2 Arrays

One very useful thing in c++ is arrays. These may be integers, doubles, booleans or even objects.
To initialize an array we use

Listing A.3: A for loop

```

double* darr = new double[N];
```

To access the values in an array we use [n] where n is the index to the specific location in the array.

Listing A.4: A for loop

```

for (int i = 0 ; i < N ; i++)
    darr[i] = 0;
```

This code walks trough all elements in the darr array and sets them to zero. Note that the first element is at index 0 and not 1. The last is at $N-1$.

A.2.3 Includes

Before using a library we must include it. This is done as:

Listing A.5: A for loop

```

#include <armadillo>
#include <iostream>
```

This includes Armadillo which is a linear algebra library, and iostream which handles input and output.

A.2.4 Namespace

To simplify the code we use namespaces. This is so we don't have to write arma or std every time we use them.

Listing A.6: A for loop

```

using namespace arma;
using namespace std;
```

A.2.5 Loops

There are several ways of making loops in c++. We can have for loops, do-while and while loops. If you know how many times the loop will run you use a for loop, if you know before the loop is done if the loop shall continue you use a while loop. If you don't know before the loop has run at least one you use a do-while loop.

A for loop might look like

Listing A.7: A for loop

```

int sum = 0;

for(int n = 0 ; n <= N; n++) {
    sum += n;
}
```

The same algorithm with while looks like

Listing A.8: A while loop

```

int sum = 0;
int n = 0;

while (n<=N) {

    sum += n;

    n++
}
```

The same algorithm with do-while looks like

Listing A.9: A do-while loop

```

int sum = 0;
int n = 0;

do {

    sum += n;

    n++
} while (n<=N);
```

These three algorithms can be written as:

$$\sum_{n=0}^{N} n \tag{A.1}$$

A.2.6 If test

An if test makes a conditional test, and runs code if a condition is true or false

Listing A.10: A if test

```

if (a == b) {
    //code for a = b
} else {
    //code for a != b
}
```

A.2.7 Case

if you have several options you can use case as shown here

Listing A.11: Case

```

switch (a) {

    case 0:
        //Code for a = 0
        break;
    case 1:
        //Code for a = 1
        break;
    default:
        //default (a != 0 and a != 1)

}
```

A.2.8 Objects

C++ is an object oriented programming language, which means we make packages of code divided in a natural way. An object is in its header file defined like:

Listing A.12: A Object

```

class AClass: {
private:
    int x,y;
    int AFunction(int AArgument);
public:
    int a,b

    AClass();
    ~AClass();


}
```

Here the functions AClass and ˜AClass are constructors and destructors respectively. The constructor is run every time the object are initialized and the destructor is run when the object is destroyed when it is no more needed. To use this object we use the keyword new, which makes a new instance of the object.

Listing A.13: Initializing a object

```

AClass* new_object = new AClass();
```

Here we use * to specify that we want the memory address of the object. To access this memory address we use ->.

Listing A.14: Using a function

```

    int returnvalue = new_object->AFunction(0);\\
```

A memory address can be 32 or 64 bits and point to a location in memory where the object is stored. Modern machines use 64 bits and a bit older use 32 bits, which limits its memory to approx 4 gig of ram. (minus screen memory and bios ++)

A.2.9 Functions

A function is defined by a return value type, a name and input variables

Listing A.15: An function

```

    int AClass::AFunction(int AArgument) {

        \\Code for function

        return 0;

    }
```

Here the function AFunction returns 0, but it can return any allowed value of any type.

A.2.10 Accessibility levels

Both functions and variables of a class have accessibility levels:

1. Public: Variables and functions are accessible everywhere
2. Private: Variables and functions are only accessible inside the class itself
3. Protected: As in Private but also accessible from subclasses

A.2.11 Armadillo

Armadillo is a linear algebra library. We use it for vectors and matrices. To allocate a matrix with zeroes we write

Listing A.16: Initializing a matrix

```

matrix = zeros( 10,10 );
```

This allocates a 10x10 matrix with only zeroes.
To find the inverse of a matrix we simply write:

Listing A.17: Inverse of matrix

```

mat inverse = inv(matrix);
```

To allocate a vector (row vector) we write:

Listing A.18: Initializing a row vector

```

rowvec vector;
```

To find the distance between two vectors we use the norm function which calculates $d = \sqrt{x^2 + y^2 + z^2}$

Listing A.19: Norm of a vector

```

double d = norm(vector1 - vector2);
```

We can extract a row from a matrix or set a row of a matrix with the row function.

Listing A.20: Matrix row manipulation

```

matrix1.row(i) = matrix2.row(j);
```

This sets row i of $matrix1$ to row j of $martix2$.

A.2.12 MPI

We here use the library MPI (Message passing interface) to run the program on a supercomputer (Abel).

This makes it possible to run the program in parallel.
First we initialize mpi with the command:

```
MPI_Init (&argc, &argv);
```

We need to know how many nodes there are, and which node we are in.

```

  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size (MPI_COMM_WORLD, &num_procs);
```

If we want an array of all results we use the command MPI_Gather which makes an array of all the results.

