Monte-Carlo Simulations of Atoms

THESIS PRESENTED FOR THE DEGREE OF
CANDIDATUS SCIENTIARUM

Simen S. Reine

DEPARTMENT OF PHYSICS
UNIVERSITY OF OSLO
MARCH 2004

'A journey of a thousand miles starts with one single step...'

An old buddhist saying

Preface

This thesis describes work undertaken between August 2002 and March 2004 in the Nuclear and Energy Physics group at the Department of Physics, University of Oslo, under the supervision of Prof. Morten Hjorth-Jensen.

Simen Sommerfelt Reine
March 2004

Acknowledgments

I dedicate this thesis to my loving wife. Without her everlasting patience, support and belief, this work would never have become what it is today.

I would like to thank my supervisor, Prof. Morten Hjorth-Jensen, for his invaluable guidance, for always having time to spare in a seemingly overbooked schedule, and for making the incomprehensible a joyous delight.

I am indebted to my fellow students whom have given without demanding in return. I am particularly thankful for the contribution to the program code by Mateusz Røstad and to Jon Kristian Nilsen for always providing computer aid. I would express my thanks to Victoria Popsueva and Elise Bergli for sharing both their humor and their theoretical insights. Also I would like to thank Simen Kvaal for making quantum mechanics look easy, and Andreas Sæbjørnsen for our tight collaboration before he went abroad.

I wish to thank my dear friends for their encouragement, for their entertainment and for their support. I hope the reason they have not called these last months is that they are being considerate and not because they have forgotten me.

Finally, I wish to thank my parents for their love and encouragement, without whom I would never have enjoyed so many opportunities.

Contents

Chapter 1

Introduction

The formulation and development of quantum theory in the first half of the 20^{th} century has led to a revolution in our understanding of fundamental physics. Quantum theory has demonstrated surprising accuracy and predictive power, and the importance of quantum theory is unchallenged. The Schrödinger equation, which is the fundamental equation of quantum mechanics, cannot be solved analytically for any but the most trivial of systems. Numerical many-body approaches provide powerful tools for solving this equation.

In electronic structure calculations the treatment of electron-electron interactions is the main source of difficulty. These interactions cannot easily be separated out or treated without approximation. Quantum Monte Carlo (QMC) methods treat electron-electron interactions almost without approximation. The accuracy of the QMC methods enables a great deal of confidence to be placed in the results obtained.

In this thesis the basics of a Variation Monte Carlo (VMC) algorithm are implemented and applied to several atoms. The VMC method forms the basis of the QMC machinery. VMC used together with Diffusion Monte Carlo (DMC) provides a powerful tool for incorporating correlation effects into the many-body wave-function, and by means of Monte Carlo (MC) integration the expectation values of different physical observables can be obtained.

Before formulating the VMC algorithm several concepts must be understood. In chapter 2 the atomic problem is formulated, and then approximated to a form suited for numerical calculations. The solutions to the one-electron hydrogen atom are presented, before we introduce the variational and perturbative methods to the helium atom.

In chapter 3, different numerical approaches are studied in greater detail. Both Hartree-Fock (HF) and Density Functional Theory (DFT) are commonly used methods. Hartree-Fock will be studied thoroughly, both because HF-solutions are used as a basis for our VMC calculations and because understanding the HF method provides useful insights to the other many-body approaches. DFT is particularly interesting for larger system (100

particles or more), and will only be mentioned briefly.

The Configuration Interaction (CI) method is a natural extension of the HF method, and the principles behind it are easy to understand. The principal shortcoming of the CI method is that it does not provide a compact description of the electron correlation, and has a large growth in the number of configurations needed for large systems, see ref. [1]. Among the different many-body methods, Coupled-Cluster (CC) represents the most effective method for atomic systems. Although computationally very fast for small systems, it is practically impossible to carry out for large systems due to the power six (or higher) scaling with the number of particles involved. Møller-Plesset Many-Body Perturbation Theory (MPPT) provides a different route to the solution of the Schrödinger equation allowing us to approach the exact solution in a systematic fashion based on an order-by-order expansion of the wave-function and the energy. For systems where several important configurations are present, the Multiconfiguration Self-Consistent Field method (MCSCF) is well suited, but a problem with this method is that it shows no clear sign of convergence.

A completely different approach to the many-body problem is realized through Monte Carlo integration. Monte Carlo methods make the evaluation of high dimensional integration possible. In the Variational Monte Carlo (VMC) approach *any* trial wave-function may be optimized with respect to either energy or variance minimization. This method therefore allows flexibility beyond the orbital representation. Furthermore, accurate results are obtained for several systems. A particularly interesting extension to the VMC method is to combine VMC with the Diffusion Monte Carlo (DMC) method. DMC yields in principle an exact solution to the many-body problem. In practice, however, the DMC method requires a guiding function, that accurately represents the basic features of the eigenfunction, as input. Commonly, a variational trial wave-function is optimized by VMC, and the optimized function provides the guiding function for DMC calculations. The accuracy of DMC is limited only by this guiding function. Therefore, development of the VMC method should be seen in this context, and together with DMC it provides a highly efficient tool for solving quantum mechanical problems with many particles. Furthermore, the VMC and DMC methods share several similarities, and in developing the VMC method the foundation of the DMC method is established. Nevertheless, VMC is also an efficient tool on its own, and for smaller systems much of the electronic correlations, approximated in for example a HF calculations, are regained.

In chapter 4 the program code developed through this thesis is outlined. The program is coded in C++, and consists of about 7 000 lines. A detailed description of the code has therefore been omitted. Instead we focus on the theoretical principles regarding the numerical optimization of the key building blocks of the program. The main effort of this thesis has been developing the code, and a few examples are provided to help clarify some of the underlying technicalities regarding its implementation.

In chapter 5 results produced by the code are presented. Tests regarding consistency with

known solutions are presented, as well as stability tests regarding wave-function optimization schemes. Such tests are essential in the development of any numerical tool, and have therefore been given special attention. Step-by-step improvement of trial wave-functions are also outlined, and results including electron-electron correlations are presented for several atoms; from helium with its two electrons to argon with a total of eighteen electrons. The results indicate the efficiency of the numerical implementation presented here. The results also indicate that electron-electron correlations become less important as the number of electrons increases. Therefore, inclusion of additional correlation effects is needed in making accurate calculations also for larger systems. A further development is the sophistication of the trial wave-functions.

To be able to effectively optimize more complicated trial wave-functions the efficiency of the VMC algorithm needs further development. In particular, we will study auto-correlation effects which are one of the key limiting factors of the QMC methods. After investigation of these effects we suggest ways to decrease them.

Finally, the insights gained through the development of this thesis will be presented, together with suggestions of further development of the program code.

Note that in this thesis it is assumed that the reader is familiar with the basics of quantum mechanics. For introductory quantum mechanics consult for example refs. [2, 3, 4, 5].

Chapter 2

Atomic Physics

In this chapter the basic principles and difficulties of atomic physics are outlined through investigation of the hydrogen and helium atoms. Applications of quantum mechanics to the atomic problem result in a partial integro-differential equation. This equation cannot be solved analytically except for the special case of the hydrogen atom. The solutions of the hydrogen atom provide useful insights regarding the nature of the atoms, but difficulties arise when we add one or more electrons. This is mainly because the strength of the electron-electron interactions is comparable to the nucleus-electron interaction.

2.1 Basics

2.1.1 The Atomic Problem

The Hamiltonian for an N-electron atomic system consists of two terms

$$\hat{H}(\mathbf{x}) = \hat{T}(\mathbf{x}) + \hat{V}(\mathbf{x}); \tag{2.1}$$

the kinetic and the potential energy operator. Here $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_N\}$ is the spatial and spin degrees of freedom associated with the different particles. The classical kinetic energy

$$T = \frac{\mathbf{P^2}}{2m} + \sum_{j=1}^{N} \frac{\mathbf{p}_j^2}{2m}$$

is transformed to the quantum mechanical kinetic energy operator by operator substitution of the momentum ($p_k \rightarrow -i\hbar\partial/\partial x_k$)

$$\hat{T}(\mathbf{x}) = -\frac{\hbar^2}{2M}\nabla_0^2 - \sum_{i=1}^{N} \frac{\hbar^2}{2m}\nabla_i^2. \tag{2.2}$$

Here the first term is the kinetic energy operator of the nucleus, the second term is the kinetic energy operator of the electrons, M is the mass of the nucleus and m is the electron mass. The potential energy operator is given by

$$\hat{V}(\mathbf{x}) = -\sum_{i=1}^{N} \frac{Ze^2}{(4\pi\epsilon_0)r_i} + \sum_{i=1,i<j}^{N} \frac{e^2}{(4\pi\epsilon_0)r_{ij}}, \tag{2.3}$$

where the r_i's are the electron-nucleus distances and the r_{ij}'s are the inter-electronic distances.

We seek to find controlled and well understood approximations in order to reduce the complexity of the above equations. The *Born-Oppenheimer approximation* is a commonly used approximation, in which the motion of the nucleus is disregarded.

2.1.2 The Born-Oppenheimer Approximation

In a system of interacting electrons and a nucleus there will usually be little momentum transfer between the two types of particles due to their differing masses. The forces between the particles are of similar magnitude due to their similar charge. If one assumes that the momenta of the particles are also similar, the nucleus must have a much smaller velocity than the electrons due to its far greater mass. On the time-scale of nuclear motion, one can therefore consider the electrons to relax to a ground-state given by the Hamiltonian of eqs. (2.1), (2.2) and (2.3) with the nucleus at a fixed location. This separation of the electronic and nuclear degrees of freedom is known as the Born-Oppenheimer approximation.

In the center of mass system the kinetic energy operator reads (ref. [5])

$$\hat{T}(\mathbf{x}) = -\frac{\hbar^2}{2(M+Nm)}\nabla_{CM}^2 - \frac{\hbar^2}{2\mu}\sum_{i=1}^{N}\nabla_i^2 - \frac{\hbar^2}{M}\sum_{i>j}^{N}\nabla_i \cdot \nabla_j, \tag{2.4}$$

while the potential energy operator remains unchanged. Note that the Laplace operators ∇_i^2 now are in the center of mass reference system.

The first term of eq. (2.4) represents the kinetic energy operator of the center of mass. The second term represents the sum of the kinetic energy operators of the N electrons, each of them having their mass m replaced by the reduced mass $\mu = mM/(m+M)$ because of the motion of the nucleus. The nuclear motion is also responsible for the third term, or the *mass polarization* term.

The nucleus consists of protons and neutrons. The proton-electron mass ratio is about 1/1836 and the neutron-electron mass ratio is about 1/1839, so regarding the nucleus as stationary is a natural approximation. Taking the limit $M \to \infty$ in eq. (2.4), the kinetic energy operator reduces to

$$\hat{T} = -\sum_{i=1}^{N}\frac{\hbar^2}{2m}\nabla_i^2 \tag{2.5}$$

The Born-Oppenheimer approximation thus disregards both the kinetic energy of the center of mass as well as the mass polarization term. The effects of the Born-Oppenheimer approximation are quite small and they are also well accounted for. However, this simplified electronic Hamiltonian remains very difficult to solve, and analytical solutions do not exist for general systems with more than one electron. The Born-Oppenheimer approximation will be used for the rest of this thesis.

The first term of eq. (2.3) is the nucleus-electron potential and the second term is the electron-electron potential. The inter-electronic potentials are the main problem in atomic physics. Because of these terms, the Hamiltonian cannot be separated into one-particle parts, and the problem must be solved as a whole. A common approximation is to regard the effects of the electron-electron interactions either as averaged over the domain or by means of introducing a density functional, such as by Hartree-Fock (HF) or Density Functional Theory (DFT). These approaches are actually very efficient, and about 99% or more of the electronic energies are obtained for most HF calculations. Other observables are usually obtained to an accuracy of about $90 - 95\%$ (ref. [1]). The main effort of the advanced numerical procedures is to reduce the errors these approximations induce. These issues will be discussed in detail in chapter 3. But first we simplify the atomic problem further by using atomic units.

2.1.3 Atomic Units

Numerical methods require proper scaling of the system in question. In atomic systems we scale to atomic units by setting $m = e = \hbar = 4\pi\epsilon_0 = 1$, see table 2.1.

Atomic Units

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

Table 2.1: Scaling from SI to atomic units

In this way the atomic problem is simplified to

$$\left[-\sum_{i=1}^{N}\frac{1}{2}\nabla_i^2 - \sum_{i=1}^{N}\frac{Z}{r_i} + \sum_{i<j}^{N}\frac{1}{r_{ij}}\right]\Psi(\mathbf{x}) = E\Psi(\mathbf{x}). \tag{2.6}$$

This is the equation we want to solve in atomic physics. The introduction of atomic units serves two purposes. In addition to making the equation easier to work with because of the neglected units, the atomic units also have an important numerical feature. When solving numerical problems the different quantities involved must have proper scaling; they must be in the same order of magnitude. Failing to do so may result in loss of numerical precision.

2.2 The Hydrogen Atom

The solutions of the hydrogen atom form the basis of our understanding of the many-electron atom. The Hamiltonian of the hydrogen atom reads

$$\hat{H} = -\frac{1}{2}\nabla^2 + V(r), \tag{2.7}$$

where $V(r) = -Z/r$. The nucleus charge Z equals unity for the hydrogen atom, but since the solutions to hydrogen-like atoms are important, we keep it throughout our calculations. In polar coordinates the Laplacian is given by (ref. [6])

$$\nabla^2 = \frac{1}{r^2}\left[\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}\right) + \frac{1}{sin^2\theta}\frac{\partial^2}{\partial\phi} + \frac{1}{sin\theta}\frac{\partial}{\partial\theta}\left(sin\theta\frac{\partial}{\partial\theta}\right)\right]. \tag{2.8}$$

Introduction of this term, combined with the fact that the potential is spherically symmetric, allows us to separate the radial part and the angular part of the equation; $\Psi(r,\theta,\phi) = R(r)\cdot Y(\theta,\phi)$. The solutions of the angular equation are known as the spherical harmonics $Y_{lm_l}(\theta,\phi)$. The first few spherical harmonics are listed in table 2.2.

The spherical harmonics introduce two quantum numbers $l = 0, 1, 2, \ldots$ and $m_l = -l, -(l-1), \ldots, (l-1), l$. These quantum numbers are called the *orbital angular momentum* and the *magnetic quantum number*, respectively. It is worth noticing that the spherical harmonics are independent of the shape of a spherical symmetric potential $V(r)$. The radial and the angular part are interconnected through the separation constants introduced when separating the equations. For the radial wave-function

$$\left[-\frac{1}{2}\frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}\right) + V(r) + \frac{l(l+1)}{r^2}\right]R(r) = ER(r). \tag{2.9}$$

we therefore have a term including the angular momentum l. The first few non-normalized radial solutions of equation (2.9) are listed in table 2.3.

The states

$$\Psi_{nlm_l}(r,\theta,\phi) = R_{nl}(r)\cdot Y_{lm_l}(\theta,\phi), \tag{2.10}$$

now have three quantum numbers n, l, m_l, where the *principal quantum number* can take any positive integer $n = 1, 2, \ldots$. By the introduction of quantum numbers the electron

Spherical Harmonics

$m_l \backslash l$	0	1	2	3
+3				$-\frac{1}{8}(\frac{35}{\pi})^{1/2} sin^3\theta e^{+3i\phi}$
+2			$\frac{1}{4}(\frac{15}{2\pi})^{1/2} sin^2\theta e^{+2i\phi}$	$\frac{1}{4}(\frac{105}{2\pi})^{1/2} cos\theta sin^2\theta e^{+2i\phi}$
+1		$-\frac{1}{2}(\frac{3}{2\pi})^{1/2} sin\theta e^{+i\phi}$	$-\frac{1}{2}(\frac{15}{2\pi})^{1/2} cos\theta sin\theta e^{+i\phi}$	$-\frac{1}{8}(\frac{21}{2\pi})^{1/2}(5cos^2\theta - 1) sin\theta e^{+i\phi}$
0	$\frac{1}{2\pi^{1/2}}$	$\frac{1}{2}(\frac{3}{\pi})^{1/2} cos\theta$	$\frac{1}{4}(\frac{5}{\pi})^{1/2}(3cos^2\theta - 1)$	$\frac{1}{4}(\frac{7}{\pi})^{1/2}(2 - 5sin^2\theta) cos\theta$
-1		$+\frac{1}{2}(\frac{3}{2\pi})^{1/2} sin\theta e^{-i\phi}$	$+\frac{1}{2}(\frac{15}{2\pi})^{1/2} cos\theta sin\theta e^{-i\phi}$	$+\frac{1}{8}(\frac{21}{2\pi})^{1/2}(5cos^2\theta - 1) sin\theta e^{-i\phi}$
-2			$\frac{1}{4}(\frac{15}{2\pi})^{1/2} sin^2\theta e^{-2i\phi}$	$\frac{1}{4}(\frac{105}{2\pi})^{1/2} cos\theta sin^2\theta e^{-2i\phi}$
-3				$+\frac{1}{8}(\frac{35}{\pi})^{1/2} sin^3\theta e^{-3i\phi}$

Table 2.2: Spherical harmonics Y_{lm_l} for the lowest l and m_l values (taken from ref. [4]).

Hydrogen-Like Atomic Radial Functions

$l \backslash n$	1	2	3
0	e^{-Zr}	$(2 - r)e^{-Zr/2}$	$(27 - 18r + 2r^2)e^{-Zr/3}$
1		$re^{-Zr/2}$	$r(6 - r)e^{-Zr/3}$
2			$r^2 e^{-Zr/3}$

Table 2.3: The first few radial functions of the hydrogen-like atoms (taken from ref. [7]).

may only occupy distinct states or *eigenstates*, and the corresponding *eigenenergy* is thus quantized. The eigenenergies

$$E_n = -\frac{1}{2n^2},$$

are independent of the orbital angular momentum and the magnetic quantum number. The eigenstates are therefore *degenerate*; several distinct states share the same energy E_n. For a given n we have a degeneracy both with respect to varying values of l and of m_l. Furthermore, we have yet another degeneracy associated with the two possible values of the electronic spin m_s $(= \pm 1/2)$.

The degeneracy due to the magnetic and spin quantum numbers becomes clear when applying a magnetic field. The magnetic field removes the degeneracy by splitting the energy levels. The degeneracy of different orbital angular momenta of the hydrogen atom is removed in multi-electronic systems. Higher angular momentum lead to higher energy. This result is manifested by the way the atoms are classified in the periodic table. States with different orbital angular momenta are for historical reasons assigned the letters s, p, d, f, etc. where s corresponds to $l = 0$, p to $l = 1$ and so forth. For example $2p$ is assigned to a state with $n = 2$ and $l = 1$. The orbitals are energetically arranged in the order $1s$, $2s$, $2p$, $3s$, $3p$, $4s$, $3d$, $4p$, $5s$, $4d$, $5p$, $6s$, $4f$, $5d$, $6p$, etc., which explains the general trends of the periodic table.

A problem with the spherical harmonics of table 2.2 is that they are complex. The introduction of *solid harmonics*, see ref. [1], allows the use of real orbital wave-functions for a wide range of applications. The complex solid harmonics $\mathcal{Y}_{lm_l}(\mathbf{r})$ are related to the spherical harmonics $Y_{lm_L}(\mathbf{r})$ through

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

By factoring out the leading r-dependency of the radial-function (see for example [2]

)

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

we obtain a relationship similar to that of eq. (2.10), namely

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

For the theoretical development of the *real solid harmonics* see ref. [1]. Here Helgaker *et al* first express the complex solid harmonics, C_{lm_l}, by (complex) Cartesian coordinates, and arrive at the real solid harmonics, S_{lm_l}, through the unitary transformation

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

This transformation will not alter any physical quantities that are degenerate in the subspace consisting of opposite magnetic quantum numbers (the angular momentum l is equal for both these cases). This means for example that the above transformation does not alter the energies, unless an external magnetic field is applied to the system. Henceforth, we will use the solid harmonics, and note that changing the spherical potential beyond the Coulomb potential will not alter the solid harmonics. The lowest-order real solid harmonics are listed in table 2.4.

Real Solid Harmonics

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

Table 2.4: The first-order real solid harmonics $\mathcal{Y}_{lm_l}$ (taken from ref. [4]).

2.3 The Helium Atom

The helium atom cannot be solved analytically. The numerical solutions, however, are in excellent agreement with experiments, see for example ref. [8]. We will not go into the details of such accurate approaches, but rather illustrate how to generate an approximate wave-function through application of perturbative and variational methods.

The Hamiltonian of the helium atom is

$$\hat{H} = -\frac{1}{2}\nabla_1^2 - \frac{1}{2}\nabla_2^2 - \frac{2}{r_1} - \frac{2}{r_2} + \frac{1}{r_{12}}. \tag{2.11}$$

2.3.1 The Perturbative Approach

In the perturbative approach controlled approximations are made so that the initial problem is transformed to an *unperturbed* problem where the solutions are easy to obtain. The controlled approximations are treated as *perturbations* of the unperturbed problem and added into the system by including higher and higher corrections of the perturbation. A requirement of the perturbative approach is that the perturbations are small compared to the unperturbed values. Straightforward perturbation of the helium atom is acquired if we first disregard the electron-electron repulsion, and then add it as a perturbative correction.

Without the inter-electronic repulsion we get a separable Hamiltonian

$$\hat{H} = -\frac{1}{2}\nabla_1^2 - \frac{1}{2}\nabla_2^2 - \frac{2}{r_1} - \frac{2}{r_2} = \hat{h}_1 + \hat{h}_2,$$

and we may solve the two one-particle equations independently. The unperturbed wavefunction becomes the product of two hydrogen atom solutions (given by equation (2.10))

$$\Psi^{(0)}_{n_1 l_1 m_{l,1} n_2 l_2 m_{l,2}}(r_1, \theta_1, \phi_1, r_2, \theta_2, \phi_2) = \Psi_{n_1 l_1 m_{l,1}}(r_1, \theta_1, \phi_1)\Psi_{n_2 l_2 m_{l,2}}(r_2, \theta_2, \phi_2)$$

with the nucleus charge $Z = 1$ replaced by $Z = 2$. This gives the energy

$$E^{(0)}_{n_1 n_2} = -2\left(\frac{1}{n_1^2} + \frac{1}{n_2^2}\right), \tag{2.12}$$

where the superscript (0) indicates the unperturbed energy. For the ground state we define the product

$$\Psi_{1,0,0}(r_1, \theta_1, \phi_1)\Psi_{1,0,0}(r_2, \theta_2, \phi_2) \equiv \Psi_{1s}(1)\Psi_{1s}(2),$$

The first order correction to the energy is then

$$J \equiv E^{(1)}_{n_1 n_2} - E^{(0)}_{n_1 n_2} = \int |\Psi_{1s}(1)|^2 \frac{1}{r_{12}} |\Psi_{1s}(2)|^2 d\tau_1 d\tau_2, \tag{2.13}$$

which is called the *Coulomb integral* and often denoted by J. The Coulomb integral is commonly encountered in the approximative methods of many-body quantum mechanics, and has an easy interpretation. The term $|\Psi_{1s}(1)|^2 d\tau_1$ is the probability of finding electron 1 in the volume element $d\tau_1$, and when multiplied with the charge -1 (in atomic units) it represents the *charge density* of that region. Similarly $-|\Psi_{1s}(2)|^2 d\tau_2$ is the charge density of electron 2 in the volume element $d\tau_2$. The Coulomb integral of eq. (2.13) may therefore be interpreted as the averaged contribution of the Coulomb repulsion between the two electrons. The value of the Coulomb integral is $J = 1.25$ according to ref. [4]. This gives a first order approximation to the ground state energy

$$E^{(1)}_0 = -2 - 2 + 1.25 = -2.75.$$

This result is not in perfect agreement with the experimental value $E_0 = -2.9037$. However, it is a clear indication that we are on the right track. One of the reasons for the disagreement is that the perturbation is not at all small, so first-order perturbation theory cannot be expected to lead to a reliable result.

2.3.2 The Variational Approach

A different way of solving the helium atom is the use of a *variational* approach. Here we start out by guessing a parametrized form of a *trial wave-function* Ψ_α, where $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_M)$ denotes the set of variation parameters. Then we optimize these parameters in accordance with the *variational principle*; the energy expectation value of a variational wave-function provides an upper bound to the true ground state energy

$$\frac{\int \Psi_\alpha^* \hat{H} \Psi_\alpha d\tau}{\int |\Psi_\alpha|^2 d\tau} \geq E_0.$$

The variational principle of quantum mechanics may be derived by expanding a normalized trial wave-function, Ψ_α, in terms of the exact orthonormal eigenstates $\{\psi_i\}$ of the Hamiltonian

$$\psi_\alpha = \sum_{i=0}^{\infty} c_i \psi_i,$$

where the expansion coefficients c_i are normalized

$$\sum_{i=0}^{\infty} |c_i|^2 = 1.$$

The expectation of the many-body Hamiltonian $\hat{H}$ is then evaluated as

$$\langle E_\alpha \rangle = \sum_{i=0}^{\infty} |c_i|^2 \epsilon_i,$$

where ϵ_i and ψ_i fulfills the stationary Schrödinger equation

$$\hat{H}\psi_i = \epsilon_i \psi_i.$$

The expectation value of the trial energy must therefore be greater than or equal to the true ground state energy

$$\langle E_\alpha \rangle \geq \langle E_0 \rangle = \epsilon_0,$$

as $\epsilon_i \geq \epsilon_0$. The variational energy computed using ψ_α thus provides an upper bound for the true ground state energy. Therefore, our strategy is to search for the variational parameters that give us the lowest variational energy.