If we want an average value we use the command MPI_Reduce which can sum the results from several nodes. We must here remember to divide by the number of nodes. (num_procs)

We can now find new values for the free variables and share them over all nodes. For this we use the MPI_Bcast command, which can send an array from one node to all nodes.

Usually we use node 0 as a "master" node, which prints to file and calculates free variables. This is when my_rank = 0.

A.2.13 Output

The program in this thesis uses a text interface. To write to screen we use the cout command as shown here

Listing A.21: Matrix row manipulation

```

cout << "Hello, World!" << endl;
```

endl signals carriage return.

A.3 Python

Python is a very flexible scripting language, with the possibility of symbolic calculations. (Sympy) In this thesis we use python to generate c++ code and to fit a curve for time used in a run.

Python does not use {} as c++ does, instead it uses indentations. This makes the code easy to read.

A.3.1 Variables

Variables in Python are not directly initialized, python figures out what type of variable you use.

A.3.2 Loops

A for loop is often done by a for loop

Listing A.22: A for loop

```

sum = 0

for n in range(N+1):
    sum += n
```

In Python this can be made much simpler:

Listing A.23: A summation

```

print sum(range(N+1))
```

These three algorithms can be written as:

$$\sum_{n=0}^{N} n \tag{A.2}$$

A.3.3 If test

A if test is similar to c++

Listing A.24: A if test

```

if (a == b):
    #code for a = b
else:
    #code for a != b
```

A.3.4 Objects

Classes in python are a bit different.

Listing A.25: defining an object

```

class AClass():
    def __init__(self):
        #Initialization code

    def __del__(self):
        #Destruction code

    def afunction(self,aArgument1, aArgument2):

        #code for funtion

        return 0


    self.x = 0
```

The keyword self refers to the object itself, and must always be the first argument of a python function in an object.

The __*init*__ function is a constructor which is run when the object is created. The __*del*__ function is the destructor, and is run when the object is destroyed.

A.3.5 Exec

Since Python is a parsed language we can generate code on the fly and execute it as ordinary code. To do this we use the exec command.

Listing A.26: Exec

```

exec 'print "Hello, World"'
```

A.3.6 Symbolic Python

If we use the package Sympy we can calculate with symbols. First we must initialize the symbols used

Listing A.27: Initializing symbols

```

x, y , z = symbols('x y z')
```

We can now use these symbols in formulas:

Listing A.28: Making a formula

```

rsqrt = sqrt(x*x + y*y + z*z)
```

We can now for example derivate this function, with respect to any of its variables.

Listing A.29: Diff

```

d = diff(rsqrt,x)
```

A.3.7 C++ code

To generate C++ code we use the ccode command from sympy.printing.

Listing A.30: Generating c++ code

```

  rsqrt_code = ccode(rsqrt)
```

A.3.8 Printing to file.

Writing ascii to file is very simple in python. First we must make an instance of the file object

Listing A.31: Open file

```

  f = open('file.cpp','w')
```

To write to file we use the write command:

Listing A.32: Write to file

```

  f.write("Hello, world")
```

Then we must close the file when done writing.

Listing A.33: Close file

```

  f.close()
```

Bibliography

[1] *Introduction to Quantum Mechanics.* Pearson, 2005.

[2] *Linear algebra and its Applications.* Pearson, 2012.

[3] *An effient sampling algorithm for Variational Monte Carlo.* 2013.

[4] N.Metropolis A.W.Rosenbluth M.N.Rosenbluth A.H.Teller and E. Teller. *Equation of state calculations by fast computing machines.* 1953.

[5] Islen Vallejo Henao. Efficient object-oriantaion for many-body physics problems, 2009.

[6] JÃ¸rgen HÃ¸gberget. *Quantum Monte-Carlo Studies of Generalized Many-body Systems.*

[7] Morten Hjort Jensen. Computational physics. `http://www.uio.no/studier/emner/matnat/fys/FYS3150/h10/undervisningsmateriale/Lecture%20Notes/lectures2010.pdf`.

[8] Jan K. Labanowski. *SIMPLIFIED INTRODUCTION TO AB INITIO BASIS SETS. TERMS AND NOTATION.*

[9] T Petersson and B Hellsing. Detailed derivation of gaussian orbital based matrix elements in electron structure calculations. `http://iopscience.iop.org/article/10.1088/0143-0807/31/1/004`.

[10] S.S. reine. *Monte carlo simulations of atoms.* Master's thesis, Univerity of Oslo, 2004.

[11] J.J Sakurai and Jim Napolitano. *Modern Quantum Mechanics.* Pearson, 2011.

[12] C. David Sherrill. An introduction to hartree-fock molecular orbital theory. `http://vergil.chemistry.gatech.edu/notes/hf-intro/`, 2000.

[13] Leon van Dommelen. Quantum mechanics for engineers. `https://www.eng.fsu.edu/~dommelen/quantum/style_a/contents.html`.