The difficulties in the variational method are to find a good variational wave-function, to evaluate the energy expectation value and to find the energy minimum in parameter space. There are several ways to generate the trial wave-function, and we will return to some of these in chapter 3. One simple approach is to start with a product of two variational hydrogen $1s$ solutions

$$\psi_\alpha = e^{\alpha r_1} e^{\alpha r_2} = e^{\alpha(r_1+r_2)}.$$

We then need to minimize the expectation value

$$\langle E_\alpha \rangle = \frac{\int e^{\alpha(r_1+r_2)} \hat{H} e^{\alpha(r_1+r_2)} d\tau_1 d\tau_2}{\int e^{2\alpha(r_1+r_2)} d\tau_1 d\tau_2},$$

with respect to the parameter α.[1] This minimization is not trivial. The integration domain is the six-dimensional configuration space. We start by a transformation to polar coordinates. For the ground state there are no angular dependencies $\partial\psi_\alpha/\partial\phi = \partial\psi_\alpha/\partial\theta = 0$, so these terms may be removed altogether from the Hamiltonian. The angular terms of the numerator are therefore cancelled by the equal angular terms of the determinator. We are left with the two-dimensional integral

$$\langle E_\alpha \rangle = \frac{\int_0^\infty \int_0^\infty e^{\alpha(r_1+r_2)} \hat{H}_r e^{\alpha(r_1+r_2)} r_1^2 r_2^2 dr_1 dr_2}{\int_0^\infty \int_0^\infty e^{2\alpha(r_1+r_2)} r_1^2 r_2^2 dr_1 dr_2}, \tag{2.14}$$

where the radial Hamiltonian $\hat{H}_r$ is obtained from eqs. (2.8) and (2.11)

$$\hat{H}_r = -\frac{1}{2r_1^2}\frac{\partial}{\partial r_1}\left(r_1^2 \frac{\partial}{\partial r_1}\right) - \frac{1}{2r_2^2}\frac{\partial}{\partial r_2}\left(r_2^2 \frac{\partial}{\partial r_2}\right) - \frac{2}{r_1} - \frac{2}{r_2} + \frac{1}{r_{12}}.$$

Eq. (2.14) can be reduced to the following, see **??**,

!!! Referanse her !!!

$$\langle E_\alpha \rangle = \alpha^2 - \frac{27}{8}\alpha, \tag{2.15}$$

which has a minima for $\alpha = 27/16 = 1.6875$, namely $\langle E_{1.6875} \rangle = -2.84766$.

Compared to the perturbative approach we have gained some, but not all of the correlation. However, adding another term to the trial wave-function

$$\psi_{\alpha,\beta} = e^{-\alpha(r_1+r_2)} exp\left\{\frac{r_{12}}{2(1+\beta r_{12})}\right\}$$

[1] Notice that by setting $\alpha = 2$ we reproduce the perturbative result.

and optimizing (see table 5.3) we arrive at $E_{\alpha,\beta} = -2.8901 \pm 0.0003$. As can be seen by comparing these results with the experimental value -2.9037, the addition of the simple electron-electron exponent is able to regain approximately 78% of the correlation compared to the first hydrogenic trial wave-function.

Variational calculations depend crucially on the form of the trial wave-function used. By selecting trial wave-functions on physically motivated grounds, accurate wave-functions may be obtained. Commonly, wave-functions obtained from a Hartree-Fock or similar calculations are used. Then additional parameters are added, building in additional physics such as known limits and derivatives of the many-body wave-function. The additional variational freedom is then exploited to further optimize the wave-function.

2.4 Beyond the Helium Atom

Before starting the description of the most common methods used to solve the many-body problem we need to adress some elementary theory regarding this problem. First, we must establish some rules regarding the construction of physically reliable wave-functions for systems with more than one electron.

The *Pauli principle* was recognized by Wolfgang Pauli (ref. [4]):

The Pauli Principle *The total wave-function must be antisymmetric under the interchange of any pair of identical fermions and symmetric under the interchange of any pair of identical bosons.*

A result of the Pauli principle is the so-called *Pauli exclusion principle*:

The Pauli Exclusion Principle *No two electrons can occupy the same state.*

Overall wave-functions that satisfy the Pauli principle are often written as *Slater Determinants.*

2.4.1 The Slater Determinant

Again we turn our attention to the helium atom. It was assumed that the two electrons were both in the $1s$ state. This fulfills the Pauli exclusion principle as the two electrons in the ground state have different intrinsic spin. However, the wave-functions we used in both the perturbative and variational approach were not antisymmetric with respect to interchange of the different electrons. This is not totally true as we only included the spatial part of the wave-function. For the helium ground state the spatial part of the wave-function is symmetric and the spin part is anti-symmetric. The product is therefore anti-symmetric as well. The Slater-determinant consists of single-particle *spin-orbitals*;

joint spin-space states of the electrons

$$\Psi_{1s}^{\uparrow}(1) = \Psi_{1s}(1) \uparrow (1),$$

and similarly

$$\Psi_{1s}^{\downarrow}(2) = \Psi_{1s}(2) \downarrow (2).$$

Here the two spin functions are given by

$$\uparrow (I) = \begin{cases} 1 & \text{if } m_s(I) = \frac{1}{2} \\ 0 & \text{if } m_s(I) = -\frac{1}{2} \end{cases},$$

and

$$\downarrow (I) = \begin{cases} 0 & \text{if } m_s(I) = \frac{1}{2} \\ 1 & \text{,if } m_s(I) = -\frac{1}{2} \end{cases}, \tag{2.16}$$

with $I = 1, 2$.
The ground state can then be expressed by the following determinant

$$\Psi(1,2) = \frac{1}{\sqrt{(2)}} \begin{vmatrix} \Psi_{1s}(1) \uparrow (1) & \Psi_{1s}(2) \uparrow (2) \\ \Psi_{1s}(1) \downarrow (1) & \Psi_{1s}(2) \downarrow (2) \end{vmatrix}.$$

This is an example of a *Slater determinant*. This determinant is antisymmetric since particle interchange is identical to an interchange of the two columns. For the ground state the spatial wave-function is symmetric. Therefore we simply get

$$\Psi(1,2) = \Psi_{1s}(1)\Psi_{1s}(2) \left[\uparrow (1) \downarrow (2) - \uparrow (2) \downarrow (1)\right].$$

The spin part of the wave-function is here anti-symmetric. This has no effect when calculating physical observables because the sign of the wave-function is squared in all expectation values.

The general form of a Slater determinant composed of n orthonormal orbitals $\{\phi_i\}$ is

$$\Psi = \frac{1}{\sqrt{N!}} \begin{vmatrix} \phi_1(1) & \phi_1(2) & \dots & \phi_1(N) \\ \phi_2(1) & \phi_2(2) & \dots & \phi_2(N) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_N(1) & \phi_N(2) & \dots & \phi_N(N) \end{vmatrix}. \tag{2.17}$$

The introduction of the Slater determinant is very important for treatment of many-body systems, and in this thesis it is the principal building block of each variational wave-function used. As long as we express the wave-function in terms of either one Slater determinant or a linear combination of several Slater determinants, the Pauli principle is

satisfied. When constructing many-electron wave-functions this picture provides an easy way to include many of the physical features. One problem with the Slater matrix is that it is computationally demanding. Limiting the number of calculations will be one of the most important issues concerning the implementation of the Slater determinant. This will be discussed in detail in chapter 4.

Chapter 3

Numerical Approaches to the Atomic Problem

The recent development in computer technology has made a revolution in our ability to model the nature around us. The making of better numerical procedures is currently a hot topic within the scientific world, and will remain a great challenge also in the years to come.

In this chapter we will outline approaches for solving the atomic problem introduced in the previous chapter. We want to find the eigenstates and eigenenergies satisfying the Born-Oppenheimer approximated Schrödinger equation

$$\left[-\sum_{i=1}^{N}\frac{1}{2}\nabla_i^2-\sum_{i=1}^{N}\frac{Z}{r_i}+\sum_{i<j}^{N}\frac{1}{r_{ij}}\right]\Psi(\mathbf{x})=E\Psi(\mathbf{x}) \tag{3.1}$$

for $N \geq 2$. With more than one electron present in eq. (3.1) we cannot find an analytical solution and must resort to numerical efforts. In this chapter we will examine the theory of several numerical methods, commonly applied to the atomic problem.

3.1 Hartree-Fock and Density Functional Theory

3.1.1 Hartree-Fock Theory

Hartree-Fock theory [1, 4, 5] is one of the simplest approximate theories for solving the many-body Hamiltonian. It is based on a simple approximation to the true many-body wave-function; that the wave-function is given by a single Slater determinant of N orthonormal orbitals.

$$\Psi = \frac{1}{\sqrt{N!}} \begin{vmatrix} \psi_1(\mathbf{x_1}) & \psi_1(\mathbf{x_2}) & \dots & \psi_1(\mathbf{x_N}) \\ \psi_2(\mathbf{x_1}) & \psi_2(\mathbf{x_2}) & \dots & \psi_2(\mathbf{x_N}) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_N(\mathbf{x_1}) & \psi_N(\mathbf{x_2}) & \dots & \psi_N(\mathbf{x_N}) \end{vmatrix}. \tag{3.2}$$

Here the variables $\mathbf{x_i}$ include the coordinates of spin and space. The wave-function is antisymmetric with respect to an interchange of any two electrons, as required by the Pauli principle

$$\Psi(\mathbf{x_1}, \mathbf{x_2}, \dots, \mathbf{x_i}, \dots, \mathbf{x_j}, \dots \mathbf{x_N}) = -\Psi(\mathbf{x_1}, \mathbf{x_2}, \dots, \mathbf{x_j}, \dots, \mathbf{x_i}, \dots \mathbf{x_N}).$$

We rewrite the Hamiltonian

$$\hat{H} = -\sum_{i=1}^{N} \frac{1}{2}\nabla_i^2 - \sum_{i=1}^{N} \frac{Z}{r_i} + \sum_{i<j}^{N} \frac{1}{r_{ij}}$$

as

$$\hat{H} = \hat{H}_1 + \hat{H}_2 = \sum_{i=1}^{N} \hat{h}_i + \sum_{i<j=1}^{N} \frac{1}{r_{ij}}, \tag{3.3}$$

where

$$\hat{h}_i = -\frac{1}{2}\nabla_i^2 - \frac{Z}{r_i}. \tag{3.4}$$

The first term of eq. (3.3), H_1, is the sum of the N identical *one-body* Hamiltonians $\hat{h}_i$. Each individual Hamiltonian $\hat{h}_i$ contains the kinetic energy operator of an electron and its potential energy due to the attraction of the nucleus. The second term, H_2, is the sum of the $N(N-1)/2$ *two-body* interactions between each pair of electrons. Let us denote the ground state energy by E_0. According to the variational principle we have

$$E_0 \leq E[\Phi] = \int \Phi^* \hat{H} \Phi d\tau$$

where Φ is a trial function which we assume to be normalized

$$\int \Phi^* \Phi d\tau = 1.$$

In the Hartree-Fock method the trial function is the Slater determinant of eq. (3.2) which may be recast as

$$\Psi = \frac{1}{\sqrt{N!}} \sum_P (-)^P P \psi_1(\mathbf{x_1}) \psi_2(\mathbf{x_2}) \dots \psi_N(\mathbf{x_N}) = \sqrt{N!} \mathcal{A} \Phi_H, \tag{3.5}$$

with the *anti-symmetry* operator given by a summation of the different permutations

$$\mathcal{A} = \frac{1}{N!}\sum_{P}(-)^{P}P, \tag{3.6}$$

and where the Hartree-function is given by

$$\Phi_H = \psi_1(\mathbf{x_1})\psi_2(\mathbf{x_2})\ldots\psi_N(\mathbf{x_N}).$$

Both $\hat{H}_1$ and $\hat{H}_2$ are invariant under electron permutations, and hence commute with $\mathcal{A}$

$$[H_1, \mathcal{A}] = [H_2, \mathcal{A}] = 0. \tag{3.7}$$

Furthermore, $\mathcal{A}$ satisfies

$$\mathcal{A}^2 = \mathcal{A}, \tag{3.8}$$

since every (either positive or negative) permutation of the Slater determinant reproduces it. The expectation value of $\hat{H}_1$

$$\int \Phi^* \hat{H}_1 \Phi d\tau = N! \int \Phi_H^* \mathcal{A}\hat{H}_1 \mathcal{A}\Phi_H d\tau$$

is readily reduced to

$$\int \Phi^* \hat{H}_1 \Phi d\tau = N! \int \Phi_H^* \hat{H}_1 \mathcal{A}\Phi_H d\tau,$$

where we have used eqs. (3.7) and (3.8). The next step is to replace the anti-symmetry operator by its definition eq. (3.5) and to replace $\hat{H}_1$ with the sum of one-body operators

$$\int \Phi^* \hat{H}_1 \Phi d\tau = \sum_{i=1}^{N}\sum_{P}(-)^{P}\int \Phi_H^* \hat{h}_i P\Phi_H d\tau.$$

The integral vanishes if two or more electrons are permuted in only one of the Hartree-functions Φ_H because the individual orbitals are orthogonal. We obtain then

$$\int \Phi^* \hat{H}_1 \Phi d\tau = \sum_{i=1}^{N}\int \Phi_H^* \hat{h}_i \Phi_H d\tau.$$

Orthogonality allows us to further simplify the integral, and we arrive at the following expression for the expectation values of the sum of one-body Hamiltonians

$$\int \Phi^* \hat{H}_1 \Phi d\tau = \sum_{\mu=1}^{N}\int \psi_\mu^*(\mathbf{x_i})\hat{h}_i \psi_\mu d\mathbf{x_i}(\mathbf{x_i}). \tag{3.9}$$

The expectation value of the two-body Hamiltonian may be obtained in a similar manner. We have

$$\int \Phi^* \hat{H}_2 \Phi d\tau = N! \int \Phi_H^* \mathcal{A} \hat{H}_2 \mathcal{A} \Phi_H d\tau,$$

which reduces to

$$\int \Phi^* \hat{H}_2 \Phi d\tau = \sum_{i \leq j=1}^{N} \sum_{P} (-)^P \int \Phi_H^* \frac{1}{r_{ij}} P \Phi_H d\tau,$$

by following the same arguments as for the one-body Hamiltonian. Because of the inter-electronic distances permutations of two electrons no longer vanish, and we get

$$\int \Phi^* \hat{H}_2 \Phi d\tau = \sum_{i<j=1}^{N} \int \Phi_H^* \frac{1}{r_{ij}} (1 - P_{ij}) \Phi_H d\tau.$$

where P_{ij} is the permutation operator that interchanges electrons i and j. Again we use the assumption that the orbitals are orthogonal, and obtain

$$\begin{aligned} \int \Phi^* \hat{H}_2 \Phi d\tau = \frac{1}{2} \sum_{\mu=1}^{N} \sum_{\nu=1}^{N} \Bigg[& \int \psi_\mu^*(\mathbf{x_i}) \psi_\nu^*(\mathbf{x_j}) \frac{1}{r_{ij}} \psi_\mu(\mathbf{x_i}) \psi_\nu(\mathbf{x_j}) d\mathbf{x_i} \mathbf{x_j} \\ & - \int \psi_\mu^*(\mathbf{x_i}) \psi_\nu^*(\mathbf{x_j}) \frac{1}{r_{ij}} \psi_\nu(\mathbf{x_i}) \psi_\mu(\mathbf{x_j}) d\mathbf{x_i} \mathbf{x_j} \Bigg]. \end{aligned} \tag{3.10}$$

Here the fraction 1/2 is introduced because we now run over all pairs twice. Combining eq. (3.9) and (3.10) we have the functional

$$E[\Phi] = \sum_{\mu=1}^{N} \int \psi_\mu^* \hat{h_i} \psi_\mu d\mathbf{x_i} + \frac{1}{2} \sum_{\mu=1}^{N} \sum_{\nu=1}^{N} \left[\int \psi_\mu^* \psi_\nu^* \frac{1}{r_{ij}} \psi_\mu \psi_\nu d(\mathbf{x_i} \mathbf{x_j}) - \int \psi_\mu^* \psi_\nu^* \frac{1}{r_{ij}} \psi_\nu \psi_\mu d(\mathbf{x_i} \mathbf{x_j}) \right]. \tag{3.11}$$

Having obtained the functional $E[\Phi]$, we now proceed to the second step of the calculation. The functional is stationary with respect to variations of the spin orbitals, subject to the N^2 conditions imposed by the orthogonality requirements,

$$\int \psi_\mu(\mathbf{x}) \psi_\nu(\mathbf{x}) d\mathbf{x} = \delta_{\mu,\nu}$$

To satisfy these conditions, we introduce N^2 Lagrange multipliers which we denote by $\epsilon_{\mu\nu}$. The variational equation then reads

$$\delta E - \sum_{\mu=1}^{N} \sum_{\nu=1}^{N} \epsilon_{\mu\nu} \delta \int \psi_\mu^* \psi_\nu = 0. \tag{3.12}$$

For the orthogonal ψ_μ this reduces to

$$\delta E - \sum_{\mu=1}^{N} \epsilon_\mu \delta \int \psi_\mu^* \psi_\mu = 0. \tag{3.13}$$

Variation with respect to the spin-orbitals ψ_μ yields

$$\sum_{\mu=1}^{N} \int \delta\psi_\mu^* \hat{h}_i \psi_\mu d\mathbf{x_i} + \frac{1}{2} \sum_{\mu=1}^{N} \sum_{\nu=1}^{N} \left[\int \delta\psi_\mu^* \psi_\nu^* \frac{1}{r_{ij}} \psi_\mu \psi_\nu d(\mathbf{x_i x_j}) - \int \delta\psi_\mu^* \psi_\nu^* \frac{1}{r_{ij}} \psi_\nu \psi_\mu d(\mathbf{x_i x_j}) \right]$$
$$+ \sum_{\mu=1}^{N} \int \psi_\mu^* \hat{h}_i \delta\psi_\mu d\mathbf{x_i} + \frac{1}{2} \sum_{\mu=1}^{N} \sum_{\nu=1}^{N} \left[\int \psi_\mu^* \psi_\nu^* \frac{1}{r_{ij}} \delta\psi_\mu \psi_\nu d(\mathbf{x_i x_j}) - \int \psi_\mu^* \psi_\nu^* \frac{1}{r_{ij}} \psi_\nu \delta\psi_\mu d(\mathbf{x_i x_j}) \right]$$
$$- \sum_{\mu=1}^{N} E_\mu \int \delta\psi_\mu^* \psi_\mu d\mathbf{x_i} - \sum_{\mu=1}^{N} E_\mu \int \psi_\mu^* \delta\psi_\mu d\mathbf{x_i} = 0.$$

Although the variations $\delta\psi$ and $\delta\psi^*$ are not independent, they may in fact be treated as such, so that the terms dependent on either $\delta\psi$ and $\delta\psi^*$ individually may be set equal to zero. To see this, simply replace the arbitrary variation $\delta\psi$ by $i\delta\psi$, so that $\delta\psi^*$ is replaced by $-i\delta\psi^*$, and combine the two equations. We thus arrive at the Hartree-Fock equations

$$\left[-\frac{1}{2}\nabla_i^2 - \frac{Z}{r_i} + \sum_{\nu=1}^{N} \int \psi_\nu^*(\mathbf{x_j}) \frac{1}{r_{ij}} \psi_\nu(\mathbf{x_j}) d\mathbf{x_j} \right] \psi_\mu(\mathbf{x_i})$$
$$- \left[\sum_{\nu=1}^{N} \int \psi_\nu^*(\mathbf{x_j}) \frac{1}{r_{ij}} \psi_\mu(\mathbf{x_j}) d\mathbf{x_j} \right] \psi_\nu(\mathbf{x_i}) = \epsilon_\mu \psi_\mu(\mathbf{x_i}). \tag{3.14}$$

Notice that the integration $\int d\mathbf{x_j}$ implies an integration over the spatial coordinates $\mathbf{r_j}$ and a summation over the spin-coordinate of electron j.

The two first terms are the one-body kinetic energy and the electron-nucleus potential. The third or *direct* term is the averaged electronic repulsion of the other electrons. This term is identical to the Coulomb integral introduced in the simple perturbative approach to the helium atom. As written, the term includes the 'self-interaction' of electrons when $i = j$. The self-interaction is cancelled in the fourth term, or the *exchange* term. The exchange term results from our inclusion of the Pauli principle and the assumed determinantal form of the wave-function. The effect of exchange is for electrons of like-spin to avoid each other. A theoretically convenient form of the Hartree-Fock equation is to regard the direct and exchange operator defined through

$$V_\mu^d(\mathbf{x_i}) = \int \psi_\mu^*(\mathbf{x_j}) \frac{1}{r_{ij}} \psi_\mu(\mathbf{x_j}) d\mathbf{x_j}$$

and

$$V_\mu^{ex}(\mathbf{x_i})g(\mathbf{x_i}) = \left(\int \psi_\mu^*(\mathbf{x_j}) \frac{1}{r_{ij}} g(\mathbf{x_j}) d\mathbf{x_j} \right) \psi_\mu(\mathbf{x_i}),$$

respectively. The function $g(\mathbf{x_i})$ is an arbitrary function, and by the substitution $g(\mathbf{x_i}) = \psi_\nu(\mathbf{x_i})$ we get

$$V_\mu^{ex}(\mathbf{x_i})\psi_\nu(\mathbf{x_i}) = \left(\int \psi_\mu^*(\mathbf{x_j}) \frac{1}{r_{ij}} \psi_\nu(\mathbf{x_j}) d\mathbf{x_j} \right) \psi_\mu(\mathbf{x_i}).$$

We may then rewrite the Hartree-Fock equations as

$$H_i^{HF}\psi_\nu(\mathbf{x_j}) = \epsilon_\nu \psi_\nu(\mathbf{x_i}), \tag{3.15}$$

with

$$H_i^{HF} = h_i + \sum_{\mu=1}^{N} V_\mu^d(\mathbf{x_i}) - \sum_{\mu=1}^{N} V_\mu^{ex}(\mathbf{x_i}), \tag{3.16}$$

and where h_i is defined by equation (3.4).

3.1.2 Solving the Hartree-Fock Equations

Both the direct and exchange terms of equation (3.14) depend on the orbitals we want to find. We need the solutions to generate the two terms. This problem is usually solved by iteration. Start by some initial guess for the orbitals. Then the direct and exchange terms are computed using these fixed guesses, and the orbitals are solved for one at a time. These solutions are used to recalculate the direct and the exchange terms, and the HF equations are solved to find a new set of spin-orbitals. This cycle is carried out until the solutions form a *self-consistent field* (SCF). Convergence problems are sometimes encountered but they are usually not a problem in most calculations.

The Fock operator of eq. (3.14) depends on the N occupied spin-orbitals. Once these orbitals have been determined, the Fock operator can be treated as a well-defined hermitian operator. As for any hermitian operator there is an infinite number of eigenfunctions of the Fock operator. In other words, there is an infinite number of spin-orbitals ψ_μ, each having an eigenvalue E_μ. In practice a finite number $k \geq N$ of spin orbitals are used.

The N lowest energy spin-orbitals are called the *occupied orbitals*, and the remaining $k-N$ orbitals are called the *virtual orbitals*. The Slater determinant composed of the occupied orbitals is the Hartree-Fock ground-state wave-function of the system, and we shall denote it Φ_0.

For closed shell atoms it is natural to consider the spin-orbitals as paired. For example, two $1s$ orbitals with different spin have the same spatial wave-function, but orthogonal spin

functions. For open-shell atoms two procedures are commonly used; the *restricted Hartree-Fock* (RHF) and *unrestricted Hartree-Fock* (UHF). In RHF all the electrons except those occupying open-shell orbitals are forced to occupy doubly occupied spatial orbitals, while in UHF all orbitals are treated independently. The UHF, of course, yields a lower variational energy than the RHF formalism. One disadvantage of the UHF over the RHF, is that whereas the RHF wave function is an eigenfunction of S^2, the UHF function is not; that is, the total spin angular momentum is not a well-defined quantity for a UHL wave-function. Here we limit our attention to closed shell RHF's, and show how the coupled HF equations may be turned into a matrix problem by expressing the spin-orbitals using known sets of basis functions.

We expand the orbitals ψ_μ by M basis-functions θ_j,

$$\psi_\mu(\mathbf{x}) = \sum_{j=1}^{M} c_{j,\mu}\theta_j(\mathbf{x}). \tag{3.17}$$

where $c_{j,\mu}$ are the unknown coefficients. From a set of M basis function, we can obtain M independent orbital wave-functions, and the problem of calculating the wave-functions has now been transformed to one of computing the coefficients $c_{j,\mu}$. We insert the expanded wave-function of eq. (3.17) into the HF equation, eq. (3.15), and get

$$H_i^{HF}\sum_{j=1}^{M} c_{j,\mu}\theta_j(\mathbf{x}) = \epsilon_\mu \sum_{j=1}^{M} c_{j,\mu}\theta_j(\mathbf{x}).$$

Multiplication of both sides with $\theta_i(\mathbf{x})$ and integration over $d\mathbf{x}$ yields

$$\sum_{j=1}^{M} c_{j,\mu}\int \theta_i(\mathbf{x})H_j^{HF}\theta_j(\mathbf{x})d\mathbf{x} = \epsilon_\mu \sum_{j=1}^{M} c_{j,\mu}\int \theta_i(\mathbf{x})\theta_j(\mathbf{x})d\mathbf{x}.$$

The set of Hartree-Fock equations have now been turned into a matrix equation. This is seen by defining the *overlap matrix*, $\mathbf{S}$, with elements

$$S_{ij} = \int \theta_i(\mathbf{x})\theta_j(\mathbf{x})d\mathbf{x},$$

and the *Fock matrix*, $\mathbf{F}$, with elements

$$F_{ij} = \int \theta_i(\mathbf{x})H_j^{HF}\theta_j(\mathbf{x})d\mathbf{x}.$$

Further, defining the matrix $\mathbf{c}$ corresponding to the $c_{i,\mu}$ and the diagonal matrix $\mathbf{E}$ of the orbital energies ϵ_μ we get the matrix equation

$$\mathbf{Fc} = \mathbf{ScE},$$

know as the *Roothaan equations* (ref. [4]). The elements of the Fock matrix depend on the orbital wave-functions, and as before we must apply a self-consistent field approach.

In principle, a complete set of basis functions must be used to represent spin-orbitals exactly, but this is not computationally feasible. A given finite set of basis functions is, due to the incompleteness of the basis set, associated with a *basis-set truncation error*. The limiting HF energy, with truncation error equal to zero, will be referred to as the *Hartree-Fock limit*.

The computational time depends on the number of basis-functions and of the difficulty in computing the integrals of both the Fock matrix and the overlap matrix. Therefore we wish to keep the number of basis functions as low as possible and choose the basis-functions cleverly. By cleverly we mean that the truncation error should be kept as low as possible, and that the computation of the matrix elements of both the overlap and the Fock matrices should not be too time consuming.

One choice of basis functions are *Slater type orbitals* (STO) (ref. [4]),

$$\Psi_{nlm_l}(r,\theta,\phi) = \mathcal{N} r^{n_{eff}-1} e^{\frac{Z_{eff}\rho}{n_{eff}}} Y_{lm_l}(\theta,\phi). \tag{3.18}$$

Here $\mathcal{N}$ is a normalization constant that for the purpose of basis set expansion may be put into the unknown $c_{i\mu}$'s, Y_{lm_l} is a spherical harmonic (see table 2.2) and $\rho = r/a_0$. [1]
The normalization constant of the spherical harmonics may of course also be put into the expansion coefficients $c_{i\mu}$. The effective principal quantum number n_{eff} is related to the true principal quantum number N by the following mapping (ref. [4])

$$n \to n_{eff} : 1 \to 1 \quad 2 \to 2 \quad 3 \to 3 \quad 4 \to 3.7 \quad 5 \to 4.0 \quad 6 \to 4.2.$$

The effective atomic number Z_{eff} for the ground state orbitals of some neutral ground-state atoms are listed in table 3.1. The values in table 3.1 have been constructed by fitting STOs to numerically computed wave-functions [9].

We will in this thesis limit our attention to STO basis functions, as these basis-functions are well suited for atoms.
It should be mentioned however, that depending on the system studied improvements can be made with regard to the basis-set truncation error by choosing different sets of basis-functions; for example *Gaussian-type orbitals* (GTO), *double-zeta basis set* (DZ), *triple-zeta basis set* (TZ) or several other basis sets.

[1] In atomic units $a_0 = 1$ so that $\rho = r$.

Effective Atomic Number

	H							He
1s	1							1.6875
	Li	Be	B	C	N	O	F	Ne
1s	2.6906	3.6848	4.6795	5.6727	6.6651	7.6579	8.6501	9.6421
2s	1.2792	1.9120	2.5762	3.2166	3.8474	4.4916	5.1276	5.7584
2p			2.4214	3.1358	3.8340	4.4532	5.1000	5.7584
	Na	Mg	Al	Si	P	S	Cl	Ar
1s	10.6259	11.6089	12.5910	13.5754	14.5578	15.5409	16.5239	17.5075
2s	6.5714	7.3920	8.2136	9.0200	9.8250	10.6288	11.4304	12.2304
2p	6.8018	7.8258	8.9634	9.9450	10.9612	11.9770	12.9932	14.0082
3s	2.5074	3.3075	4.1172	4.9032	5.6418	6.3669	7.0683	7.7568
3p			4.0656	4.2852	4.8864	5.4819	6.1161	6.7641

Table 3.1: Values of Z_{eff} for neutral ground-state atoms [9].

3.1.3 Density Functional Theory

In the different HF methods one works with large basis sets. This poses a problem for large systems. An alternative to the HF methods is *density functional theory* (DFT) [4]. DFT takes into account electron correlations but is less demanding computationally than for example CI and MP2.

The electronic energy E is said to be a *functional* of the electronic density, $E[\rho]$, in the sense that for a given function $\rho(r)$, there is a single corresponding energy. The *Hohenberg-Kohn theorem* [10] confirms that such a functional exists, but does not tell us the form of the functional. As shown by Kohn and Sham, the exact ground-state energy E of an N-electron system can be written as

$$E[\rho] = -\frac{1}{2}\sum_{i=1}^{N}\int \Psi_i^*(\mathbf{r_1})\nabla_1^2\Psi_i(\mathbf{r_1})d\mathbf{r_1} - \int \frac{Z}{r_1}\rho(\mathbf{r_1})d\mathbf{r_1} + \frac{1}{2}\int \frac{\rho(\mathbf{r_1})\rho(\mathbf{r_2})}{r_{12}}d\mathbf{r_1}d\mathbf{r_2} + E_{XC}[\rho]$$

with Ψ_i the *Kohn-Sham* (KS) *orbitals*. The ground-state charge density is given by

$$\rho(\mathbf{r}) = \sum_{i=1}^{N}|\Psi_i(\mathbf{r})|^2,$$

where the sum is over the occupied Kohn-Sham orbitals. The last term, $E_{EX}[\rho]$, is the *exchange-correlation energy* which in theory takes into account all non-classical electron-electron interaction. However, we do not know how to obtain this term exactly, and are

forced to approximate it. The KS orbitals are found by solving the *Kohn-Sham equations*, which can be found by applying a variational principle to the electronic energy $E[\rho]$. This approach is similar to the one used for obtaining the HF equation in section 3.1.1. The KS equations reads

$$\left\{-\frac{1}{2}\nabla_1^2 - \frac{Z}{r_1} + \int \frac{\rho(\mathbf{r_2})}{r_{12}} d\mathbf{r_2} + V_{XC}(\mathbf{r_1})\right\} \Psi_i(\mathbf{r_1}) = \epsilon_i \Psi_i(\mathbf{r_1})$$

where ϵ_i are the KS orbital energies, and where the *exchange-correlation potential* is given by

$$V_{XC}[\rho] = \frac{\delta E_{XC}[\rho]}{\delta \rho}.$$

The KS equations are solved in a self-consistent fashion. A variety of basis set functions can be used, and the experience gained in HF calculations are often useful. The computational time needed for a DFT calculation formally scales as the third power of the number of basis functions.

The main source of error in DFT usually arises from the approximate nature of E_{XC}. In the *local density approximation* (LDA) it is approximated as

$$E_{XC} = \int \rho(\mathbf{r}) \epsilon_{XC}[\rho(\mathbf{r})] d\mathbf{r},$$

where $\epsilon_{XC}[\rho(\mathbf{r})]$ is the exchange-correlation energy per electron in a homogeneous electron gas of constant density. The LDA approach is clearly an approximation as the charge is not continuously distributed. To account for the inhomogeneity of the electron density, a nonlocal correction involving the gradient of ρ is often added to the exchange-correlation energy.

3.2 Many-Body Methods

Hartree-Fock theory, by assuming a single-determinant form of the wave-function, neglects correlation between electrons. The electrons are subject to an *average* non-local potential arising from the other electrons. This can lead to a poor description of the electronic structure; it does not take into account the *instantaneous* electrostatic interaction between the electrons. A great deal of work in the field of electronic structure calculation aims at including such correlation.

The HF method yields a finite set of spin-orbitals when a finite basis set expansion is used. In general, a set of M basis functions results in $2M$ different spin-orbitals. The N occupied orbitals are used to form the HF ground state, while there remain $2M - N$ virtual orbitals. By using the single determinantal wave-function Φ_0 as a reference, it is possible to classify all other determinants according to how many electrons have been promoted

from occupied orbitals to virtual orbitals. A *singly excited determinant* corresponds to one for which a single electron is excited, a *doubly excited determinant* corresponds to one for which two electrons are excited, etc. Each of the determinants, or a linear combination of a small number of them constructed so as to have the correct symmetry, is called a *configuration state function* (CSF). These excited CSFs can be taken to approximate excited-state wave-functions, or they can be used in a linear combination with Φ_0 to improve the ground (or any excited) state.

3.2.1 Configuration Interaction

In *configuration interaction* (CI) [4] calculations, the ground- or excited-state wave-function is represented as a linear combination of Slater determinants. The method is flexible and give highly accurate wave-functions for small systems, and can be used to describe complex electronic-structure problems. The principal shortcoming of the CI method is that it does not provide a compact description of electron correlation and has a large growth in the number of configurations needed for large systems (ref. [1]). From a complete set of spin-orbitals we can write the exact electronic wave-function Ψ for any state of the system in the form

$$\Psi = C_0\Phi_0 + \sum_{a,i} C_a^i \Phi_a^i + \sum_{ab,ij} C_{ab}^{ij}\Phi_{ab}^{ij} + \sum_{abc,ijk} C_{abc}^{ij}k\Phi_{abc}^{ijk} + \ldots \tag{3.19}$$

where the C's are expansion coefficients, and the Φ_a^i's are the singly excited determinants, the Φ_{ab}^{ij}'s are the doubly excited determinants and so on; the electrons have been promoted from ϕ_a to ϕ_i, from ϕ_a and ϕ_b to ϕ_i and ϕ_j, and so on. A calculation is classified as a *full CI* if all CFS's of the appropriate symmetry are used for a given basis set. As the number of Slater determinants to be determined in a full CI is

$$\begin{pmatrix} 2M \\ N \end{pmatrix}$$

equation (3.19) must for most practical purposes be truncated. This approach is referred to as the *limited CI*; the state is given as the linear combination

$$\Psi_s = \sum_{I=1}^{L} C_{I,s}\Phi_I. \tag{3.20}$$

where the sum is over a finite number (L) of CSF's. The expansion coefficients are determined variationally by minimizing the energy expectation value, or the so-called *Rayleigh ratio*, ref. [4],

$$\langle E\rangle = \frac{\int \Psi^*\hat{H}\Psi}{\int \Psi^*\Psi}.$$

As in the HF approach we arrive at a matrix equation

$$\mathbf{HC} = \mathbf{ESC}$$

with

$$H_{IJ} = \int \Psi_I^* \hat{H} \Psi_J$$

and the overlap matrix defined through

$$S_{IJ} = \int \Psi_I^* \Psi_J.$$

In CI a wave-function is a linear combination of Slater determinants. The orbitals in each determinant are again a linear combination of basis functions. This makes the evaluation of the matrix-elements of **H** and **S** very important in CI calculations.

One deficiency that plagues limited CI calculations is the lack of *size-consistency.* A method is size-consistent if the energy of a many-electron system is proportional to the number of electrons N in the limit $N \to \infty$. In particular, the energy of a system AB computed when the systems are far apart should equal the sum of energies of the two systems A and B when computed separately.

In *multireference configuration iteration* (MRCI), a set of reference configurations is created, from which excited determinants are formed for use in a CI calculation. For example, a MCSCF (section 3.2.4) calculation could be performed and a set of reference configurations composed of those determinants having a coefficient greater than a given threshold value. For each of the reference determinants, electrons are moved from occupied to unoccupied orbitals to create more orbitals for inclusion in the CI expansion of eq. (3.20).

3.2.2 Coupled-Cluster

The coupled-cluster (CC) method [11] is one of the the most important practical advances over the CI method. Although non-variational, it resolves the problem of size extensivity, and is very accurate. Of the different many-body methods, Coupled-Cluster (CC) represents the most effective method for atomic systems. In practice, the CC method is restricted to systems that are dominated by a single electronic configuration, and the CC wave-function is best regarded as providing an accurate correction to the HF description, see ref. [1].

The CC method assumes an exponential ansatz for the wave-function

$$\Psi_{CC} = exp(\hat{T}\Psi_{HF})$$

where the coupled-cluster wave-function is given by an excitation operator acting on a reference wave-function, usually the Hartree-Fock determinant $\Psi_{HF} = D_0$. The operator $\hat{T}$ generates k-fold excitations from a reference state

$$\hat{T} = \sum_k \hat{T}_k$$

so that, for example,

$$\hat{T}_2 \Psi_{HF} = \sum_{ij,ab} t_{ij}^{ab} D_{ij}^{ab}$$

with D_{ij}^{ab} the excitation of the occupied states ij to the virtual states ab. The expansion coefficients t_{ij}^{ab} are determined self-consistently. A 'coupled-cluster doubles' wave-function is written as

$$\begin{aligned}\Psi_{CCD} &= exp(\hat{T}_2)\Psi_{HF} = (1 + \hat{T}_2 + \frac{1}{2!}\hat{T}_2^2 + \ldots)\Psi_{HF} \\ &= D_0 + \sum_{ij,ab} t_{ij}^{ab} D_{ij}^{ab} + \frac{1}{2}\sum_{ij,ab}\sum_{kl,cd} t_{ij}^{ab} t_{kl}^{cd} D_{ij}^{ab} D_{kl}^{cd} + \ldots .\end{aligned}$$

The CC expansion is usually terminated after all double excitations or all quadruple excitations have been included. It can be shown [11] that this expansion is size consistent. By including many terms in the expansion, CC methods are computationally very expensive relative to HF calculations. Formally, CC singles and doubles scale as the sixth power of the number of basis states included in the expansion, and calculations including up to quadruple excitations scale as the tenth power of the number of states. The key limitation of the CC methods is the rapid increase in computational cost with system size; scalings are to the sixth power of the number of particles (or higher).

3.2.3 Møller-Plesset Many-Body Perturbation Theory

Perturbation theory (PT) provides an alternative approach to finding the correlation energy. Perturbative methods are size-consistent, but the energies are not in general upper bounds to the exact energy.

Perturbation theory applied to a system of many interacting particles is generally called *many-body perturbation theory* (MBPT). In the method called *Møller-Plesset Many-Body Perturbation Theory* (MPPT) the zero-order Hamiltonian $H^{(0)}$ is taken as the sum of one-electron Fock operators defined through equation (3.16).

The perturbation $H^{(1)}$ is given by

$$H^{(1)} = \hat{H} - H^{(0)} = \hat{H} - \sum_{i=1}^{N} H_i^{HF}.$$

The HF energy E_{HF} associated with the (normalized) ground-state HF wave-function Φ_0 is the expectation value

$$\langle\Phi_0|\hat{H}|\Phi_0\rangle = \langle\Phi_0|H^{(0)} + H^{(1)}|\Phi_0\rangle.$$

If we define the zero-order energy

$$E^{(0)} = \langle\Phi_0|H^{(0)}|\Phi_0\rangle,$$

and the first-order correction

$$E^{(0)} = \langle\Phi_0|H^{(1)}|\Phi_0\rangle,$$

The HF energy becomes the sum of these

$$E^{HF} = E^{(0)} + E^{(1)}.$$

Therefore the first correction to the HF ground state energy is given by the second order perturbation

$$E^{(2)} = \sum_{J\neq 0} \frac{\langle\Phi_0|H^{(1)}|\Phi_J\rangle\langle\Phi_J|H^{(1)}|\Phi_0\rangle}{E^{(0)} - E_J}. \tag{3.21}$$

This equation is readily shown by considering the following set of expansions:

$$\hat{H} = H^{(0)} + \lambda H^{(1)} + \lambda^2 H^{(2)} + \ldots,$$

for the Hamiltonian,

$$\Psi = \Psi_0^{(0)} + \lambda\Psi_0^{(1)} + \lambda^2\Psi_0^{(2)} + \ldots,$$

for the state, and finally

$$E = E_0^{(0)} + \lambda E_0^{(1)} + \lambda^2 E_0^{(2)} + \ldots,$$

for the energy. Here λ is introduced to keep track of the order of the perturbation, and set equal to unity after the order of the terms is worked out. We start by inserting the above expansions into Schrödinger's equation

$$\hat{H}\Psi = E\Psi,$$

and rearrange the terms by the order of λ. The expressions of the different orders of perturbation is then obtained by setting the coefficients of each power of λ equal to zero, and by integrating from the left with $\int \Phi_J$. This can be done because, as of yet, λ is arbitrary. In our particular case $\hat{H} = H^{(0)} + \lambda H^{(1)}$ so the only remaining term in the second order perturbation is the one given by equation (3.21).

Note that the following matrix elements are all zero

$$\langle\Phi_J|H_{HF}|\Phi_0\rangle = \int \Phi_J H_{HF}\Phi_0 d\tau = 0.$$

This is because the Φ_J's are eigenfunctions of H_{HF} and hence orthogonal. Therefore, the following must be true

$$\langle \Phi_J | H | \Phi_0 \rangle = \langle \Phi_J | H^{(1)} | \Phi_0 \rangle.$$

By *Brillouin's theorem* [4], Hamiltonian matrix elements between Ψ_0 and all singly excited states vanish, so as a first approximation we keep the doubly excited states. An analysis of these matrix elements yields the following expression

$$E^{(2)} = \frac{1}{4} \sum_{ab}^{occ} \sum_{pq}^{vir} \frac{\langle ab||pq\rangle \langle pq||ab\rangle}{\epsilon_a + \epsilon_b - \epsilon_p - \epsilon_q},$$

where

$$\langle ab||pq\rangle = \int \phi_a^*(1)\phi_b^*(2)H^{(1)}\phi_p(1)\phi_a(2)d\mathbf{x_1}d\mathbf{x_2} - \int \phi_a^*(1)\phi_b^*(2)H^{(1)}\phi_p(2)\phi_a(1)d\mathbf{x_1}d\mathbf{x_2}$$

with ϕ_a, ϕ_b occupied spin-orbitals and ϕ_p, ϕ_q virtual spin-orbitals. The inclusion of the second-order energy correlation is labeled MP2. Third and fourth order corrections are referred to as MP3 and MP4. As one moves to higher orders of perturbation, the algebra involved becomes more and more complicated and it is common to use diagrammatic representation. These diagrams can be used to prove that MPPT is size-consistent in all orders.

3.2.4 Multiconfiguration Methods

Electronic wave-functions are often dominated by more than one electronic configuration, and the presence of several important configurations poses a challenge for electronic structure theory. For such systems CC and MP are not so well suited [1]; the orbitals generated self-consistently in the field of a single electronic configuration may have little relevance to a multiconfigurational system. In the *multiconfiguration self-consistent field method* (MCSCF) both the coefficients $C_{I,s}$ of equation (3.20) and the coefficients $c_{j,\mu}$ of equation (3.17) are optimized. This simultaneous optimization of both the orbitals and the CSF's expansion coefficients makes MCSCF computationally demanding. However, accurate results can be obtained with the inclusion of even a relatively small number of CSF's. The development of effective MCSCF methods is still actively being pursued and is particularly important for excited states.

One such scheme is the *complete active-space self-consistent field method* (CASSCF). In this approach the spin-orbitals are divided into three classes; *inactive*, *virtual* and *active* orbitals. The inactive orbitals are the doubly occupied orbitals, the virtual orbitals are the sets of very high energy spin-orbitals which are unoccupied, and the active orbitals are the energetically intermediate orbitals. The CSF's included in the CASSCF calculation are configurations (of the appropriate symmetry and spin) that arise from all possible ways of distributing the active electrons over the active orbitals.

3.3 Monte Carlo Theory

In this thesis the Variational Monte Carlo (VMC) method is studied for atomic systems. A *Monte Carlo* (MC) method is a method based on random numbers. It is therefore stochastic and has associated statistical properties. Monte Carlo methods are widely used, and are particularly interesting for solving high-dimensional integrals, which is the case for VMC. In this section we will outline the basics of MC methods, and develop the theoretical description of the VMC routine. Also, we will look at the basic principles of the Diffusion Monte Carlo (DMC) method. Other methods like Path Integral Monte Carlo, Auxiliary Field Monte Carlo and Reptation Monte Carlo have become popular over the last decade, but are omitted in this presentation.

The VMC method is more effective when combined with DMC. This combination has proved accurate for several quantum mechanical systems. The ultimate accuracy of the DMC is only limited by the trial function. A common procedure is to use the VMC algorithm to find an optimized trial wave-function, and use this optimized form as the input function for the DMC calculation.

Even though DMC calculations are not performed in this thesis, one should keep in mind that VMC is used primarily as a starting point for DMC calculations. Trial wave-functions are optimized through VMC and are used as the input to a DMC calculation. The DMC method in itself gives little insight in terms of understanding the physics involved in the system, whereas our physical understanding is incorporated into the variational trial wave-function. Therefore, understanding how the VMC algorithm works and how to construct trial-wave-functions is important before implementing the DMC method. Furthermore, when developing the VMC code most of the key building blocks of the DMC method are also created. The purpose of this thesis is to develop the VMC algorithm. Our focus is mainly to make the code fast and flexible. In addition, some of the basic insights regarding the making of trial wave-functions will be established.

3.3.1 Monte Carlo Integration

To motivate for Monte Carlo integration we first take a look at the conventional methods. Conventional integration methods involve choosing some evaluation points and combine the value of the integrand with weights for each and every point,

$$\int_{\Omega} f(\mathbf{x}) d\Omega = \sum_{i=1}^{m} \omega_i f(\mathbf{x}_i). \tag{3.22}$$

The value of the weights ω_i are associated with how the evaluation points have been chosen. In one dimension the simplest form of eq. (3.22) is to choose the evaluation points to be equally spaced. The weights then become the length of the interval divided by the number of integration points, m. For simplicity we assume that the integration interval has been

transformed to the unit length. This gives

$$\int_0^1 f(x)dx = \frac{1}{m}\left(\sum_{i=1}^{m} f(x_i) + \mathcal{O}(\frac{1}{m})\right).$$

Similarly, a two dimensional integration on the unit square may be approximated as

$$\int_0^1\int_0^1 f(x,y)dxdy = \frac{1}{m^2}\left(\sum_{i=1}^{m}\sum_{j=1}^{m} f(x_i,y_j) + \mathcal{O}(\frac{1}{m})\right),$$

where we have taken the number of integration points to be equal in each of the two dimensions. For two dimensions we therefore need N^2 integration points instead of the m points we needed in one dimension to obtain the same order of accuracy. By the same argument, integration over a d dimensional unit cube needs m^d integration points to obtain an accuracy of order $1/m$. For a d dimensional quantum N-body systems we have dN spatial degrees of freedom. The integral becomes an integral over dN dimensions, and we need m^{dN} points to acquire an accuracy of order $1/m$. Sophistication of the traditional methods is obtained by choosing the points cleverly. By choosing more points where the function varies more than where the function is smooth, the number of integration points may be reduced while at the same time preserving the accuracy. However, the number of calculations needed is still of the order of dN.

As can be seen from this argument the conventional integration procedure is practically impossible to carry out as N grows large. This problem is avoided in the HF approach by solving the one-particle mean-field HF equations. The problem is here reduced to N $2d$-dimensional ones, where the factor two is due to the direct and the exchange terms of the HF approximation. Also, by choosing the basis functions cleverly the evaluation of these integrals can be made an easy task. Integral-evaluations in most other numerical many-body methods are similarly reduced to integrals over d or $2d$ dimensions.

The introduction of Monte Carlo integration makes high-dimensional integration possible. Here the integrand is evaluated at random points $\mathbf{x}_i$ taken from an arbitrary probability distribution $\rho(\mathbf{x})$ (ref. [12]),

$$\int_\Omega f(\mathbf{x})d\Omega = \int_\Omega \frac{f(\mathbf{x})}{\rho(\mathbf{x})}\rho(\mathbf{x})d\Omega \equiv \int_\Omega g(\mathbf{x})\rho(\mathbf{x})d\Omega. = \sum_{i=1}^{m} g(\mathbf{x}_i) + \mathcal{O}(\frac{1}{\sqrt{m}}). \tag{3.23}$$

In the limit of $m \to \infty$ this approximation is exact, but in a numerical approach we are forced to truncate the summation at some finite value m.

By choosing $\rho(\mathbf{x}) = 1$ we simply sample the integrand uniformly at freely chosen random points. If the function varies considerably over the integration domain, the variation of the individual samples will be considerable. The trick is therefore to choose the function $\rho(\mathbf{x})$ to duplicate the behavior of the function $f(\mathbf{x})$ we want to integrate. If we choose

$\rho(\mathbf{x}) = \mathcal{N} f(\mathbf{x})$ the fraction $g(\mathbf{x}) = f(\mathbf{x})/\rho(\mathbf{x}) = \mathcal{N}$ is simply a constant. This means that we obtain the true value of the integral (here the normalization constant $\mathcal{N}$) by taking only one sample of the function $g(\mathbf{x})$. Of course this mean that we must know the value $\mathcal{N}$ of the integral in advance, and this makes little sense as this is the value we want to find. From a theoretical point of view, however, this example illustrates how we can optimize the Monte Carlo integration scheme by choosing the shape of the probability distribution to be as close to the integrand $f(\mathbf{x})$ as possible. This optimization routine is referred to as *importance sampling*.

The true beauty of the Monte Carlo integration scheme is that it is inherently independent of the number of dimensions. The evaluation time of the integrand depends solely on its functional form. Also, the variance of the integral estimate depends solely on how much the integrand varies. For a practical application to the quantum mechanical N-body fermionic system, the evaluation of the Slater determinant in the integrand yields approximately N^3 calculations. Furthermore, an additional factor N^s is included because, with increasing N, the trial wave-function (section 3.3.6) becomes less accurate and the auto-correlation effects (section 3.3.4) become increasingly significant. For atomic systems the VMC routine is thus of the order N^{3+s}. We will provide an example of this in section 5.2.

In the case of atoms, the wave-function varies greatly near the nucleus and is smoother further away. For eigenstates the energy does not vary at all, since

$$\hat{H}\Psi_k = E_k\Psi_k.$$

Therefore, for the eigenstate the energy expectation value

$$\langle E \rangle = \frac{\int \Psi^*(\mathbf{X})\hat{H}\Psi(\mathbf{X})d\tau}{\int \Psi^*(\mathbf{X})\Psi(\mathbf{X})d\tau},$$

depends only on $|\Psi(\mathbf{X})|^2$. This implies that sampling random points from the $|\Psi(\mathbf{X})|^2$ distribution, is equivalent to sampling from the integrand $\Psi^*(\mathbf{X})\hat{H}\Psi(\mathbf{X})$. In Variational Monte Carlo we sample from the square of a trial wave-function $\Psi_{trial}(\mathbf{X})$, not the eigenstate. The relation $\hat{H}\Psi = E\Psi$ no longer holds, but as the trial wave-function limits the true eigenstate the variation of the integrand vanishes. The number of samples needed to obtain a given accuracy depends therefore directly on the quality of the trial wave-function.

Before we are ready to formulate the VMC routine we need a routine to sample probability-distributions.

3.3.2 The Metropolis Algorithm

The *Metropolis algorithm* [13] generates a stochastic or random sequence of phase space points that sample a given probability distribution. In Quantum Monte Carlo (QMC) methods, each point in phase space is a vector $\mathbf{X} = \{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_N}\}$ in the Hilbert space.

Here $\mathbf{x_i}$ represents the spatial and the spin coordinates of electron i. Each point in the phase space, coupled with a quantum mechanical operator, can be associated with physical quantities, such as the kinetic and potential energy. The sequence of individual *samples* of these quantities can be combined to arrive at average values which describe the quantum mechanical state of the system. This is the fundamental idea behind the Metropolis algorithm, and the algorithm is used to generate the sample points. We will refer to the randomized walk through the phase space as a *random walk*. From an initial position in phase or configuration space, a *proposed move* is generated and the move either *accepted* or *rejected* according to the Metropolis algorithm. A random walk thus generates a sequence

$$\{\mathbf{X}_0, \mathbf{X}_1, \ldots, \mathbf{X}_i, \ldots, \}$$

of points in the phase space. An important requirement is for the random walk to be *ergodic*, which means that all points in the phase or configuration space can be reached from any initial point. By taking a sufficient number of trial steps all of phase space is then explored and the Metropolis algorithm ensures that the points are distributed according to the required probability distribution.

Let us for the time being suppose that we know the probability distribution $\rho(\mathbf{X})$ we are to draw the points from. Metropolis *et al* [13] showed that the sampling is most easily accomplished if the points $\mathbf{X}$ form a *Markov chain*. A random walk is Markovian if each point in the chain depends only on the position of the preceding point. A Markov process may be completely specified by choosing values of the transition probabilities $P(\mathbf{X}_n, \mathbf{X}_m)$ of moving from $\mathbf{X}_n$ to $\mathbf{X}_m$. The Metropolis algorithm works by choosing the transition probabilities in such a way that the sequence of points generated by the random walk sample the required probability distribution. To understand the Metropolis algorithm it is necessary to work out the statistical properties of the points on the Markov chain. This may be done by considering a large ensemble of random walkers all evolving simultaneously. The walkers all move step by step in accordance with the transition probabilities. At a given time the number of walkers at a point $\mathbf{X}_n$ is $N(\mathbf{X}_n, t)$. As the Markov chains evolve in time the number of walkers develops according to the Master equation,

$$\frac{d}{dt}N(\mathbf{X}_n, t) = -\sum_{\mathbf{X}_m} P(\mathbf{X}_n, \mathbf{X}_m)N(\mathbf{X}_n, t) + \sum_{\mathbf{X}_m} P(\mathbf{X}_m, \mathbf{X}_n)N(\mathbf{X}_m, t).$$

As $t \to \infty$ the derivative $dN(\mathbf{X}_n, t)/dt \to 0$ so that

$$\sum_{\mathbf{X}_m} P(\mathbf{X}_n, \mathbf{X}_m)N(\mathbf{X}_n) = \sum_{\mathbf{X}_m} P(\mathbf{X}_m, \mathbf{X}_n)N(\mathbf{X}_m),$$

where $N(\mathbf{X}_n, t) \to N(\mathbf{X}_n)$. Metropolis realized that the distribution of walkers will settle down to the required distribution $\rho(\mathbf{X})$ as long as the transition probabilities obey the equation of *detailed balance*

$$P(\mathbf{X}_n, \mathbf{X}_m)\rho(\mathbf{X}_n) = P(\mathbf{X}_m, \mathbf{X}_n)\rho(\mathbf{X}_m).$$

Imposing the condition of detailed balance is a sufficient but not a necessary requirement for a random process to sample the phase space with the probability density $\rho(\mathbf{X})$. Other transition probabilities can be created, and are particularly interesting for reducing autocorrelation effects (see section 3.3.4). Such approaches have unfortunately not been studied in this thesis. Imposing the condition of detailed balance on the transition probabilities gives

$$\sum_{\mathbf{X}_m} P(\mathbf{X}_n, \mathbf{X}_m) \left(N(\mathbf{X}_n) - \frac{\rho(\mathbf{X}_n)}{\rho(\mathbf{X}_m)} N(\mathbf{X}_m) \right) = 0,$$

so that

$$\frac{\rho(\mathbf{X}_n)}{\rho(\mathbf{X}_m)} = \frac{N(\mathbf{X}_n)}{N(\mathbf{X}_m)}.$$

This shows us that the number of walkers in the state $\mathbf{X}_n$ becomes proportional to the steady state distribution $\rho(\mathbf{X}_n)$, which we wish to sample.

We still have some freedom in choosing the transition probabilities, which are not uniquely defined by the detailed balance condition. In the Metropolis approach, the walk is generated by starting from a point $\mathbf{X}_n$ and making a *trial move* to a new point $\mathbf{X}_m$ somewhere nearby in phase space. The way we choose our trial move is not crucial, except that it is important to satisfy the detailed balance condition. One such approach is to choose

$$P_{trial}(\mathbf{X}_n, \mathbf{X}_m) = P_{trial}(\mathbf{X}_m, \mathbf{X}_n),$$

and then *accepted* or *rejected* according to the rule

$$P_{accept}(\mathbf{X}_n, \mathbf{X}_m) = min\left(1, \frac{\rho(\mathbf{X}_m)}{\rho(\mathbf{X}_n)}\right).$$

Note that since this method involves the ratios of probabilities there is no need to worry about normalization of the distribution $\rho(\mathbf{X}_n)$. Combining the trial and acceptance probabilities we get

$$\frac{P(\mathbf{X}_n, \mathbf{X}_m)}{P(\mathbf{X}_m, \mathbf{X}_n)} = \frac{P_{trial}(\mathbf{X}_n, \mathbf{X}_m) P_{accept}(\mathbf{X}_n, \mathbf{X}_m)}{P_{trial}(\mathbf{X}_m, \mathbf{X}_n) P_{accept}(\mathbf{X}_m, \mathbf{X}_n)} = \frac{\rho(\mathbf{X}_m)}{\rho(\mathbf{X}_n)}$$

and hence the detailed balance condition is satisfied.

3.3.3 Variational Monte Carlo

The Variational Monte Carlo (VMC) routine uses the Metropolis algorithm combined with Monte Carlo integration and the variational principle in finding the eigenstates of the system. Finding the ground state energy of a many-body system of N particles in d dimensions requires the minimization of the dN dimensional integral

$$\langle E \rangle = E[\Psi] = \frac{\int \Psi^*(\mathbf{X}) \hat{H} \Psi(\mathbf{X}) d\tau}{\int \Psi^*(\mathbf{X}) \Psi(\mathbf{X}) d\tau}. \tag{3.24}$$

In general the energy will, according to the variational principle, be a minimum for the exact ground state wave-function Ψ_0. The functional $E[\Psi]$ thus provides an upper bound to the ground state energy.

We rewrite eq. (3.24) as

$$\langle E \rangle = \int E_L(\mathbf{X}) \frac{|\Psi(\mathbf{X})|^2}{\int |\Psi(\mathbf{X})|^2 d\tau} d\tau, \tag{3.25}$$

where the *local energy* has been defined as

$$E_L = \frac{1}{\Psi(\mathbf{X})} \hat{H} \Psi(\mathbf{X}). \tag{3.26}$$

The square of the wave-function divided by its norm may be interpreted as the probability distribution of the system of particles. So we arrive at

$$\langle E \rangle = \int E_L(\mathbf{X}) \rho(\mathbf{X}) d\tau, \tag{3.27}$$

with

$$\rho(\mathbf{X}) = \frac{|\Psi(\mathbf{X})|^2}{\int |\Psi(\mathbf{X})|^2 d\tau}.$$

This integral is carried out by Monte Carlo integration. We move a walker randomly through the phase space according to the Metropolis algorithm, and sample the local energy with each move. This gives us a statistical evaluation of the integral.

The VMC algorithm consists of two distinct phases. In the first *thermalization* phase the walker is propagated by the Metropolis algorithm, in order to equilibrate it according to the probability distribution $\rho(\mathbf{X})$.[2] In the second phase, the walker continues to be moved, but energies and other observables are also *sampled* for computation of averages and other statistical quantities.

In this algorithm, the electrons are moved individually and not as a whole configuration. This improves the efficiency of the algorithm for larger systems, as full configuration-moves require increasingly small steps to maintain the acceptance ratio.

[2] Remember that the Metropolis algorithm reproduces the probability distribution in the limit $t \to \infty$.

VMC algorithm

Scheme for the VMC algorithm
Generate initial randomized electron configuration.
loop (*thermalization steps*)
  loop (*all electrons*)
    Propose move from $\mathbf{X}$ *to* $\mathbf{X}'$.
    Compute ratio $R = |\Psi(\mathbf{X}')/\Psi(\mathbf{X})|^2$.
    Accept or reject move according to the Metropolis probability min(1,R).
Initialize samplers.
loop (*VMC steps*)
  loop (*all electrons*)
    Propose move from $\mathbf{X}$ *to* $\mathbf{X}'$.
    Compute ratio $R = |\Psi(\mathbf{X}')/\Psi(\mathbf{X})|^2$.
    Accept or reject move according to the Metropolis probability min(1,R).
    Sample the contributions to the local energy and other observables.
Calculate and return statistical properties.

3.3.4 Statistical Analysis

The statistical analysis of the results produced through the VMC routine needs special attention. First of all, a statistically produced average or *mean* has little or no value by itself, and should always be given with its associated variance or *standard deviation*. Second, in the practical application of the VMC routine the individual samples are correlated[3].

Given a sequence of normally distributed data points $\{A_1, A_2, \ldots, A_M\}$ the mean is given by

$$\bar{A} = \frac{1}{M} \sum_{i=1}^{M} A_i,$$

and the *standard deviation* is given by

$$\sigma^2 = \frac{1}{M-1} \sum_{i=1}^{M} \sum_{j=1}^{M} \left(A_i - \bar{A}\right) \left(A_j - \bar{A}\right).$$

The standard deviation can be split into two terms

[3] Do not confuse the statistical correlation with the correlation effects we optimize through the introduction of a Jastrow factor.

$$\sigma^2 = \frac{1}{M-1}\sum_{i=1}^{M}\left(A_i - \bar{A}\right)^2 + \frac{2}{M-1}\sum_{i=1}^{M-1}\sum_{j>i}^{M}\left(A_i - \bar{A}\right)\left(A_j - \bar{A}\right). \tag{3.28}$$

The first term of eq. (3.28) is the un-correlated estimate of the variance, the second term is the covariance. For un-correlated data this last term is zero, but for the Markov chains produced by the Metropolis algorithm this is not the case. Therefore correlation effects must be included in our statistical analysis of the sample points. The straightforward approach of using eq. (3.28) directly is extremely time-consuming when a large number of samples are included. The double sum makes such calculations virtually impossible for practical applications. One way to approach this problem is to split the double sum of the covariance into $M-1$ single sums,

$$\begin{aligned}\sum_{i=1}^{M-1}\sum_{j>i}^{M}\left(A_i - \bar{A}\right)\left(A_j - \bar{A}\right) = &\sum_{i=1}^{M-1}\left(A_i - \bar{A}\right)\left(A_{i+1} - \bar{A}\right) \\ &+ \sum_{i=1}^{M-2}\left(A_i - \bar{A}\right)\left(A_{i+2} - \bar{A}\right) \\ &\cdots \\ &+ \sum_{i=1}^{1}\left(A_i - \bar{A}\right)\left(A_{i+M-1} - \bar{A}\right).\end{aligned}$$

If we define

$$\tau_m \equiv \sum_{i=1}^{M-m}\left(A_i - \bar{A}\right)\left(A_{i+m} - \bar{A}\right)$$

we simply get

$$\sum_{i=1}^{M-1}\sum_{j>i}^{M}\left(A_i - \bar{A}\right)\left(A_j - \bar{A}\right) = \sum_{m=1}^{M-1}\tau_m \tag{3.29}$$

which is computed by considering samples that are separated (in the chain of samples) by a distance m.

The value of τ_m reflects how much the samples separated by a distance m (in the chain of samples) are correlated. Given large enough distances the correlation effects should eventually die out, and τ_m should therefore be equal to zero for all $m \geq k$. The value k is called the *correlation length*. We should by this argument be able to truncate the sum in eq. (3.29) at the value k. In theory this seems promising, but in practice this approach is not well suited. Because of the statistical nature of the sample points, the value of τ_m will not die out completely, but fluctuate around zero. Truncation at different values of

m therefore yields different estimates of the variance, and these estimates do not relax to any given value even as $m >> k$. The fluctuations in τ_m are too moderate for automated procedures to give reliable estimates, and the estimation of τ_m therefore needs special care for each individual data sets.

A simple procedure commonly used in the literature is the procedure known as *reblocking* or *block analysis* (see for example ref. [12]). This procedure is based on a reblocking of the data into a series of blocks of varying sizes, and the standard deviation can be obtained by computing the variance of the block averages. For each block an estimate of the (un-correlated) standard deviation of the mean is given by

$$\sigma_b^2 = \frac{1}{M_b - 1} \sum_{j=1}^{M_b} (A_{b,j} - \bar{A})^2,$$

where M_b is the number of blocks of size b, and $A_{b,j}$ is the average of the samples in block j (Note that the average remains unchanged $\bar{A}_b = \bar{A}$). The idea behind the reblocking scheme is that as the block size increases, the estimate of the standard deviation will eventually relax to the true correlated standard deviation. This can be seen by some simple manipulation of eq. (3.28). Start by defining the un-correlated part as

$$\sigma_u^2 \equiv \frac{1}{M - 1} \sum_{i=1}^{M} \left(A_i - \bar{A}\right)^2,$$

and define for short the covariance as

$$cov \equiv \frac{1}{M - 1} \sum_{i=1}^{M-1} \sum_{j>i}^{M} \left(A_i - \bar{A}\right) \left(A_j - \bar{A}\right).$$

Rewriting of eq. (3.28) by the above equations yield

$$\sigma^2 = \sigma_u^2 \left(1 + \frac{2}{\sigma_u^2} cov\right) \equiv \kappa \sigma_u^2, \tag{3.30}$$

where κ is the *auto-correlation time*. When the size of the blocks are longer than the correlation length ($b >> \tau_k$) the individual block samples $A_{b,j}$ become un-correlated. When the blocks are un-correlated the auto-correlation time is equal to unity, $\kappa = 1$. This procedure works much better than the first procedure, as the fluctuations in τ_m are 'smeared out' due to the block averages. However, fluctuations are experienced in this approach also.

Another problem that may occur is auto-correlation effects on different time scales. Consider the one-dimensional trial wave-function depicted in figure 3.1. In this example the instantaneous evaluation of different physical quantities may vary in the two regions A and B. Random movement between the two regions is unlikely by making only short steps in the Metropolis algorithm, but it will eventually happen. This results in an auto-correlation that becomes visible only on large time scales.

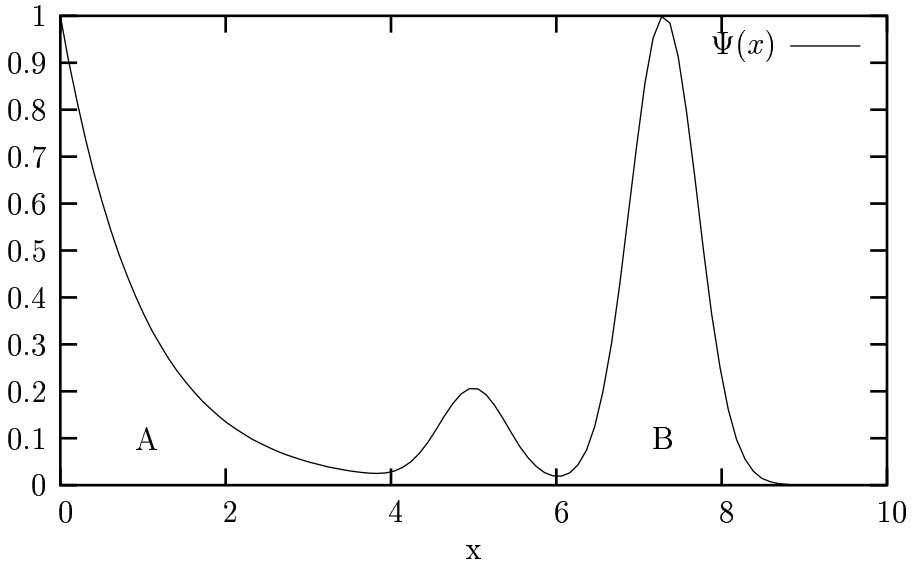

Figure 3.1: Fictive one-dimensional wave-function included for illustrative purposes.

3.3.5 Kato Cusp Conditions

An important physical feature in quantum mechanics are the so-called *cusp conditions*, see ref. [14]. When two particles are close in proximity the Coulomb forces between these two particles become dominant, and the influence of all other particles become unimportant. This fact should be incorporated into the wave-function, so that the diverging Coulomb potential is cancelled by a corresponding divergence in the kinetic energy. This constraint on the wave-function is *local* and is a constraint on the derivatives of the wave-function.

To examine the cusp condition we return our attention to the radial part of the hydrogenic wave-function (given by eq. 2.9 with $V(r) = -Z/r$),

$$\left[-\frac{1}{2}\frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}\right) - \frac{Z}{r} + \frac{l(l+1)}{r^2}\right] R(r) = ER(r). \tag{3.31}$$

For $l = 0$ the two $1/r$ terms must cancel, which leads to the cusp condition

$$\left.\frac{dR(r)}{dr}\right|_{r=0} = -ZR(r)|_{r=0}.$$

For $l \neq 0$ the angular momentum term must be canceled by the kinetic energy. By factoring out the leading r dependency, as we did when introducing the solid harmonics in section 2.2, namely

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

the kinetic energy term becomes

$$r^l\mathcal{R}''(r) + \frac{2(l+1)}{r}r^l\mathcal{R}'(r) + \frac{l(l+1)}{r^2}r^l\mathcal{R}(r). \tag{3.32}$$

We see that the $1/r^2$ term in eq. 3.31 is cancelled by the last term of eq. 3.32, yielding the following equation for $\mathcal{R}(r)$

$$\left(\frac{2(l+1)}{r}\frac{\mathcal{R}'(r)}{\mathcal{R}(r)} + \frac{2Z}{r} + \frac{\mathcal{R}''(r)}{\mathcal{R}(r)} + 2E\right) r^l\mathcal{R}(r) = 0$$

By equating the $1/r$ terms we arrive at the general electron-nucleus cusp condition

$$\left.\frac{d\mathcal{R}(r)}{dr}\right|_{r=0} = -\frac{Z}{l+1}\mathcal{R}(r)|_{r=0}. \tag{3.33}$$

For hydrogenic systems the Kato cusp condition uniquely determines the overall exponential behavior of the wave-function for each value of l, $R(l=0) \propto e^{-Zr}$, $R(l=1) \propto e^{-Zr/2}$, $R(l=2) \propto e^{-Zr/3}$, etc., which is in accordance with table 2.3 of section 2.2.

The electron-electron cusp can be derived by a similar argument. In this case, both electrons contribute to the kinetic energy. Expanding the wave-functions in spherical coordinates centered on electron i, leads to (ref. [15])

$$\left(2\frac{d^2}{dr_{ij}^2} + \frac{4}{r_{ij}}\frac{d}{dr_{ij}} + \frac{2}{r_{ij}} - \frac{l(l+1)}{r_{ij}^2} + 2E\right) R_{ij}(r_{ij}) = 0,$$

which gives the electron-electron cusp condition

$$\left.\frac{d\mathcal{R}(r_{ij})}{dr_{ij}}\right|_{r_{ij}=0} = \frac{1}{2(l+1)}\mathcal{R}(r_{ij})|_{r_{ij}=0}.$$

For electrons with anti-parallel spin the most likely configuration is the energetically lowest one, in which $l = 0$. This implies a cusp condition equal to $1/2$. Two electrons cannot both occupy the same state by the Pauli principle. For two electrons with parallel spin the energetically lowest configuration is therefore for $l = 1$. We arrive at the general electron-electron cusp condition

$$\left.\frac{d\mathcal{R}(r_{ij})}{dr_{ij}}\right|_{r_{ij}=0} = \begin{cases} 1/2 \text{ for } spin(i) \neq spin(j) \\ 1/4 \text{ for } spin(i) = spin(j) \end{cases}.$$

The hydrogen-like cusp conditions apply to the many-electron system. As one single electron approaches the nucleus or another electron, the exact wave-function behaves asymptotically to the hydrogenic wave-function. Higher order cusp conditions are of course also possible. As two electrons are close to the nucleus simultaneously, an extension to the above picture applies. The theoretical derivation of such higher-order terms is omitted here.

3.3.6 The Trial Wave-Function

The development of Slater determinants is outlined in the beginning of this chapter. Even though both HF and DFT are able to generate good single-electron orbitals, the approximations introduced by both of these methods result in a poor description of the instantaneous structure of the system. In the many-body methods these approximations are accounted for by generating linear combinations of several Slater determinants. In the limit of including an infinite number of Slater determinants these methods are exact. These methods are limited by basis set truncation errors and slow convergences.

Variational approaches are limited by how well the variational or *trial wave-function* approximates the behavior of the true eigenstate, and by the method used to optimize the parameters of the trial function. Most variational methods rely on a double basis set expansion in one-electron orbitals, combined into one or more N-electron Slater determinants. A unique characteristics of Monte Carlo methods is their ability to use arbitrary wave-function forms, thereby enabling treatment beyond the Slater determinants constructed solely with one-electron functions.

In the literature, see for example refs. [12, 15], the common approach is to begin with a Slater determinant. The Slater determinant is then multiplied with a variational *Jastrow-factor*, G_β. This Jastrow factor is constructed to include two-body and higher order correlation effects. To obtain even better results, the Slater determinant is substituted with a linear-combination of several Slater determinants. This latter approach is not implemented in the code developed in this thesis, but is a natural extension of the existing code. For now we limit our attention to single determinants, D_α. This results in a trial wave-function of the form

$$\Psi_{\alpha,\beta}(\mathbf{X}) = D_\alpha(\mathbf{X})G_\beta(\mathbf{X}),$$

where $\alpha = \{\alpha_i\}$ and $\beta = \{\beta_i\}$ are the variational parameters.

In this thesis we have used two forms of the Slater determinant. The first form which consisted of variational hydrogen orbitals was easy to implement and therefore provided a practical means for testing the code as it developed. The radial functions listed in table 2.3 were used, with the charge Z exchanged with a variational parameter α, and combined with the solid harmonics of table 2.4; for example $\phi_{2p_{m=-1}}(\mathbf{r}_i) = y_i e^{-\alpha r_i}$. The determinant used for producing results, however, was composed of Slater Type Orbitals (STO) optimized through restricted Hartree-Fock (RHF), see ref. [16], also combined with the solid harmonics. The HF energies were accurately reproduced by the VMC code of this thesis, providing an excellent test of the Slater-dependent part of our code.

Several forms of the Jastrow-factor exist in the literature, and we will mention only a selected few to give the general idea of how they are constructed. A form given by Hylleraas (taken from ref. [15]) for the helium atom is

$$G_\beta = e^{-\epsilon s} \sum_\mu c_\mu r_\mu^l s^{m_\mu} t^{n_\mu},$$

where r the inter-electronic distance, $s = r_1 + r_2$ and $t = r_1 - r_2$. The convergence of the Hylleraas function is quite slow, but for small systems, like that of helium above, excellent results can be obtained. The Padé-Jastrow factor, however, is more suited for larger systems. It is constructed to include both electron-electron and electron-nucleus correlations and takes an exponential ansatz, namely e^U, with, see ref. [15],

$$U(r_i, r_{ij}) = \sum_{i=1}^{N} \left(\frac{\sum_{k=1}^{m_\alpha} \alpha_k r_i^k}{1 + \sum_{k=1}^{m_\alpha} \alpha'_k r_i^k} \right) + \sum_{j>i=1}^{N} \left(\frac{\sum_{k=1}^{m_\beta} \Delta_{ij} \beta_k r_{ij}^k}{1 + \sum_{k=1}^{m_\beta} \beta'_k r_{ij}^k} \right).$$

Here the parameter α_1 describes the behavior as r_i limits zero, the different α_k and α'_k describe the behavior in the intermediate distances, and the ratio $\alpha_{m_\alpha}/\alpha'_{m_\alpha}$ the long distance limit. The different β_k and β'_k similarly describe the electron-electron correlations.

The Padé-Jastrow factor must be constructed to include the cusp-conditions described in the previous section. Failing to incorporate these conditions would lead to divergences in the short distance limits, and optimizing the parameters will therefore give a poor result. The Δ_{ij} term is included to satisfy the different cusp conditions for parallel or anti-parallel spin electrons,

$$\Delta_{ij} = \begin{cases} 1/2 \text{ for } spin(i) \neq spin(j) \\ 1/4 \text{ for } spin(i) = spin(j) \end{cases}.$$

Another Jastrow-factor frequently encountered in the literature is the Schmidt-Moskowitz function e^U, with U given by (ref. [17])

$$U_{ij} = \sum_k \Delta(m_k, n_k) c_k (\bar{r}_i^{m_k} \bar{r}_j^{n_k} + \bar{r}_j^{m_k} \bar{r}_i^{n_k}) \bar{r}_{ij}^{o_k}.$$

Here $\bar{r} = r/(1 + \alpha r)$, c_k are the expansion coefficients, m_k, n_k and o_k are integers that may be set to include electron-electron ($m_k = n_k = 0$, $o_k \geq 1$), electron-nucleus (either $m_k \neq 0$ or $n_k \neq 0$, or both $m_k, n_k \neq 0$, and $o_k = 0$) and finally electron-electron-nucleus correlations (either $m_k \neq 0$ or $n_k \neq 0$, or both $m_k, n_k \neq 0$ and $o_k \neq 0$). To maintain consistency with Boys and Handy, ref. [15], the term, ref. [17],

$$\Delta(m_k, n_k) = \begin{cases} 1, \text{ for } m \neq n \\ 1/2, \text{ for } m = n \end{cases},$$

is included. Furthermore, to satisfy the electron-electron cusp condition for unlike spins, the only term with $o = 1$ is with both $m = n = 0$. Similarly, to satisfy the nuclear cusp-condition, additional terms with $n = 1$ or $m = 1$ are not included.

3.3.7 Efficiency

The efficiency, η, of a MC integration process is inversely proportional to the time, $\tau_{s.e.}$, it takes to obtain a result with a given standard error, $s.e.$,

$$\eta \propto \frac{1}{\tau_{s.e.}}.$$

The amount of time spent in obtaining a given accuracy is the product of the time it takes to perform one MC cycle, τ_{MC}, with the number of MC steps needed, $N_{s.e.}$

$$\tau_{s.e.} = N_{s.e.}\tau_{MC}.$$

The standard error of an MC estimate is given by

$$s.e. = \frac{\sigma}{\sqrt{M}},$$

where M is the number of samples. Therefore, the number of steps $N_{s.e.}$ needed is again proportional to the square of the variance of the integrand, namely

$$N_{s.e.} \propto \sigma^2.$$

Finally, the variance is related to the un-correlated variance, σ_u^2, obtained through the Metropolis algorithm, and the auto-correlation time κ, through eq. (3.30). We arrive at an efficiency proportional to

$$\eta \propto \frac{1}{\tau_{MC}\kappa\sigma_u^2} \tag{3.34}$$

Therefore, there are three ways to improve the efficiency of the VMC integration: (i) reduce the time it takes to perform one MC cycle, (ii) reduce auto-correlation effects and (iii) reducing the variance.

The first way to improve the efficiency (i) was discussed in sections 4.2 and 4.3. The bottle-neck of each MC cycle is the evaluation of the wave-function and both its gradient and its Laplacian. The main concern here was the Slater determinant, but as the energy and variance optimization schemes were introduced, efficient evaluation of the Jastrow-factor became increasingly important as well. For larger systems like for example molecules, some of the one-particle orbitals may have negligible overlap. Therefore given an instantaneous electronic configuration, the negligible parts of the Slater determinant may be identified and removed from the calculation of the wave-function and its derivatives. Also, if the Jastrow-factor is given only as function of the distances between the different particles, a requirement is that with increasing distance the corresponding terms in the Jastrow factor should approach a constant. These terms will not contribute to the local energy because both the first and second derivatives of these terms are almost zero. Such an approach could therefore, for large molecules and similar systems, result in a method that is linear with respect to the evaluation of the integrand, without violating the Pauli principle.

The second way to improve the efficiency (ii) is to reduce the auto-correlation effects. If the individual samples are correlated, it implies that overhead calculations are performed without providing additional information about the fluctuations of the integrand. Reducing the auto-correlations is therefore essential in the further development of the program code.

The third way to improve the efficiency (iii) regards choosing good trial-wave function. The better the wave-function represents the true eigenstate, the lower the variance. The trial wave-function must both include the local cusp conditions of two-body or higher interaction as well as the global trends of the eigenfunction.

In addition, trial wave-function optimization is essential when regarding the overall efficiency of the VMC method, not just the efficiency of the integration process. In the next two sections, approaches for optimizing trial wave-functions are studied.

3.3.8 Energy and Variance Optimization

We will here discuss two approaches for optimizing the trial wave-function. When optimizing we seek the parameter configuration of the trial wave-function Ψ_T that best approximates the behavior of the true eigenfunction Ψ_n. For the true ground state Ψ_0 a natural choice is to optimize with respect to energy minimization. This scheme, which is in accordance with the variational principle, is known as *energy optimization*. Optimization with respect to variance is another natural choice in that for every eigenfunction, not just the ground state, the variance of the local energy vanishes. This means that a *variance optimization* scheme could be applied in finding any eigenstate, because we know in advance that the value of the variance should be zero.

The energy optimization scheme is based on the variational principle (introduced in section 2.3),

$$\langle E_L \rangle = \frac{\int \Psi_\alpha^* \hat{H} \Psi_\alpha d\tau}{\int |\Psi_\alpha|^2 d\tau} \geq E_0.$$

Here the fact that the variational energy provides an upper bound to the ground state energy is exploited. The parameters $\alpha = \{\alpha_1, \alpha_2, \dots\}$, which correspond to a global energy minimum, are chosen as the best fit to the true ground state wave-function. A drawback in this seemingly brilliant scheme, is instability problems. In section 5.1 an example where the energy optimization fails is given.

In the literature there is a consensus that the variance optimization scheme provides the most stable approach, and a thorough investigation is given by Kent, ref. [12]. The

straightforward estimate of the un-correlated variance[4] is

$$\sigma_u^2 = \frac{1}{M-1}\sum_{i=1}^{M}\left(E_{L,i} - \langle E_L\rangle\right)^2,$$

where $\langle E_L\rangle$ is the average of the individual samples $E_{L,i}$. Many have chosen instead to optimize,

$$\sigma_d^2 = \frac{1}{M-1}\sum_{i=1}^{M}\left(E_{L,i} - E_{ref}\right)^2, \tag{3.35}$$

where E_{ref} is taken to be as close to the expected average of the local energy as possible.

3.3.9 Correlated Sampling

Wave-function optimization is one of the most critical, time consuming and important stages of a VMC calculation. In VMC calculations, the accuracy of the trial wave-function limits the statistical efficiency of the calculation and the final accuracy of the result obtained. Therefore, several variational parameters are put into the trial wave-function. As more and more parameters are put into the wave-function the accuracy needed to obtain statistically significant improvements becomes more demanding and time-consuming. We wish of course to limit the number of parameters by choosing the trial functions as wisely as possible, but as the systems grow larger the number of parameters needed is increasing.

The straightforward approach to optimize the parameters numerically, is to use well established statistical tools to fit a surface to a set of data-points chosen by the user. The minimum of the surface can then be obtained. This procedure, however, is not very efficient. First, the data points are statistical and we therefore need several (or a few very accurate) data points to be able to significantly pinpoint a parameter minimum. Further, we must choose the shape of the surface. Close to the minimum, a parabolic surface would be a good approximation, but as we do not know where the minimum is we must use intuition and insight to choose the shape of the surface. We want a procedure that is fast and able to localize the minimum without much effort. Therefore, we have incorporated an optimizing procedure commonly used in the literature known as *correlated sampling*. Introduction of *guiding functions*, $\Psi_{\alpha'}$, allows the same random walk to produce several *local* estimates of the integral,

$$\langle E\rangle_{\alpha'} = \frac{\int |\Psi_{\alpha'}(\mathbf{X})|^2 E_L^{\alpha'}(\mathbf{X}) d\tau}{\int |\Psi_{\alpha'}(\mathbf{X})|^2 d\tau}.$$

Each of these local estimates of the energy $\langle E\rangle_{\alpha'}$ must be in the neighborhood of the *central* parameter set α in parameter space. By the central parameter set we mean the set that

[4] The auto-correlation time κ is a constant, so optimization with respect to either the variance $\sigma^2 = \kappa\sigma_u^2$ or the un-correlated variance σ_u^2 are equivalent.

produces the random walk by means of the Metropolis algorithm. Multiplication of

$$1 = \frac{|\Psi_\alpha(\mathbf{X})|^2}{|\Psi_\alpha(\mathbf{X})|^2}$$

inside the integrals of both the numerator and the determinator yields

$$\langle E \rangle_{\alpha'} = \frac{\int \omega_{\alpha,\alpha'}(\mathbf{X}) E_L^{\alpha'}(\mathbf{X}) |\Psi_\alpha(\mathbf{X})|^2 d\tau}{\int \omega_{\alpha,\alpha'}(\mathbf{X}) |\Psi_\alpha(\mathbf{X})|^2 d\tau},$$

with

$$\omega_{\alpha,\alpha'}(\mathbf{X}) = \frac{|\Psi_{\alpha'}(\mathbf{X})|^2}{|\Psi_\alpha(\mathbf{X})|^2}.$$

By dividing with the norm,

$$\mathcal{N}_\alpha = \int |\Psi_\alpha(\mathbf{X})|^2 d\tau,$$

in both the numerator and the determinator we have

$$\langle E \rangle_{\alpha'} = \frac{\int \omega_{\alpha,\alpha'}(\mathbf{X}) E_L^{\alpha'}(\mathbf{X}) \rho_\alpha(\mathbf{X}) d\tau}{\int \omega_{\alpha,\alpha'}(\mathbf{X}) \rho_\alpha(\mathbf{X}) d\tau},$$

with

$$\rho_\alpha(\mathbf{X}) = \frac{|\Psi_\alpha(\mathbf{X})|^2}{\int |\Psi_\alpha(\mathbf{X})|^2 d\tau}. \tag{3.36}$$

Here $\rho_\alpha(\mathbf{X})$ is the probability distribution of the central parameter set. The random walk of the central parameter set may therefore be used to generate estimates of several local variations in parameter space. We arrive at

$$\langle E \rangle_{\alpha'} \approx \frac{\sum_{i=1}^{M} \omega_{\alpha,\alpha'}(\mathbf{X}_i) E_L^{\alpha'}(\mathbf{X}_i)}{\sum_{i=1}^{M} \omega_{\alpha,\alpha'}(\mathbf{X}_i)},$$

where the sample points are taken from the distribution $\rho_\alpha(\mathbf{X})$ given by eq. (3.36).

This approach, in theory, looks very promising, but in fact it poses a few problems. The weights $\omega_{\alpha,\alpha'}$ may vary by several orders of magnitude, especially close to the nodes. The sample points generated by the Metropolis algorithm depends only on the central wave-function. If the value of the central wave-function is small compared to the local wave-function, it implies that the value of the weight becomes large. This manifests itself near the nodes due to lack of more complicated many-body correlations. This could lead to a few sample points dominating the estimate of the integral. These few dominant points

may give really poor estimates of for example the energy, as the trial wave-functions fail to cancel divergent terms. Also, if the nodes of the local variation do not coincide with the nodes of the central wave-function we may actually allow sampling at the nodes!

Nevertheless, the introduction of guiding functions allows a fast and effective routine for optimizing the wave-function. A thorough investigation of the numerical instabilities induced by the introduction of guiding functions is given by Kent in ref. [12]. In this thesis Kent argues the use of variance optimization with the weights set to unity. This reduces the instability due to the few dominating weights, without changing the optimized parameters considerably. An example is given in section 5.1.

3.3.10 Diffusion Monte Carlo

Diffusion Monte Carlo (DMC) is a method of solving for the ground state of the many-body Schrödinger equation. In principle DMC could solve this equation exactly, but in practice it must be approximated. The DMC method is based on rewriting the Schrödinger equation in imaginary time by defining $\tau = it$ (ref. [12]),

$$\frac{\partial \Psi}{\partial \tau} = -\hat{H}\Psi. \tag{3.37}$$

As there is no time-dependency[5] in the Hamiltonian, a formal solution to the imaginary time Schrödinger equation is

$$\Psi(\tau + \delta\tau) = e^{-\hat{H}\delta\tau}\Psi(\tau),$$

where the state Ψ evolves from an imaginary time τ to a later time $\tau + \delta\tau$. By expanding the initial state $\Psi(\tau)$ in the true eigenstates Φ_i the final state becomes

$$\Psi(\tau + \delta\tau) = \sum_i^\infty c_i e^{-\epsilon_i \delta\tau}\Phi_i,$$

with ϵ_i the eigenenergies and c_i the expansion coefficients. Over time the term with the lowest eigenvalue will dominate. Hence, any initial state Ψ that is not orthogonal to the ground state Φ_0, will evolve to the ground state, that is

$$\lim_{\delta\tau\to\infty} \Psi(\tau + \delta\tau) = c_0 e^{-\epsilon_0\tau}\Phi_0. \tag{3.38}$$

In the DMC method the imaginary time evolution results in excited states decaying exponentially fast, whereas in the VMC method they remain. The ground state of eq. (3.38) also decays exponentially fast, and this problem is solved by introducing a constant offset to the energy $E_T \approx \epsilon_0$ in the Hamiltonian. If the Hamiltonian is separated into the kinetic

[5] We still seek the stationary solution, so the imaginary equations above has nothing to do with the 'real' time propagation of the wave-function.

energy and potential energy terms, the imaginary time Schrödinger equation takes a form similar to a diffusion equation,

$$\frac{\partial \Psi(\mathbf{X},\tau)}{\partial \tau} = \left[\sum_i^N \frac{1}{2}\nabla_i^2 \Psi(\mathbf{X},\tau)\right] + (E_T - V(\mathbf{X}))\Psi(\mathbf{X},\tau). \tag{3.39}$$

By considering a population of walkers $\mathbf{X}_i$ at a given time τ, where $\Psi(\mathbf{X}_i,\tau)$ is the density of walkers at $\mathbf{X}_i$, the interpretation of eq. (3.39) becomes clear. The first term on the right hand side is simply a diffusion term that tends to drive the walkers from dense areas to less populated areas. The second term describes a *branching* process. A branching process induces a growth of walkers when positive and a decay of walkers when negative. The method given by eq. (3.39) is in theory exact, but it leads to a very inefficient algorithm. The Coulomb potential $V(\mathbf{X})$ is unbounded and therefore the rate term $E_T - V(\mathbf{X})$ can diverge. This results in a very large fluctuation of walkers, and gives large statistical errors.

Importance sampling is a way of efficiently reducing these large fluctuations, and is essential for a DMC simulation to be efficient. A trial or guiding wave function $\Psi_T(\mathbf{X})$, which closely approximates the ground state wave function, is introduced. The optimized trial wave-function of VMC calculations are typically used in this respect. A new distribution $f(\mathbf{X},\tau)$ is defined as

$$f(\mathbf{X},\tau) \equiv \Psi_T(\mathbf{X})\Psi(\mathbf{X},\tau). \tag{3.40}$$

This new distribution is a solution of the modified Schrödinger equation

$$\frac{\partial f(\mathbf{X},\tau)}{\partial \tau} = \frac{1}{2}\nabla\left[\nabla - F(\mathbf{X})\right] f(\mathbf{X},\tau) + (E_T - E_L(\mathbf{X}))f(\mathbf{X},\tau), \tag{3.41}$$

where the *force-term* F is given by

$$F(\mathbf{X}) = \frac{2\nabla\Psi_T(\mathbf{X})}{\Psi_T(\mathbf{X})},$$

and where the local energy E_L is defined as before

$$E_L(\mathbf{X}) = -\frac{1}{\Psi_T(\mathbf{X})}\frac{\nabla^2\Psi_T(\mathbf{X})}{2} + V(\mathbf{R}). \tag{3.42}$$

The first term of eq. (3.41) is still a diffusion term. The second term caused by the force F is a *drift* term; a drift in a diffusional process drives the population according to the drift-force. The branching term is now in a form suited for Monte Carlo. The branching effect is greatly reduced compared to the branching of eq. (3.39). In the limit $\Psi_T = \Psi_0$ and $E_T = \epsilon_0$ there is no branching at all. The branching is simply a result of the trial wave-function failing to duplicate the ground state, and because of the inaccuracy of the trial energy. This last effect can be accounted for in the DMC method, as we will come back to at the end of this section.

Insertion of eq. (3.40) into eq. (3.41) reproduces eq. (3.39), and therefore, eq. (3.41) is apparently exact. However, there are problems associated with the introduction of the distribution f. At the nodes of the trial wave-function this term vanishes. This implies that the nodes are fixed by the form of the trial wave-function, and will limit the accuracy of the DMC algorithm. Furthermore, both the force of the drift term and the local energy in the branching term are extremely sensitive to the trial wave-function in the close proximity to these nodes. Failing to incorporate physical effects into Ψ_T thus leads to inaccuracies in the solutions. Still, DMC provides a very good method for solving the many-body system, and excellent results can be obtained even by using quite simple guiding-functions, at least for small systems. The ultimate accuracy depends solely on how well the trial wave-function duplicates the behavior of Φ_0.

The above interpretation of eq. (3.41) poses a problem. The diffusion and drift term move the population of walkers around, whereas the branching either produces or removes walkers. These two effects cannot both be performed simultaneously, and in practice we need to go to the limit of short time and perform successive diffusion-drift and branching processes. This approximation is established through the use of Green's functions. The details are left to the reader, see for example ref. [15], but we will summarize some of the basic implications. The diffusion and drift terms of eq. (3.41) reduce to the Green's function

$$\tilde{G}_{diff}(\mathbf{Y},\mathbf{X},d\tau) = (2\pi d\tau)^{-3n/2} e^{-(\mathbf{X}-\mathbf{Y}-d\tau F(\mathbf{Y})/2)^2/2d\tau},$$

where we move from $\mathbf{X}$ to $\mathbf{Y}$. Detailed balance must be imposed, because

$$\tilde{G}_{diff}(\mathbf{Y},\mathbf{X},d\tau) \neq \tilde{G}_{diff}(\mathbf{X},\mathbf{Y},d\tau),$$

which is achieved by the following Metropolis acceptance

$$A(\mathbf{Y},\mathbf{X},d\tau) \equiv min\left(1,\ \frac{|\Psi(\mathbf{Y})|^2}{|\Psi(\mathbf{X})|^2}\frac{\tilde{G}_{diff}(\mathbf{X},\mathbf{Y},d\tau)}{\tilde{G}_{diff}(\mathbf{Y},\mathbf{X},d\tau)}\right).$$

The branching is realized by destruction and generation of walkers. The branching *rate* is given by (ref. [15])

$$\tilde{G}_B(\mathbf{Y},\mathbf{X},d\tau) = e^{-([E_L(\mathbf{X})+E_L(\mathbf{Y})]/2-E_T)\tau},$$

which ignores the path of the random walk. Nevertheless, as $\tau \to 0$ $\tilde{G}_{diff}\tilde{G}_B$ converges to the exact Green's function. To get a form suited for Monte Carlo, the rate is written as a sum of an integer r and fraction $0 \leq \delta r < 1$, namely

$$\tilde{G}_B(\mathbf{Y},\mathbf{X},d\tau) = r + \delta r.$$

The population of walkers are then updated by generation of a random number $\mathcal{X}$, between zero and one, for each walker. If $\mathcal{X} \leq \delta r$ the current walker is copied to $r+1$ walkers,

and if $\mathcal{X} > \delta r$ the current walker becomes r walkers. For example if the rate of a walker is 0.36 and $\mathcal{X} = 0.70$ the walker is removed, and if the rate is 2.12 and $\mathcal{X} = 0.10$ two extra duplicates of the walker are created.

The diffusion-drift and branching process is only valid in the limit $d\tau \to 0$, whereas the stationary solution to the imaginary time Schrödinger equation is obtained only in the limit $d\tau \to \infty$. This is realized by first starting with a population of walkers that represents the true ground state in the best possible way, for example selected equilibrated walkers from a VMC run. Then the imaginary time evolution $d\tau$ is taken to be a small number, for example $d\tau = 1/1000$, and the population is evolved till it equilibrates.

Finally, the trial energy E_T is updated in accordance with the population changes. If the total population is growing, reduce the energy, and vice versa. As the DMC walkers equilibrate, the average of this fluctuating energy provides an excellent estimate to the ground state energy.

Chapter 4

Implementation

In this chapter we will outline the program structure, and we will demonstrate the flexibility as well as the limitations of our implementation. The program code developed in this thesis consists of about 7000 lines and more than 25 different classes. We therefore limits the description of the code, and focus on the key building-blocks of the VMC algorithm. The full code is provided as an appendix, in the hope that the comments therein, and the description provided in this chapter, can help clearify the different aspects of the VMC implementation.

Several computers may work for days solving even simple looking physical problems. Therefore, a great deal of effort must be put into the design and development of the numerical algorithms. Three major concepts concerning the implementation of a large program are

1. Speed - the computational time (CPU).
2. Flexibility - the ability to handle many different special cases.
3. Readability - how easy it is to understand the program (and make modifications).

To acquire a fast running program the number of calculations must be kept at a minimum. This often requires storing a lot of information so that the same calculations are not performed multiple times.

In order to obtain a flexible program, the program should consist of many small building blocks that can easily be replaced by other similar blocks. We want the blocks to work independently of each other. In this way changing one block will not affect the others. But, at the same time we want to keep the flow of data at a minimum.

The program structure should also be easy to read. Often, modifications of existing programs need to be done, either by the programmers themselves, or by other users. This task can be very time consuming, especially if the program is difficult to understand. The algorithms and data structure need to be well documented, and the variables and building

blocks should have names that are logical.

When making a large program the three criteria cannot all be met to a full extent, and compromises have to be made. The two most important parts of our program are the Slater determinant part and the part including correlation effects. Our first concern will be the program speed. Second, is the need for flexibility. We want to test different one-particle orbitals in the Slater determinant as well as different Jastrow-factors for the correlation. The unfortunate effect of reducing the CPU time and increasing the flexibility is the program becoming less transparent.

Generating a good code for scientific purposes is a continuous process, and the final program may differ considerably from the original outline. Numerical tests of different algorithms gives insight along the way, and the data structure is modified accordingly. Another problem is interconnected data; the data of one block often depends on the changes done in a different block. Changing data in one block can therefore have consequences in a completely different part of the program. As a result, good routines to control the data-flow should be established.

4.1 Structure of the QMC Program

Our first concern is to recognize how we want the program to operate, and to work out a program structure. We divide the program into blocks, with each block performing a few specific tasks. Figure 4.1 illustrates the principal building blocks of our main program. These blocks are again divided into smaller blocks as will be outlined in this section. All input is handled by the *Domain* class. The user-specified input tells the program what to do. In our program the input consists of several files that are associated with different parts of the code. Here the trial wave-function is chosen and variational parameters are set. The input specifies whether we will optimize the variational parameters with respect to energy or variance optimization. Several other variables are also chosen, such as the acceptance ratio, number of thermalization steps, number of cycles, whether we allow interchange of spin between electrons, and the name of the output file[1].

In the Metropolis test the ratio

$$\frac{|\Psi(\mathbf{X}^{new})|^2}{|\Psi(\mathbf{X}^{old})|^2}.$$

is needed. When evaluating the local energy we also need the first and second derivatives. These three factors constitutes the most time-demanding part of the VMC algorithm. In this thesis the trial wave-function we want to optimized have the general form

$$\Psi_\alpha = D_\alpha^\uparrow D_\alpha^\downarrow G_\beta. \tag{4.1}$$

[1] When we are running parallel computations by MPI, we generate several output-files.

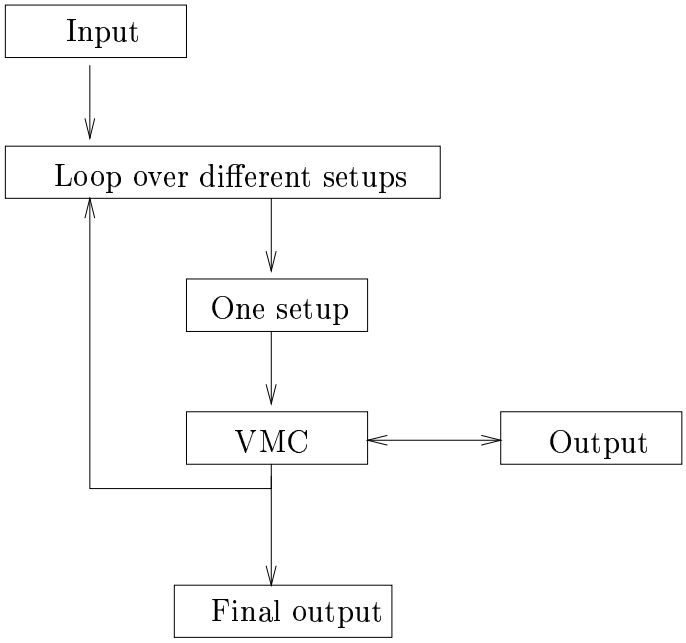

Figure 4.1: Variational Monte-Carlo program structure

This allow us to treat the Slater determinant part and the part concering the Jastrow-factor separately, which is seen because

$$\frac{\Psi_\alpha(\mathbf{X}^{new})}{\Psi_\alpha(\mathbf{X}^{old})} = \frac{D_\alpha^\uparrow(\mathbf{X}^{new})}{D_\alpha^\uparrow(\mathbf{X}^{old})} + \frac{D_\alpha^\downarrow(\mathbf{X}^{new})}{D_\alpha^\downarrow(\mathbf{X}^{old})} \frac{G_\beta(\mathbf{X}^{new})}{+} G_\beta(\mathbf{X}^{old}), \tag{4.2}$$

and

$$\frac{\nabla\Psi_\alpha}{\Psi_\alpha} = \frac{\nabla D_\alpha^\uparrow}{D_\alpha^\uparrow} + \frac{\nabla D_\alpha^\downarrow}{D_\alpha^\downarrow} + \frac{\nabla G_\beta}{G_\beta}, \tag{4.3}$$

and finally

$$\frac{\nabla^2\Psi_\alpha}{\Psi_\alpha} = \frac{\nabla^2 D_\alpha^\uparrow}{D_\alpha^\uparrow} + \frac{\nabla^2 D_\alpha^\downarrow}{D_\alpha^\downarrow} + \frac{\nabla^2 G_\beta}{G_\beta}. \tag{4.4}$$

4.2 Optimizing the Slater Determinant

Determining a determinant of an $N \times N$ matrix by standard Gaussian elimination is of the order of $\mathcal{O}(N^3)$ calculations. As there are $N \cdot d$ independent coordinates we need to evaluate $N \cdot d$ Slater determinants for the gradient and $N \cdot d$ for the Laplacian[2].

[2] In this thesis three-dimensional atoms will be considered. However, the program allows varying numbers of dimensions d and we keep this notation for generalization.

We therefore need $2d \cdot N + 1$ evaluations of the Slater determinant. These evaluations may be reduced by an efficient evaluation of the Slater determinant as described in [15]. Instead of working with the $2d \cdot N + 1$ matrices separately we perform most of the manipulations on the inverse of the Slater Matrix.
By changing the position of only one particle at the time, we may limit the number of calculation to update the inverse matrix to approximately N^3 per electron, or approximately N^4 calculations for all electrons.

First, define the Slater matrix to be

$$D_{ij}(\mathbf{x}) \equiv \phi_j(x_i).$$

The inverse of the Slater matrix is the adjoint of the cofactors C_{ji} divided by the determinant of the Slater matrix D_α, i.e.

$$D_{ij}^{-1} = \frac{C_{ji}}{D_\alpha}. \tag{4.5}$$

By definition, the Slater matrix and its inverse must satisfy the relation

$$\sum_{k=1}^{N} D_{ik}(\mathbf{x}) D_{kj}^{-1}(\mathbf{x}) = \delta_{ij}. \tag{4.6}$$

The inverse matrix may therefore be used to evaluate the determinant in the usual cofactor expansion. Consider the fraction of two such expansions

$$\frac{D_\alpha(\mathbf{y})}{D_\alpha(\mathbf{x})} = \frac{\sum_{j=1}^{N} D_{ij}(\mathbf{y}) C_{ij}(\mathbf{y})}{\sum_{j=1}^{N} D_{ij}(\mathbf{x}) C_{ij}(\mathbf{x})}, \tag{4.7}$$

with $\mathbf{y}$ and $\mathbf{x}$ two electron configurations that are equal except for the coordinates of electron i. The cofactors $C_{ij}(\mathbf{y})$ and $C_{ij}(\mathbf{x})$ are independent of the coordinates of electron i and therefore equal. This observation combined with eq. (4.5) yield

$$\frac{D_\alpha(\mathbf{y})}{D_\alpha(\mathbf{x})} = \frac{\sum_{j=1}^{N} D_{ij}(\mathbf{y}) C_{ij}(\mathbf{x})}{\sum_{j=1}^{N} D_{ij}(\mathbf{x}) C_{ij}(\mathbf{x})} = \frac{\sum_{j=1}^{N} D_{ij}(\mathbf{y}) D_{ji}^{-1}(\mathbf{x})}{\sum_{j=1}^{N} D_{ij}(\mathbf{x}) D_{ji}^{-1}(\mathbf{x})}.$$

By applying eq. (4.6) we arrive at

$$\frac{D_\alpha(\mathbf{y})}{D_\alpha(\mathbf{x})} = \sum_{j=1}^{N} D_{ij}(\mathbf{y}) D_{ji}^{-1}(\mathbf{x}), \tag{4.8}$$

for the ratio. In a similar manner the derivatives are computed, see ref. [15],

$$\frac{\nabla_i D_\alpha(\mathbf{x})}{D_\alpha(\mathbf{x})} = \sum_{j=1}^{N} \nabla\phi_j(x_i) D_{ji}^{-1}(\mathbf{x}), \tag{4.9}$$

and

$$\frac{\nabla_i^2 D_\alpha(\mathbf{x})}{D_\alpha(\mathbf{x})} = \sum_{j=1}^{N} \nabla^2\phi_j(x_i) D_{ji}^{-1}(\mathbf{x}). \tag{4.10}$$

Therefore, all the determinant derived quantities required at each step of the walk can be computed using the single inverse matrix D^{-1}.

The new inverse matrix is evaluated *only* if the move from $\mathbf{x}$ to $\mathbf{y}$ is accepted. It is possible to update the inverse matrix one column at a time, rather than recomputing the entire matrix. Using eq. (4.8), the ratio of the determinant at $\mathbf{y}$ to the determinant at $\mathbf{x}$ is

$$R \equiv \sum_{j=1}^{N} D_{ij}(\mathbf{y}) D_{ji}^{-1}(\mathbf{x}). \tag{4.11}$$

If this move is rejected by the Monte Carlo algorithm, then no further computations are needed. If the move is accepted, the new inverse matrix is related to the old inverse matrix by, see ref. [15],

$$D_{kj}^{-1}(\mathbf{y}) = \begin{cases} D_{kj}^{-1}(\mathbf{x}) - \dfrac{1}{R} D_{ki}^{-1}(\mathbf{x}) \displaystyle\sum_{l=1}^{N} D_{il}(\mathbf{x}) D_{lj}^{-1}(\mathbf{x}) & \text{for } j \neq i, \\ \dfrac{1}{R} D_{ki}^{-1}(\mathbf{x}) & \text{for } j = i. \end{cases} \tag{4.12}$$

To compute the initial inverse matrix, start with the identity matrix and replace each column one at a time by using eqs. (4.11) and (4.12).

The two spin-degrees of freedom allow us to split the Slater matrix in two separate matrices

$$D = D_\uparrow \cdot D_\downarrow,$$

reducing the number of calculations by up to a factor of eight; for systems with equal number of spin-up and spin-down electrons we now have two matrices that each need $(N/2)^3 = N^3/8$ calculations to update their inverses. When moving one electron only one of these need to be updated, thereof the factor 1/8. Each matrix is treated independently, but have the same structure. The relation

$$R = \frac{D(\mathbf{y})}{D(\mathbf{x})} = \frac{D_\uparrow(\mathbf{y}) \cdot D_\downarrow(\mathbf{y})}{D_\uparrow(\mathbf{x}) \cdot D_\downarrow(\mathbf{x})}$$

follows from the orthogonality of the spin-up and spin-down states. We see that when we move only one particle the ratio R is either $D_\uparrow(\mathbf{y})/D_\uparrow(\mathbf{x})$ or $D_\downarrow(\mathbf{y})/D_\downarrow(\mathbf{x})$.

The class *SlaterMatrix* uses the algorithms of eqs. (4.11) and (4.12) to calculate the ratio and to update the inverse matrix. The two matrices are sewn together in the *SlaterDet* class. The *SlaterDet* class needs access to the variational form of the one-particle orbitals. These orbitals are defined in the *SingleParticlesFuncs* class. Arrays of these single-particle orbitals represents one column of either $D_\uparrow$ or $D_\downarrow$, and are defined through the class *FuncSetMultivar*. One such array depends on the spatial coordinates of one particle. When performing numerical derivation we need only to differentiate the array of orbitals corresponding to the particle we have moved. This recipe allows for an easy update of the numerical derivatives, since only the terms involving the coordinates of electron i change as we move the i'th electron. We have used the following approximations of the first and second derivatives

$$\frac{\delta\phi_j(\mathbf{x_i})}{\delta\xi_i} = \frac{\phi_j(\xi_i+h)-\phi_j(\xi_i-h)}{2h} + \mathcal{O}(h^2), \tag{4.13}$$

and

$$\frac{\delta^2\phi_j(\mathbf{x_i})}{\delta\xi_i^2} = \frac{\phi_j(\xi_i+h)+\phi_j(\xi_i-h)-2\phi_j(\xi_i)}{h^2} + \mathcal{O}(h^2). \tag{4.14}$$

Here the coordinates of electron i is given as $\mathbf{x_i} = (\xi_i^1, \xi_i^2, \ldots, \xi_i^d)$ with d the number of dimensions.

Which *SingleParticleFuncs* to be used by the class *FuncSetMultivar* are defined in the class *FuncUp* and *FuncDown*. We illustrate how this is coded by three examples.

A first approximation to the many-electron atom is to use the hydrogen-like orbitals (of section 2.2) where we allow variation of one parameter α. We may code for example the $1s$ orbital as

```
template <class Param>
class Hydr1s : public SingleParticleFunc<Param> {
 protected:
  virtual inline double phi(Param& coordinate) {
    return exp(- param[0] * coordinate.r());
  }
 public:
  Hydr1s() {}
  virtual ~Hydr1s() {}
};
```

Here the function phi(Param& coordinate) returns the value of the one-particle $1s$ orbital. It is assumed that the class Param has an algorithm r() that returns the electron-nucleus distance. The variational parameter param[0] is defined through the *Domain* class, and attached to the *Hydr1s* class within the class *FuncUp* of *FuncDown*. Similarly, the generic function

```
virtual inline double phi(Param& coordinate) {
  double result = 0;
  for (int i=0; i<numCoeff; i++)
    result += coefficients[i] * solidHarmonics()(coordinate,i)
      * STOfuncs()(coordinate,i);
  return result;
}
```

returns the value of a HF orbitals. Here the coefficients , as well as the parameters needed for the STO function, are given by the input-files handled by *Domain*. There are no logical loops testing which function to be used once they have been assigned in the *FuncUp* and *FuncDown* classes. An example of how this is implemented in the *FuncUp* class is

```
virtual void init(Domain* domain, int alphaVar) {
  int index = 0;
  ifstream ifile;
  ifile.open(fixedParamsUp);
  if (domain->getOrbitalType() == "Hydrogen") {
    if (domain->getUp1s())  {function[index]=new Hydr1s<Param>;  index
       ++;}
    if (domain->getUp2s())  {function[index]=new Hydr2s<Param>;  index
       ++;}
    if (domain->getUp2px()) {function[index]=new Hydr2px<Param>; index
       ++;}
    if (domain->getUp2py()) {function[index]=new Hydr2py<Param>; index
       ++;}
    if (domain->getUp2pz()) {function[index]=new Hydr2pz<Param>; index
       ++;}
    if (index != len )
      cerr << "Error creating FuncUp!\nNot matching dimensions of
         number"
           <<" particles spin up (" << len << ") and number of input"
           << " functions\n";
    for (int i=0; i<len; i++) {
      function[i]->attach(domain->getNumAlpha(),
                          domain->getAlphaParam(alphaVar));
    }
  }
  else if (domain->getOrbitalType() == "HartreeFock") {
    for (int i=0; i<len; i++) {
      function[i] = new HF<Param>;
      function[i]->readHFparams(ifile);
    }
  }
  ifile.close();
}
```

As can be seen from this example the *Domain* class manages the information of which orbitals are to be used and which configuration is studied. Because the HF orbitals are generic they are more easily implemented in this class than the hydrogen orbitals. Making the hydrogen orbitals generic as well can be achieved by combining a library of the hydrogen radial functions with the already existing library of the solid harmonics[3].

Generation of additional single-particle orbital sets should be implemented using a combination of radial functions and the solid harmonics. The solid harmonics may be used whenever the potential is spherical symmetric, which is the case for atoms. The solid harmonics are implemented in the class *SolidHarmonics* and *SolidHarmonicsFuncs*, and the Slater-type orbitals are implemented in *STOBasis* and *STOBasisFuncs*.

We have now outlined how the Slater determinant is created and optimized, and will continue with discussing how to use it. Care must be taken to perform all the different steps in the correct order. As the determinant is the most time-consuming part of our code we have omitted tests that remember what the program has already done, and it is assumed that the necessary computations and initializations are performed.

Only the *SlaterDet* class is used by other parts of the program to perform steps with respect to the Slater determinant. Information about ratios, derivatives and so on are also available through the *SlaterDet* class. Take a look back at figure 4.1. The input is handled by the *Domain* class. When one setup has been created, information and manipulations concerning the Slater determinant are all performed through the *SlaterDet* class. In this code only one such object is allowed. However, linear combinations of Slater-determinants should be implemented in the future to allow for example distinctions between states of different symmetry.

Table 4.1 lists some of the key algorithms of the *SlaterDet* class. When a *SlaterDet* object has been created and initialized, the three ratios are available. Because they are returned as references and pointers we do not have to copy the information they contain, but may access it directly. This can also be 'dangerous' because it allows a user to change this information outside the class, which should not be done unless the programmer really knows what this implies. Within the VMC algorithm we propose a move and then either accept or reject it. The *suggestMove()* algorithm suggests the move of one electron only. We want to move all the particles around to sample the entire space. The *setToNextParticle()* swaps which electron is the next to be moved. The particle to be considered must be in agreement with the particle considered in all other parts of the code. Also, in-between separate VMC runs the *initNewVmcRun()* routine must be called.

There are several other algorithms in the *SlaterDet* than the methods listed in table 4.1. To mention a few: allowing spin-flip, getting the pointers to determinants instead of the

[3] This has not been implemented in our code.

SlaterDet

Algorithm	**Usage**
SlaterDet()	The empty constructor.
init(Domain* _domain, int alphaVar)	Initializes the *SlaterDet* object and associated objects with a *Domain* object (and which local set of parameters to be used)#.
initNewVmcRun()	Makes the Slater determinant ready for a VMC calculation.
setToNextParticle()	Increases *currentParticle* by one$.
suggestMove()	Suggest a move of *currentParticle*.
acceptMove()	Accepts move.
rejectMove()	Rejects move.
calcDiffRatios()	Calculates the gradient divided by the determinant.
calcDDiffRatios()	Calculates the Laplacian divided by the determinant.
double& getRatio()	Returns the reference to the ratio.
double* getDiffRatiosPtr()	Returns the pointer to the gradient.
double& getDDiffRatio()	Returns the reference to the Laplacian.

This is to choose between the different local wave-functions needed for energy or variance optimization.
$ This routine is circular in that it moves from the last electron to the first, the first to the second and so on.

Table 4.1: Some of the key algorithms of the class *SlaterDet*.

ratios and getting the pointers to the individual spin up and spin down matrices.

4.3 Optimizing the Correlation

The Slater determinant is the most time-consuming part of one single VMC run, but in practice it is often the correlation part of the wave-function, the Jastrow-factor, that uses most of the CPU time. The Slater-determinant and the Jastrow-factor may be treated independently as the total wave-function is a product of the two. When we optimize the wave-function we introduce guiding functions that duplicates the behavior of the wave-function in a neighbourhood of some initial parameter guess $\{\beta_0\}$. The introduction of guiding functions allow generation of several local variations of the wave-function in parameter space. When using for example the Hartree-Fock orbitals in the Slater determinant part, the determinant is fixed, and is only computed once for all local variations. The parameter variation will in this case only affect the Jastrow-factor, and for each local variation all Jastrow-factor derived values must be updated. As a consequence, this means that we evaluate one Slater determinant and several Jastrow-factors, in a practical optimization routine. As more parameters are added into the Jastrow-factor (to include additional physical effects) efficient evaluation of the Jastrow-factors becomes increasingly important.

There are several similarities between how we treat the correlation and how we treat the Slater determinant. We need the ratio of the new wave-function to the old. In addition, we need the two ratios of the gradient and the Laplacian to the determinant for sampling of the local energy. Furthermore, we must propose a move for the Metropolis algorithm, and either reject or accept it. The way we store data and update it is somewhat different though, as we will demonstrate in this and the following section.

We assume the following form of the correlation part of the wave-function: the correlation G is given by either J or e^J, where J is given by

$$J = \sum_{i=0}^{N-1} \sum_{j>i=0}^{N-1} f_{ij}, \tag{4.15}$$

and where

$$f_{ij} = f(r_i, r_j, r_{ij}). \tag{4.16}$$

The above form is common in the literature, see for example [15, 12], and is constructed to include additional electron-electron, electron-nucleus and electron-electron-nucleus correlation.

The program is made to allow different number of dimensions d. In this thesis we will only study three-dimensional atoms. However, changing to two dimensions in order to study of for example a quantum dot, is easily done. The program is coded to allow different number

of dimensions and we define ξ_i^k as coordinate i of electron k. The inter-electronic distance is thus given by

$$r_{ij} = \sqrt{\sum_{k=0}^{d-1}(\xi_i^k - \xi_j^k)^2}, \tag{4.17}$$

and the nucleus-electron distance is

$$r_i = \sqrt{\sum_{k=0}^{d-1}(\xi_i^k)^2}. \tag{4.18}$$

Before we are ready to discuss the correlation part of the wave-function we must first establish the routines for handling these distances. The nucleus-electron distances were used to calculate the Slater determinant orbitals. These are handled by the *CoorSpinDiff* class. This class keeps track of the coordinates, the spin and the distance to the nucleus of one electron. Also, it contains the Cartesian differences $\xi_i^k + h$ and $\xi_i^k - h$ as well as the radial differences $r_i(\xi_i^k + h)$, $r_i(\xi_i^k - h)$. There are N *CoorSpinDiff* objects, coors[i], representing the different electrons, and one *CoorSpinDiff* object, trialCoors[0], containing the trial-coordinate of the single electron suggested moved in the Metropolis algorithm. The values are only updated with a suggested move, and if the move is accepted all values of trialCoors[0] are copied into the coors[currentParticle] where currentParticle is the particle currently being moved. These coordinates are handled by, and are accessed through, the *Domain* class.

We now turn our attention to the inter-electronic distances of eq. (4.17). The two classes, *Distance* and *DistanceDiff*, keep track of these distances. For *Distance* the matrix of the distances r_{ij} is given by:

$$r_{ij} = \begin{bmatrix} 0 & r_{01} & r_{02} & \cdots & r_{0i} & \cdots & \cdots & r_{0(N-1)} \\ r_{01} & 0 & r_{12} & \cdots & r_{1i} & \cdots & \cdots & r_{1(N-1)} \\ r_{02} & r_{12} & 0 & \ddots & \vdots & \cdots & \cdots & \vdots \\ \vdots & \vdots & \ddots & 0 & r_{(i-1)i} & \cdots & \cdots & \vdots \\ r_{0i} & r_{1i} & \cdots & r_{(i-1)i} & 0 & r_{i(i+1)} & \cdots & r_{i(N-1)} \\ \vdots & \vdots & \vdots & \vdots & r_{i(i+1)} & 0 & \ddots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & 0 & r_{(N-2)(N-1)} \\ r_{0(N-1)} & r_{1(N-1)} & \cdots & \cdots & \cdots & \cdots & r_{(N-2)(N-1)} & 0 \end{bmatrix}. \tag{4.19}$$

This information is stored in a blitz matrix interElectronicDistances[4]. We could of course

[4] The blitz library is easy to use and well suited for matrix manipulations. Consult for example http://www.oonumerics.org/blitz/ for documentation and downloading the blitz c++ library.

store only the upper-triangular matrix or the lower-triangular matrix of eq. (4.19). However, for consistency with the *DistanceDiff* class, we have chosen the above form.
With a proposed move of electron i, only $N-1$ values need to be recalculated;

$$|r_{0i}, r_{1i}, \ldots, r_{(i-1)i}, 0, r_{i(i+1)}, \ldots, r_{i(N-1)} > . \tag{4.20}$$

These values are stored in both trialColumn and trialRow. With an accepted move, the row and the column of interElectronicDistances corresponding to the particle moved are exchanged with the values given in trialColumn and trialRow, respectively.

Distance and *DistanceDiff* are quite similar, except *DistanceDiff* allows a variation of a coordinate. For the *DistanceDiff* class, the matrix interElectronicDistances

$$r_{ij}^{+} = \begin{bmatrix} 0 & r_{01}(\xi_1^+) & r_{02}(\xi_2^+) & \cdots & r_{0(N-1)}(\xi_{(N-1)}^+) \\ r_{01}(\xi_0^+) & 0 & r_{12}(\xi_2^+) & \cdots & r_{1(N-1)}(\xi_{(N-1)}^+) \\ r_{02}(\xi_0^+) & r_{12}(\xi_1^+) & 0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 & r_{(N-2)(N-1)}(\xi_{(N-1)}^+) \\ r_{0(N-1)}(\xi_0^+) & r_{1(N-1)}(\xi_1^+) & \cdots & r_{(N-2)(N-1)}(\xi_{(N-2)}^+) & 0 \end{bmatrix}. \tag{4.21}$$

stores all the values of the differentiated distances where one of the d Cartesian coordinates of particle i is incremented, $\xi_i^+ = \xi_i + h$. We see from eq. (4.17) that the following equality holds,

$$r_{ij}(\xi_j + h) = r_{ij}(\xi_i - h).$$

We thus have that the negative differences are the adjoint of the positive differences, or

$$r_{ij}^{-} = (r_{ij}^{+})^T. \tag{4.22}$$

All the information needed of the inter-electronic distances for numerical calculations of both the first and second derivatives, are held by one object of class *Distance* and d objects of class *DistanceDiff*. Movement of one electron consists of $(2d+1)(N-1)$ updates of the inter-electronic distances, and is therefore of order $\mathcal{O}(N)$.

Our treatment of both the nucleus-electron and the electron-electron distances has now been outlined, and we are ready to turn our attention to the correlation part of the wave-function. We have limited our attention to correlations of the form of either $G = J$ or $G = e^J$ where

$$J(\mathbf{x}) = \sum_{j>i=1}^{N-1} g(r_i, r_j, r_{ij}). \tag{4.23}$$

Remember that the total atomic wave-function is the product of a Slater-determinant and a correlation part. By multiplying the determinantal part with a Jastrow-factor, G, one

hopes to include most of the correlation induced by the approximate form of the Slater determinant. As will be seen in the next chapter much of the correlation is in this way included. However, the Jastrow-factor chosen is limited to include corrections solely depending on the inter-particle distances. But the mean-field approach of a HF-calculation, for example, does not only induce correlations of this form. So however good the Jastrow-factor is, we cannot gain all the correlation. This is due to the approximate form of the Slater-determinant. The Slater-determinant generates a set of fixed nodes in the Nd-dimensional space[5]. The Jastrow-factor alone is unable to move these nodes around. Also, correlation effects that are not only dependent of the nodal structure or on the inter-particle distances are possible. We will not go into detail here, but emphasize that the form given by eq. (4.23), although quite good, has limitations. Furthermore, the VMC calculation is usually meant simply as a starting point for a DMC calculation, and the above form is very good in combination with DMC. However, for DMC the nodal problem needs to be addressed for highly accurate calculations.

When we move one particle only the $N-1$ terms that involve the particle moved are changed. By storing the data in a fashion similar to that of the inter-electronic distances given by eq. (4.19), we may update only these values and then use the stored values for calculating the ratio

$$R = \frac{G^{new}}{G^{old}}.$$

The square of the ratio is needed by the Metropolis algorithm, and is easily updated. For $G = J$ we have

$$R = \frac{J^{old} + \Delta J}{J^{old}}$$

and for $G = e^J$ we have

$$R = e^{\Delta J},$$

with

$$\Delta J = \sum_{k=0}^{i-1} g_{ki}^{new} - g_{ki}^{old} + \sum_{k=i+1}^{N-1} g_{ik}^{new} - g_{ik}^{old}.$$

With $G = J$ the gradient of G is given by

$$\nabla G = \nabla J = \nabla \sum_{j>i=0}^{N-1} g(r_i, r_j, r_{ij}). \tag{4.24}$$

The gradient is given by

[5] At the nodes the determinant equals zero.

$$\nabla = \left[\frac{\delta}{\delta \xi_0^0}, \frac{\delta}{\delta \xi_0^1}, \ldots, \frac{\delta}{\delta \xi_i^k}, \ldots, \frac{\delta}{\delta \xi_{n-1}^{d-1}} \right].$$

For each Nd terms in the gradient of $G = J$ there are $N-1$ terms given by

$$\frac{\delta J}{\delta \xi_i^k} = \sum_{l>m=0}^{N-1} \frac{\delta}{\delta \xi_i^k} g(r_l, r_m, r_{lm}).$$

The only surviving terms inside the double sum are

$$\frac{\delta J}{\delta \xi_i^k} = \sum_{j=0}^{i-1} \frac{\delta}{\delta \xi_i^k} g(r_j, r_i, r_{ji}) + \sum_{j=i+1}^{N-1} \frac{\delta}{\delta \xi_i^k} g(r_i, r_j, r_{ij}).$$

For the numerical derivatives we arrive at

$$\begin{aligned} \frac{\delta J}{\delta \xi_i^k} =& \frac{1}{2h} \sum_{j=0}^{i-1} \left\{ g(r_j, r_i(\xi_i^{k^+}), r_{ji}(\xi_i^{k^+})) - g(r_j, r_i(\xi_i^{k^-}), r_{ji}(\xi_i^{k^-})) \right\} \\ &+ \frac{1}{2h} \sum_{j=i+1}^{N-1} \left\{ g(r_i(\xi_i^{k^+}), r_j, r_{ij}(\xi_i^{k^+})) - g(r_i(\xi_i^{k^-}), r_j, r_{ij}(\xi_i^{k^-})) \right\}. \end{aligned} \tag{4.25}$$

where $\xi_i^{k^+} = \xi_i^k + h$ and $\xi_i^{k^-} = \xi_i^k - h$. Following the same procedure for the second derivatives we arrive at

$$\begin{aligned} \nabla^2 J = \sum_{k=0}^{d-1} \sum_{i=0}^{N-1} \Bigg[& \frac{1}{h^2} \sum_{j=0}^{i-1} \left\{ g(r_j, r_i(\xi_i^{k^+}), r_{ji}(\xi_i^{k^+})) + g(r_j, r_i(\xi_i^{k^-}), r_{ji}(\xi_i^{k^-})) - 2g(r_j, r_i, r_{ji}) \right\} \\ & + \frac{1}{h^2} \sum_{j=i+1}^{N-1} \left\{ g(r_i(\xi_i^{k^+}), r_j, r_{ij}(\xi_i^{k^+})) + g(r_i(\xi_i^{k^-}), r_j, r_{ij}(\xi_i^{k^-})) - 2g(r_j, r_i, r_{ij}) \right\}. \Bigg] \end{aligned} \tag{4.26}$$

With $G = e^J$ we have

$$\frac{\nabla G}{G} = \nabla J \tag{4.27}$$

and

$$\frac{\nabla^2 G}{G} = \nabla^2 J + (\nabla J)^2, \tag{4.28}$$

which is straightforward. By using eqs. (4.25) and (4.26) we have the necessary expressions for the first and second derivatives of the correlation part of the wave-function. These expressions are handled similar to the way we treated the inter-electronic distances. We have

$$g_{ij}^{+}=\begin{bmatrix} 0 & g_{01}(\xi_1^+) & g_{02}(\xi_2^+) & \dots & \dots & g_{0(N-1)}(\xi_{(N-1)}^+) \\ g_{01}(\xi_0^+) & 0 & g_{12}(\xi_2^+) & \dots & \dots & g_{1(N-1)}(\xi_{(N-1)}^+) \\ g_{02}(\xi_0^+) & g_{12}(\xi_1^+) & 0 & \ddots & \dots & \vdots \\ \vdots & \vdots & \ddots & 0 & \dots & g_{(N-2)(N-1)}(\xi_{(N-1)}^+) \\ g_{0(N-1)}(\xi_0^+) & g_{1(N-1)}(\xi_1^+) & \dots & \dots & g_{(N-2)(N-1)}(\xi_{(N-2)}^+) & 0 \end{bmatrix}, \tag{4.29}$$

with $g_{ij}(\xi_j^+) = g(r_i, r_j(\xi_j + h), r_{ij}(\xi_j + h))$ where ξ_j is one of the three Cartesian coordinates of electron j. We don't have the equality $g_{ij}(\xi_j^+) = g_{ij}(\xi_i^-)$ as was the case for the distances. Therefore, we need a matrix g_{ij}^- similar to that of g_{ij}^+. This approach allow us to update only $4d(n-1)$ values of g_{ij} for the differences.

In our code the different g_{ij}'s and their differences are stored in the *Jastrow* and *JastrowDiff* classes, respectively. In addition to updating and storing the values of the g_{ij}'s the *Jastrow* class calculates

$$\Delta J = \sum_{k=0}^{i-1} g_{ki}^{new} - g_{ki}^{old} + \sum_{k=i+1}^{n-1} g_{ik}^{new} - g_{ik}^{old}.$$

needed for the ratio. The *Correl* class uses the values in *Jastrow* and *JastrowDiff* to find the ratio R needed by the Metropolis algorithm, and to find the two ratios $\nabla G/G$ and $\nabla^2 G/G$. All the information needed by the other parts of the program are accessible through the *Correl* class.

Table 4.2 lists some key algorithms of the *Correl* class. Comparing this table with the one for the Slater-determinants part of table 4.1, reveals several similarities. The routines to handle the Metropolis algorithm are almost the same, as well as the return statement of the different ratios. These ratios need only be assigned once during the program, and the calculateRatio() and calculateGradAndLaplacianRatios() algorithms updates the values of the corresponding ratios.

4.4 Parameter Optimization

The most important aspects of the VMC routine is that it enables the optimization of *any* trial wave-funciton. Due to the statistical nature of the induvidual VMC estimates, correlated sampling, see section 3.3.9, is implemented to allow a fast and efficient way to perform parameter optimization. In this section we will take a closer look at how this is realized in our code.

We recall that the introduction of guiding functions, $\Psi_{\alpha'}$, allow the same random walk to produce several local estimates of the integral,

Correl

Algorithm	**Usage**
Correl()	Empty constructor.
attach(Domain& _domain)	Attaches the domain object.
setToNextParticle()	Interchanges the studied particle.
suggestMove()	Suggests a move of one particle.
acceptMove()	Accepts the move.
rejectMove()	Rejects the move.
calculateRatio()	Calculates the ratio.
calculateGradAndLaplacianRatios()	Calculates the $\nabla G/G$ and $\nabla^2 G/G$.
getRatioPtr()	Returns the reference to the ratio.
getGradRatio()	Returns the pointer to $\nabla G/G$.
getLaplaceRatio()	Returns the reference to $\nabla^2 G/G$.

Table 4.2: Some of the key algorithms of the class *Correlation*.

$$\langle E \rangle_{\alpha'} = \frac{\int |\Psi_{\alpha'}(\mathbf{X})|^2 E_L^{\alpha'}(\mathbf{X}) d\tau}{\int |\Psi_{\alpha'}(\mathbf{X})|^2 d\tau}.$$

The random walk here samples the probability distribution of the central wave-function Ψ_α. This resulted in

$$\langle E \rangle_{\alpha'} \approx \frac{\sum_{i=1}^{M} \omega_{\alpha,\alpha'}(\mathbf{X}_i) E_L^{\alpha'}(\mathbf{X}_i)}{\sum_{i=1}^{M} \omega_{\alpha,\alpha'}(\mathbf{X}_i)},$$

where $\langle E \rangle_{\alpha'}$ is the local estimate of the energy, and where the weights were defined by

$$\omega_{\alpha,\alpha'}(\mathbf{X}) = \frac{|\Psi_{\alpha'}(\mathbf{X})|^2}{|\Psi_\alpha(\mathbf{X})|^2}.$$

The trial wave-function optimized in this thesis have the general form

$$\Psi_\alpha = D_\alpha^\uparrow D_\alpha^\downarrow G_\beta,$$

This allow us to separate the variations of α and the variations of β, see eqs. (), () and (). Therefore, we can treat the n_α variational parameters α of the Slater determinant and the n_β variational parameters β of the Jastrow factor independently. This fact is exploited in the *Variations* class. In this class the *SlaterDet* and *Correl* objects needed for the local variations are generated, and all the required updates of these objects performed; such

as calculating the induvidual ratios and updating the gradient and the Laplacian ratios. Each of the different local variations are objects of the class *LocalWaveFunction*. These objects does not manipulate the *SlaterDet* and *Correl* objects at all, but simply extract information (needed to sample for example the kinetic energy) from the two associated *SlaterDet* and *Correl* objects. In the *LocalWaveFunction* objects the samples of for example the local energy, the variance of local energy, etc., are accumulated for statistical analysis. Furthermore, the blocking scheme described in section 3.3.4 is incorporated here.

To give an example of the above, consider that we have a Slater determinant and a Jastrow-factor each with one variational parameter, and we want to find the minimum of a surface consisting of a rectangular grid in parameter space. The rectangular grid consists of three variations of the Slater determinant parameter α, and five variations of the correlation parameter β, which means that we have a total of 15 grid points. We then create three *SlaterDet* objects and five *Correl* objects in the *Variations* object. The random walk of the Metropolis algorithm is conducted only for one of these 15 wave-functions, and we take it to be the central one. Each of the 15 local variations are assigned one *SlaterDet* object and one *Correl* object, corresponding the the grid generated by the parameters, and the value of the central wave-function to calculate the weights. In this way we are able to construct 15 variations of the wave-function by only calculating the properties of three Slater determinants, and five Jastrow-factors.

The code is implemented to allow any number of variational parameters, and the above procedure is generic; it may be used to create any (odd) number of local variations of each individual parameter.

The parameter optimization schemes implemented in our code allow the parameter search to pinpoint the parameter minimum with little effort from the user. This is realized by a procedure we have called *uni-directional movement*. In this approach we start by a rough parameter grid, and moves to the minimum (of either the energy or the variance estimates) on this grid. This minimum forms the central parameters for another optimization, and we continue this procedure as long as walk in parameter space is in one general direction; we stop the procedure only when all the parameters have been moved in both directions (or they have ended up where they started at least one of the VMC runs). When one uni-directional is stopped, the grid is refined, and the number of Monte Carlo cycles are increased to narrow down the parameter minimum. This procedure may be performed any number of times, specified by the user.

4.5 Parallel Computataion

Almost all of the calculations in this thesis were performed on parallel machines. Extension of the VMC algorithm to parallel machines requires very few changes, as the algorithm is naturally parallel. To parallelise the VMC algorithm it is sufficient to note that provided

care is taken to ensure each calculation uses a different random number sequence, the algorithm can be parallelised by performing independent VMC calculations on each node of a parallel machine. Communication between the nodes is only necessary to obtain global averages and is therefore negligible: the parallel efficiency is essentially 100%, and the calculation can theoretically exploit any number of nodes without loss of efficiency. For this purpose we have used the LAM/MPI. The general LAM web site can be found at: *http://www.lam-mpi.org/*.

Chapter 5

Results

In this chapter we start by testing whether the program code can reproduce known solutions, and proceed by testing the energy and variance optimization schemes. Then the results regarding the atomic problem are presented and estimates of the computational time dependency with number of particles are exemplified. The next and final step is to suggest how to increase the efficiency of the code.

5.1 Testing the code

Regarding the development of any numerical tool, the first concern is to test whether the code has been correctly implemented. This is usually done by considering special cases where the solutions are already known.

The Slater-determinant part of the code is validated by reproduction of the HF results. This is realized by setting the Jastrow-factor equal to unity, and using the Roothaan Hartree-Fock wave-functions optimized by Clementi and Roetti, see ref. [16]. In table 5.1 the results for some atoms are listed, and these are in excellent agreement with the HF energies. This indicates that the Slater determinant part of the code works the way it is supposed to.

The correlation part was tested for a two-particle quantum dot. This test was performed by a fellow student, ref. [18], and was in agreement with the analytical solution, ref. [19].

Having tested the two main building blocks of our program, we turn our attention to the two different optimization schemes. Both the energy and the variance optimization schemes are applied to a few simple helium atom trial wave-functions. When introducing the variational approach to the helium atom (in section 2.3) we started out by a trial wave-function which is a product of two hydrogen-like $1s$ orbitals, namely

$$\psi_\alpha = e^{-\alpha(r_1+r_2)}. \tag{5.1}$$

Test of VMC Slater determinant

Atom	HF	VMC
He	-2.8617	-2.8618(1)
Li	-7.4327	-7.4325(11)
B	-24.5290	-24.534(12)
C	-37.6886	-37.599(50)
Ne	-128.547	-128.551(19)
Ar	-526.817	-526.84(15)

Table 5.1: VMC results for a Roothaan Hartree-Fock wave-function Slater determinant, compared with the exact HF results of ref. [16].

Listed in table 5.2 are the results of the energies optimized both through energy and variance optimization. Also listed is the energy for $\alpha = 2$ which is in excellent agreement with the exact result -2.75 (see section 2.3). The energy optimization performed to obtain the optimized result of ψ_α is given step by step in figure 5.1. This figure shows how we move from the initial guess of $\alpha = 2$ and relax to the value $\alpha = 1.5875$. The energy estimate for $\alpha = 1.5875$ is $\langle E_\alpha \rangle = -2.8365 \pm 0.0004$, which is in good, though not excellent, agreement with the analytical energy minimum $\langle E_\alpha \rangle = -2.85$ for $\alpha = 1.6875$ (see section 2.3). In general, the energy optimization scheme cannot be trusted completely. We will discuss this shortly, but first we look at an extension to the trial wave-function ψ_α.

In Quantum Monte Carlo methods, a common extension to wave-functions consisting of one-particle orbitals is to include a Jastrow-factor. We select a one-parameter Padé-Jastrow (introduced in section 3.3.6) for a first approximation, and now have two parameters α and β to optimize,

$$\psi_{\alpha,\beta} = e^{-\alpha(r_1+r_2)} exp(\frac{r_{ij}}{2(1+\beta r_{ij})}). \tag{5.2}$$

In table 5.2 the results of the two different optimization schemes to this function are listed. Even though the wave-function is simple, these results are quite good. By comparison, the HF result given by ref. [16], is -2.8617, and the 'exact' result as given by ref. [15] is -2.9037.

Different results are obtained for the two individual optimization schemes. The trial wave-functions, ψ_α and $\psi_{\alpha,\beta}$, fail to incorporate the electron-nucleus cusp condition for $\alpha \neq 2$. This results in an energy divergence when one of the electron-nucleus distances approaches zero. On the other hand, the electron-electron repulsion shields some of the nucleus attraction. Therefore, the 'tails' of the orbitals should be stretched, indicating a lower value of α. Lowering α results in larger divergences near the nucleus, but at the same time it gives

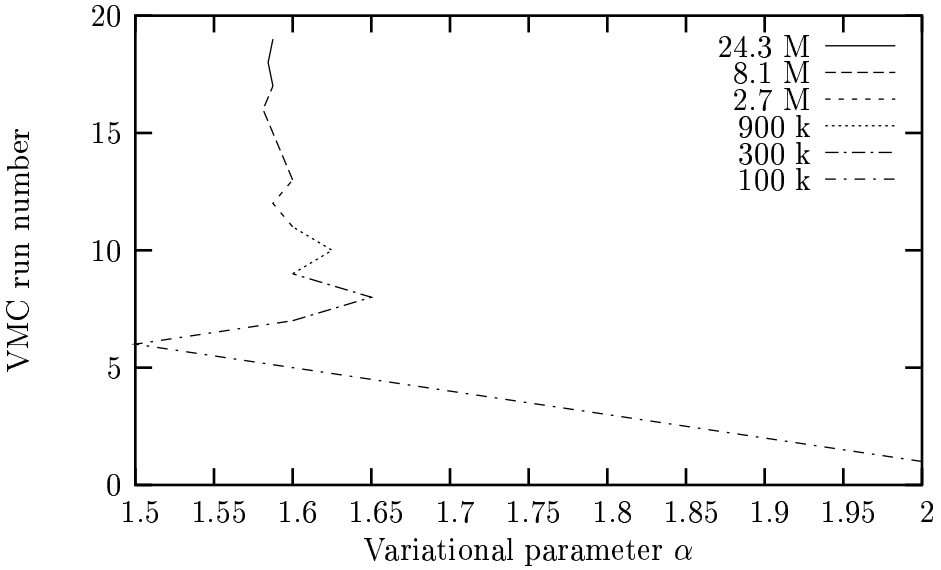

Figure 5.1: Figure illustrating the step-by-step energy optimization routine. The plot was made optimizing ψ_α, given by eq. (5.1), with respect to energy. We start out by an initial guess (VMC run number 0) for the parameter α and move to the lowest energy estimate of the three points $\alpha - \delta\alpha, \alpha, \alpha + \delta\alpha$. Here the walker is generated by the central wave-function ψ_α, and the energy estimates of the two local variation $\psi_\alpha - \delta\alpha$ and $\psi_\alpha + \delta\alpha$ are generated using correlated sampling. We continue the process until the parameter movement is no longer in the same general direction, and then increase the number of cycles and reduce the variation $\delta\alpha$ to narrow down the minimum search. This process is continued a total of six times; the number of MC cycles is increased (from 100 000 to 24 300 000) and the parameter variation reduced (from 0.08 to 0.0025).

Helium Hydrogenic Trial-Functions

Wave-function	α	β	**VMC result**
ψ_α	2.0		-2.7504(4)
ψ_α	1.5875		-2.8365(4)$^{(a)}$
ψ_α	1.9625		-2.7707(4)$^{(b)}$
$\psi_{\alpha,\beta}$	1.71	0.6625	-2.8800(2)$^{(a)}$
$\psi_{\alpha,\beta}$	1.9525	0.39	-2.8778(2)$^{(b)}$

Table 5.2: VMC energies for He using different two different trial wave-functions ψ_α and $\psi_{\alpha,\beta}$, given by eqs. (5.1) and (5.2) respectively. Two different optimization schemes have been used; (a) energy optimization, (b) variance optimization.

a better representation of the wave-function at a distance from the nucleus. The energy optimization scheme therefore results in a lower value of α than the variance optimization. This illustrates some of the differences of the two optimization schemes, but a more thorough investigation of the methods are needed. Two important questions arises; will we arrive at the same minimum if we conducted a new search, and do we move towards the actual minimum?

The answer to the first question is ambiguous, and is rather put as a reminder that consistency-tests *should* be provided before any trust can be put into the results. The optimization routine must be stable, which means that when applied to different starting points all must relax to some small confined subspace of the parameter space. A problem we often encountered was that the variation of the parameters were small compared to the accuracy of the energy estimates. Such an approach makes the routine relax to the wrong parameter set. At the same time, increasing the accuracy requirement reduces the efficiency of the optimization procedure. Therefore, some experience is needed for selecting the number of MC cycles, choosing the parameter step lengths etc.

The second question - do we move toward the 'true' minimum - is essential. The VMC energy optimization routine should, given a trial wave-function, find a set of parameters that are in the neighbourhood of the true energy minimum. Similarly, the variance optimization scheme should arrive at a parameter set near the true variance minimum. Finding a *wrong* parameter set *many* times does not make the procedure more correct.

The simulation of an energy optimization scheme is given in figure 5.2. Here the two parameters of $\psi_{\alpha,\beta}$, given by eq. (5.2) are optimized. As can be seen from figure 5.2 the values of the two parameters are narrowed down, and they relax to a small confined subspace of the parameter space. Unfortunately, the two parameters we obtain do not represent the energy minimum. In figure 5.3 accurate energy estimates are given for a few selected parameter pairs (α, β). The pairs have been selected from the parameter pairs generated by the energy optimization scheme.

Figure 5.3 indicates that the energy minimum is not the one obtained through the energy optimization routine. The lowest energy in this series is for the parameter pair $(\alpha, \beta) = (1.76, 0.54)$. In table 5.3, the parameter space in the close vicinity of this parameter set is investigated further.

Table 5.3 pinpoints the energy minimum of $\psi_{\alpha,\beta}$ to be in the close to $(\alpha, \beta) = (1.82, 0.40)$, which is significantly different from the result obtained through the energy optimization scheme (see table 5.2). These results give a clear indication that there is something wrong with the energy optimization procedure. This signifies the importance of using another approach.

We now take a look at the variance optimization scheme with weights set to unity (see

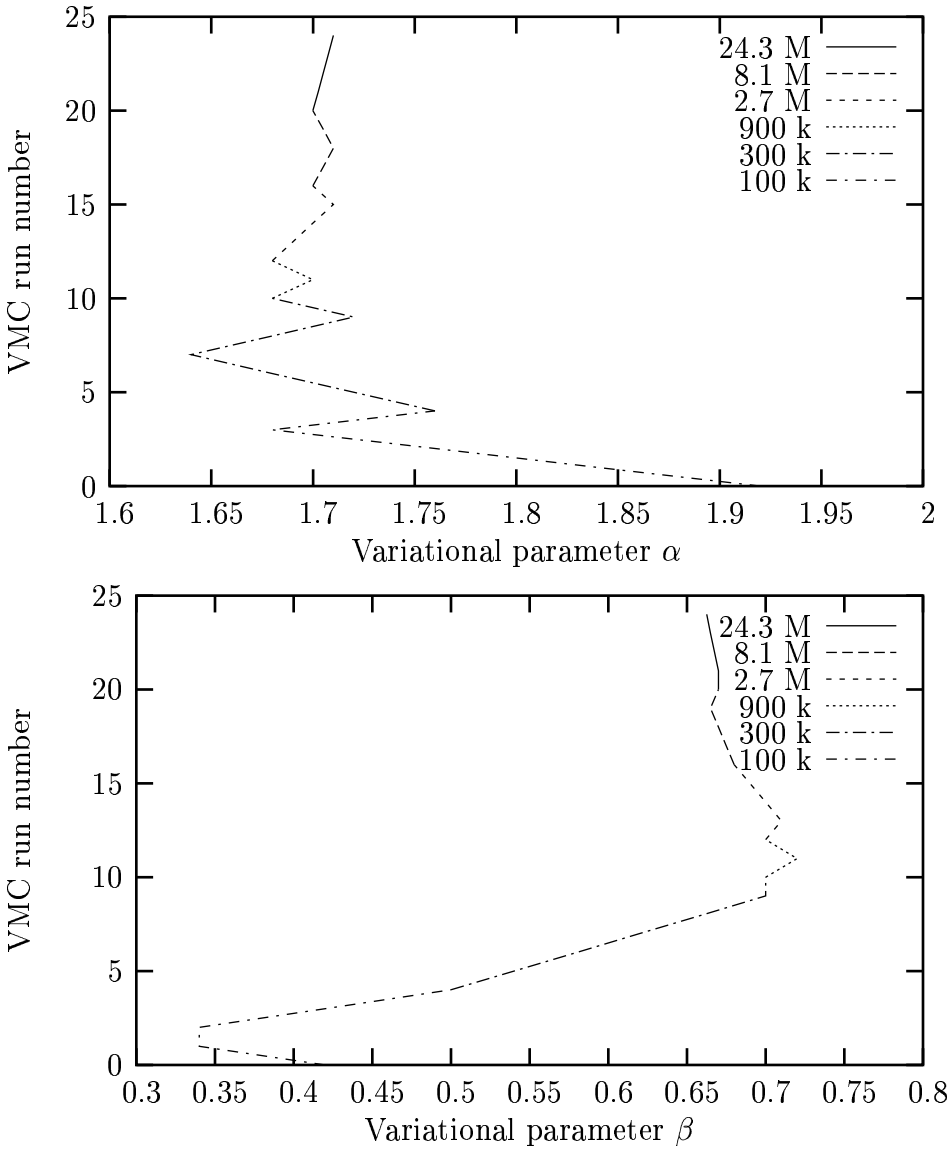

Figure 5.2: Figure illustrating the step by step parameter walk of the energy optimization routine. The parameter walk was generated by optimizing $\psi_{\alpha,\beta}$, given by eq. (5.2), with respect to energy. We start out by an initial guess (VMC run number 0) for the parameters (α, β) and move to the lowest energy estimate on the rectangle specified by the two corners $(\alpha - \delta\alpha, \beta - \delta\beta)$ and $(\alpha + \delta\alpha, \beta + \delta\beta)$, using correlated sampling. We continue the process until the general movement is no longer uni-directional. We then increase the number of cycles and reduce the variations $\delta\alpha$ and $\delta\beta$. This process is continued a total of six times; the number of MC cycles is increased (from 100 000 to 24 300 000) and the parameter variation reduced (from 0.08 to 0.0025).

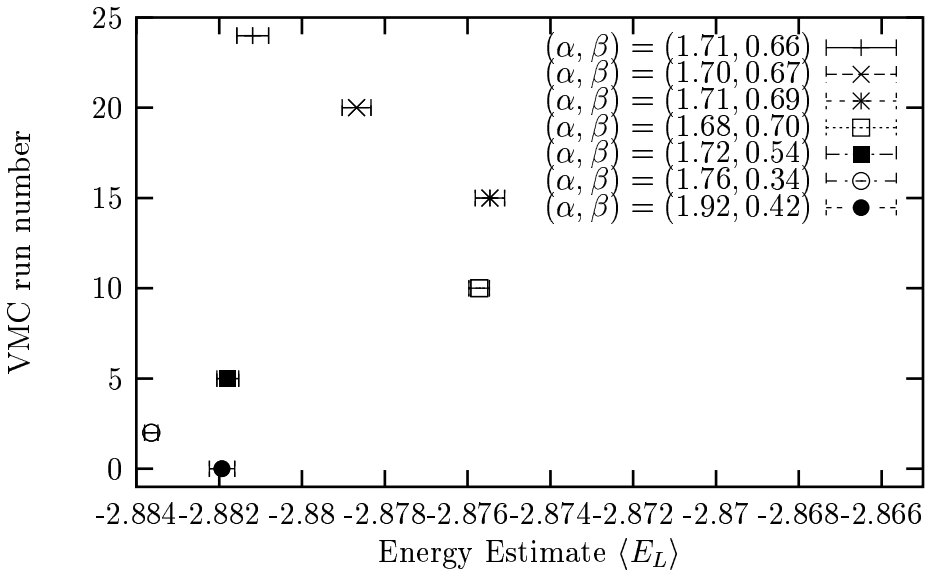

Figure 5.3: Accurate estimates of the energies of a few selected parameter pairs generated by the energy optimization algorithm for $\psi_{\alpha,\beta}$ given by eq. (5.2). The step by step optimization is depicted in figure 5.2.

Localizing the Actual Energy Minimum

$\alpha \backslash \beta$	0.30	0.34	0.38	0.40	0.42
1.72		-2.8762(4)			
1.76	-2.8819(2)	-2.8836(2)	-2.8855(2)		
1.80		-2.8883(1)	-2.8890(2)		-2.8893(2)
1.82				-2.8901(3)	
1.84			-2.8897(3)		-2.8896(2)

Table 5.3: VMC electronic energies for He using different parameter settings in the proximity of the minimum energy of figure 5.3.

section 3.3.8 and section 3.3.9). In figure 5.4 the variance optimization procedure for $\psi_{\alpha,\beta}$ is depicted.

Compared to the energy optimization scheme in figure 5.2 the variance optimization scheme pinpoints a parameter minimum with fewer VMC runs. The values of table 5.4 have been created by varying the parameters in the vicinity of the parameter minimum (obtained through variance optimization). These values give an indication that the parameters representing the variance minimum of $\psi_{\alpha,\beta}$ are in the close proximity of the parameters obtained through the optimization scheme. However, to obtain significant results a greater number of estimates are needed. Nevertheless, this result favors the variance optimization scheme over the energy optimization scheme. For additional investigation into the nature of the variance optimization scheme, consult for example ref. [12].

Localizing the Actual Variance Minimum

$\alpha\backslash\beta$	0.37	0.39	0.41
1.93		0.2676(8)	
1.95	0.2641(4)	0.2638(5)	0.2647(5)
1.97		0.2692(2)	

Table 5.4: Averaged Local energy variances for He using four different parameter settings in the proximity of the variance minimum obtained through the variance optimization procedure depicted in figure 5.4. The five individual points were averaged over 1 000 estimates of the (uncorrelated) standard error, where each estimate again was created using 100 000 MC cycles.

The results of this section indicates that the variance optimization scheme is well suited for optimization, whereas the energy optimization scheme fails to pinpoint the energy minimum. The minimum value of table 5.3, -2.8901 ± 0.0003, compared to the minimum value obtained through the energy optimization, -2.8800 ± 0.0002, clearly shows the inaccuracy of the energy optimization. Still, the variance optimization landed at an energy -2.8778 ± 0.0002. This indicates that, with regard to the optimization of $\Psi_{\alpha,\beta}$, we loose a significant amount of the obtainable correlation energy. The actual minimum of $\Psi_{\alpha,\beta}$ is approximately 30% closer to the 'exact' value -2.9037, ref. [15], than the results obtained through both of the optimization schemes. For obtaining the best energy both these methods fail, however, the energy in itself is usually not the principal result we are looking for. When VMC is used as a starting point for a DMC calculation, a low variance in the guiding function is favorable to a low energy. This is due to the reduction of the branching term of the diffusion equation, eq. (3.41). Furthermore, to incorporate the cusp conditions the variance optimization provides the best result (thereby limiting the variance due to the divergences). Therefore, for most purposes variance optimization is preferable.

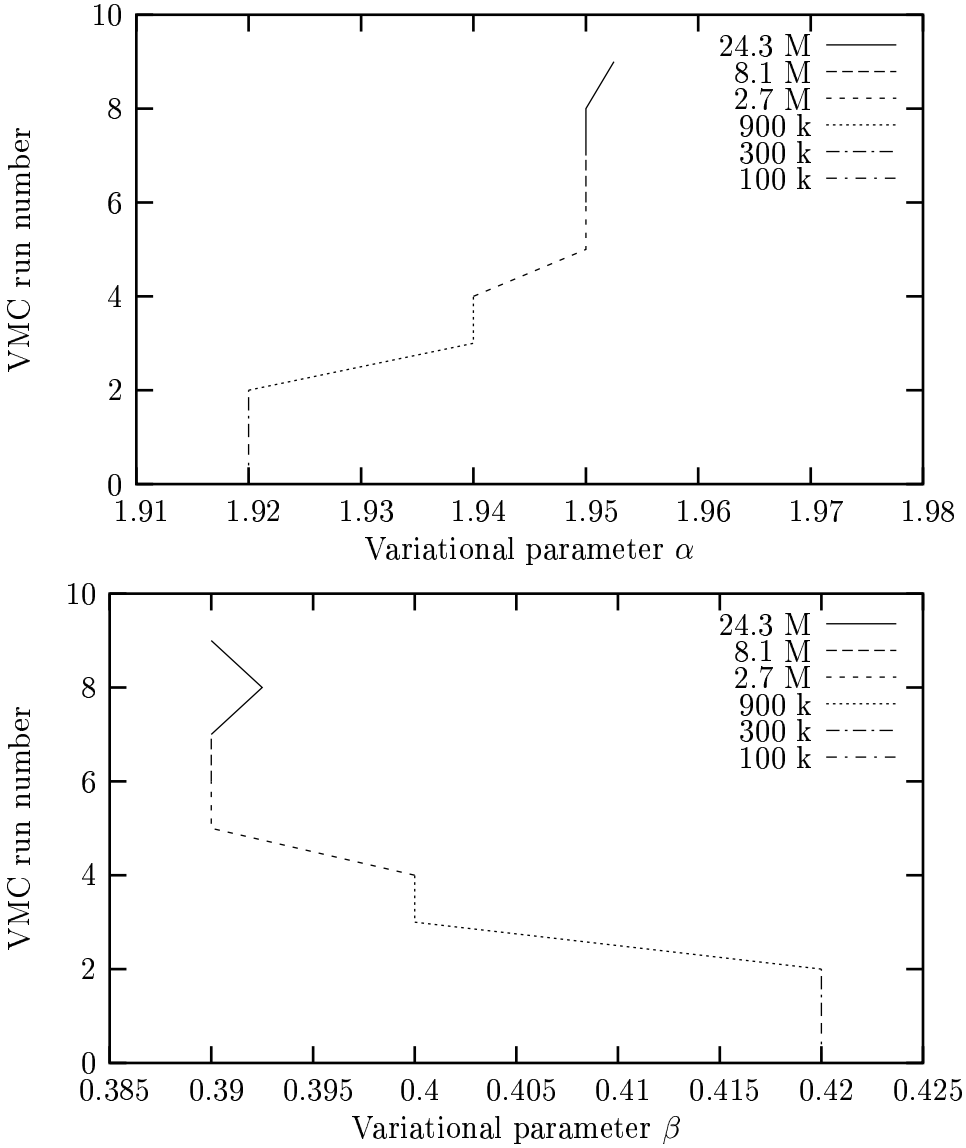

Figure 5.4: Figure illustrating the step-by-step parameter walk of the variance optimization routine. The plot was made optimizing $\psi_{\alpha,\beta}$ given by eq. (5.2) with respect to variance. We start out by an initial guess (VMC run number 0) for the parameters (α, β) and move to the lowest variance estimate on the rectangle specified by the two corners $(\alpha - \delta\alpha, \beta - \delta\beta)$ and $(\alpha + \delta\alpha, \beta + \delta\beta)$, using correlated sampling. We continue the process until the general movement in parameter space is no longer uni-directional. We then increase the number of cycles and reduce the variations $\delta\alpha$ and $\delta\beta$. This process is repeated a total of six times; the number of MC cycles is increased (from 100 000 to 24 300 000) and the parameter variation reduced (from 0.08 to 0.0025).

5.2 Atomic Results

In this section we present our results regarding the atomic problem. All the results presented in this section are generated using the fixed Roothaan Hartree-Fock determinants combined with variational electron-electron Padé-Jastrow factors. Table 5.5 lists the optimized electronic energies for the lithium atom.

Lithium

# of parameters	VMC	% CE
1	-7.4599(2)	59.9(4)
3	-7.4614(1)	63.2(3)
5	-7.4622(2)	65.0(4)

Table 5.5: VMC results for a Roothaan HF Slater determinant, ref. [16], multiplied with a one-, three- or five-parameter electron-electron Padé-Jastrow factor, ref. [15], for the Li ground state. The last column, % CE, is the difference between the HF energy -7.4327 and the VMC energy, relative to the difference between the HF and the exact -7.4781 result, see ref. [15].

Three different Padé-Jastrow factors have been optimized. These results indicate that the additional information gained by including additional terms in the electron-electron part of the Padé-Jastrow factor is small, but still important.

Table 5.6 lists the optimized electronic energies for several atoms, where we have optimized a one-parameter Padé-Jastrow factor. As indicated by this table, the effects of electron-electron correlations becomes less important with system size. To obtain even better wave-functions, a natural extension of the Jastrow factor is to include effects such as additional electron-nucleus and electron-electron-nucleus correlations. Adding such particle-particle correlations awaits further development.

The optimization procedure has not yet been parallelized and was performed on my 1.4 GHz laptop. The resulting optimized wave-functions were then used to create the results of table 5.6 by parallel computations on eighteen machines in the range 0.8-1.4 GHz. The longest run was argon that lasted approximately 10 hours and 30 minutes. This run generated 1.1 billion MC cycles. In a similar simulation for carbon, the run lasted about 9 hours and 30 minutes, and 36.3 billion MC cycles where created. If we wanted to create the same number of cycles for argon as for carbon, we would need to run the code

$$\frac{36.3 \cdot 10^9}{1.1 \cdot 10^9} \frac{10.5}{9.5} \approx 3.65$$

times longer for argon than for carbon. This suggests a dependency, with the number of electrons N, of the order $\mathcal{O}(N^{1.2})$, where the power 1.2 is found by taking the ratio $ln3.65/ln(18/6) \approx 1.2$.

Here each MC sample consists of a move of only one electron. When compared to moving all electrons at once, the information gained, by sampling the energies with each individual move, is reduced by an accuracy of approximately order $\mathcal{O}(N)$. This is because it takes the walker at least N MC cycles to reach a completely new configuration. Still the evaluation of all the required wave-function derived quantities is for this example of order $\mathcal{O}(N^{2.2})$. When including more electrons the order $\mathcal{O}(N^3)$ evaluation of the Slater determinant will eventually dominate. However, this result shows that in the range up to 18 particles we have reduced the order of the integrand evaluation by approximately a power 0.8. This reduction is considerable.

Our implemented code is very fast, but the efficiency may be greatly improved. Before we look at ways to improve the efficiency, we give an estimate of the actual order of the overall efficiency. Still, we look at the example of argon and carbon. In this example, the estimated standard errors were computed to be $s.e._{Ar} = 0.005094$ and $s.e._C = 0.0002817$ respectively. If we wanted the same accuracy for argon as for carbon we would need to to increase the number of cycles by a factor[1]

$$\left|\frac{s.e._{Ar}}{s.e._C}\right|^2 = 356.$$

In comparison with carbon we would therefore need to run the program $356 \cdot 10.5/9.5 = 403$ times longer. This implies a dependency with the number of electrons according to

$$\left|\frac{18}{6}\right|^p = 403,$$

which leads to $p \approx 5.5$. The program code therefore have an order $\mathcal{O}(N^{5.5})$ dependency to obtain a required accuracy. Furthermore, as indicated by table 5.6 the correlation gained for argon is far less than what was gained for carbon. Therefore the computational time needed to acquire a required accuracy of the energy is even greater than $\mathcal{O}(N^{5.5})$. This clearly indicates the need to sophisticate the code further. Nevertheless, these results also indicate that the code can be applied to quite large systems.

Adding additional terms to the trial wave-function does not significantly increase the time needed to obtain accurate results for the energy, however parameter optimization becomes increasingly important. Parallelizing the parameter optimization procedure is both a natural and quite easy extension of the existing code. Furthermore, including additional particle-particle correlations requires optimization of more complex trial wave-functions.

[1] Recall that $s.e. = \sigma/\sqrt{M}$, with M the number of sampling points.

Atomic Results

Atom	HF	VMC	'Exact'	% CE
He	-2.8617	-2.8865(1)	-2.9037	59.0(2)
Li	-7.4327	-7.4599(2)	-7.4781	59.9(4)
Be	-14.5730	-14.6072(4)	-14.6673	36.3(4)
B	-24.5290	-24.5679(4)	-24.6539	31.1(3)
C	-37.6886	-37.7307(3)	-37.8451	26.9(2)
Ne	-128.547	-128.595(4)	-128.937	12.1(10)
Ar	-526.817	-526.835(5)		

Table 5.6: VMC results for a Roothaan HF Slater determinant, ref. [16], multiplied with a single parameter electron-electron Padé-Jastrow factor, ref. [15], for selected atoms. The results are compared with the Roothaan HF results, and the exact results, given by ref. [15]. The results are obtained using the variance optimization scheme with weights equal to unity. The last column, % CE, is the difference between the HF energies and the VMC energies relative to the difference between the HF and the exact results.

To reduce the computational effort of such optimizations the accuracy of the VMC procedure needs improvement. In the next chapter we will study the auto-correlation effects, and outline some further advances.

5.3 Auto-Correlation Effects

Auto-correlation effects result in a reduced efficiency of the VMC integration procedure. The theoretical development of these correlations are described in section 3.3.4, and we will here make a more thorough investigation of these effects. As an example we study the lithium ground state.

The ground state of lithium has two electrons in the $1s$ orbital and one electron in the $2s$ orbitals. The two electrons in $1s$ have different spin projections due to the Pauli principle, and we choose the third electron to have a positive spin value. In figure 5.5 the electron-nucleus distances of the two electrons with positive spin projections are plotted. There are several features regarding this plot that need to be commented. First, the inter-electronic distances are strongly correlated. This can be seen from the way the electrons need time to evolve from one position to another, at least this is clearly seen for the outermost electron. Second, the electron close to the nucleus spans a shorter distance than the electron further away. This is due to the different ranges of the $1s$ and $2s$ orbitals. A third effect is that the two electrons changes roles at about 1850 MC cycles.

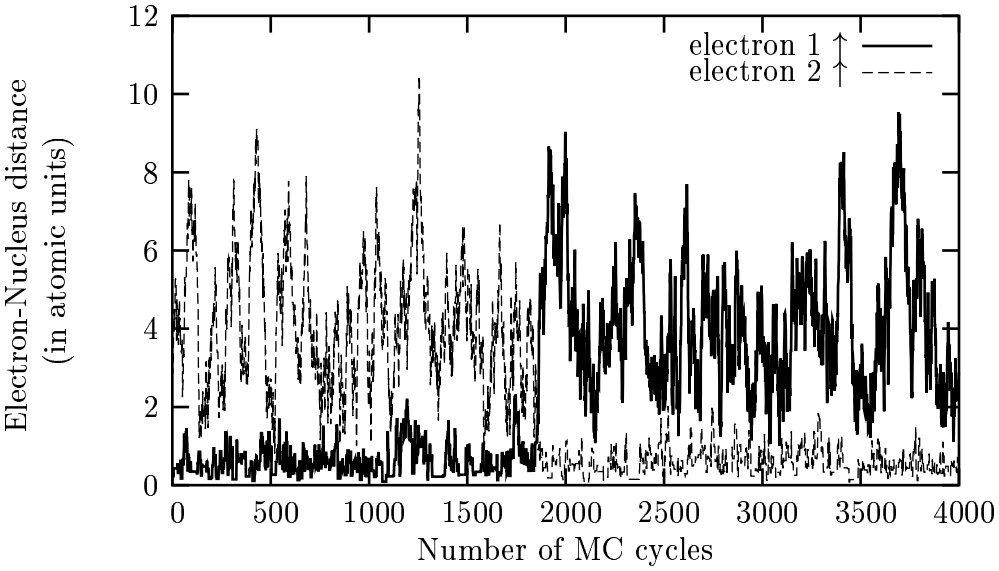

Figure 5.5: Plot of the electron-nucleus distances for the two like-spin electrons of ground state lithium. Both electrons are moved according to the Metropolis algorithm, that samples the probability distribution of the trial wave-function which is a product of a Roothaan HF Slater determinant, ref. [16], multiplied with a single parameter electron-electron Padé-Jastrow factor, ref. [15].

In figure 5.6 a fourth effect becomes apparent. The electron which is farthest away from the nucleus is moved a short distance with almost each individual MC cycle, whereas the two electrons close to the nucleus change positions only a few times. These effects are manifested in the local energy fluctuations. The variations of the electrons close to the nucleus result in large changes in the local energy, while the changes in the outlying electron only result in small changes.

In figure 5.7 the Metropolis acceptance ratios, averaged over approximately 11 000 blocks each consisting of 30 MC cycles, are plotted for the three lithium ground state electrons. What we see from this histogram is that the electron with negative spin projection has an average acceptance ratio of about 0.3, whereas the two electrons with positive spin projections have two distinct acceptance distributions; one normal distribution about an acceptance-ratio 0.3, and one distribution about 0.85. This result clearly demonstrates the inefficiency of the straightforward Metropolis sampling of many-body wave-functions. The problem being that the step lengths are equal for all electrons, whereas the one-electron orbitals span different length-scales.

This problem increases with system size, and is one of the key inefficiencies of the current program code. We have still not found the solution to this problem in the literature, and solving this particular problem remains important for further development. One approach

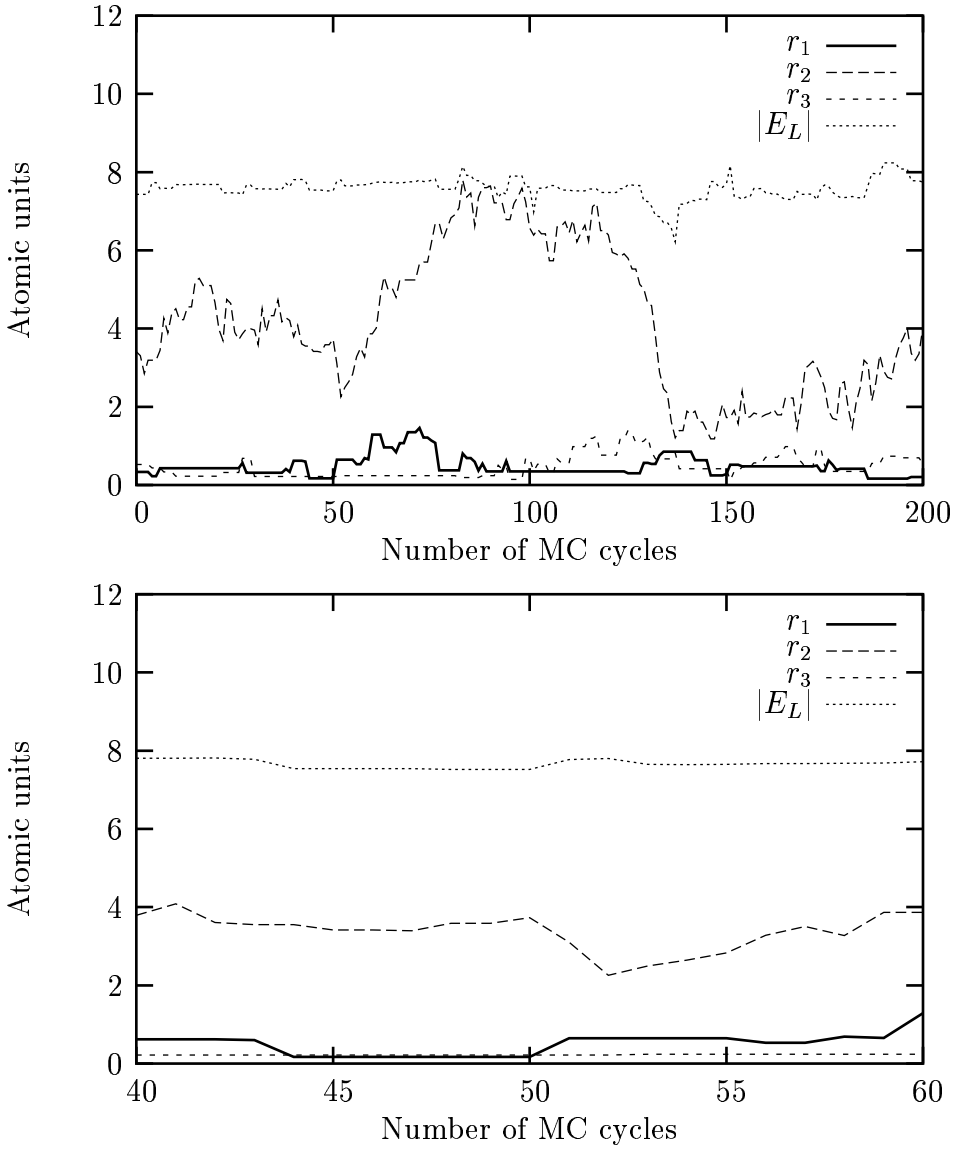

Figure 5.6: Plot of the electron-nucleus distances for the three electrons of ground state lithium, compared with the fluctuations in the local energy. All electrons are moved according to the Metropolis algorithm. The Metropolis algorithm samples the probability distribution of a trial wave-function which is a product of a Roothaan HF Slater determinant, ref. [16], multiplied with a single parameter electron-electron Padé-Jastrow factor, ref. [15]. Two different time scales are shown.

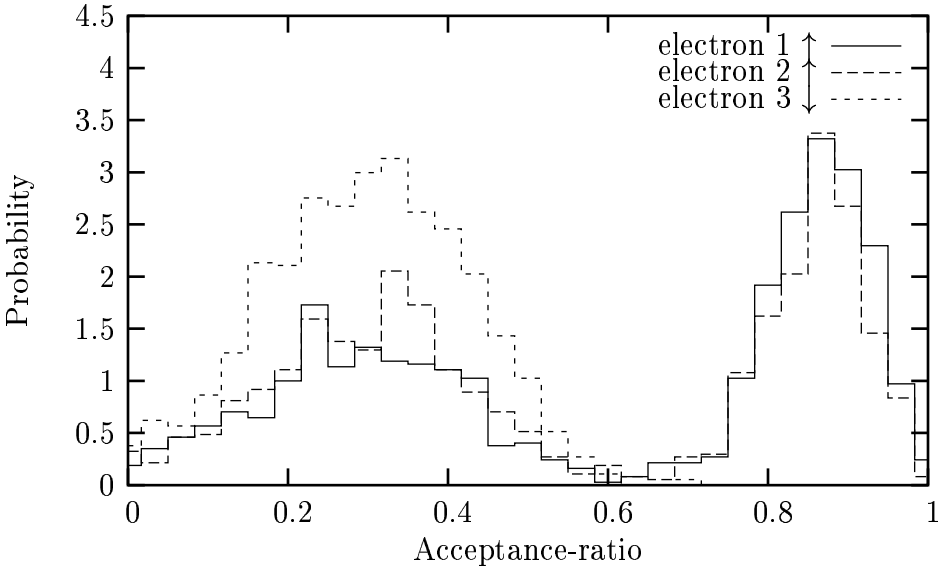

Figure 5.7: Probability distribution as a function of acceptance ratios. The three electrons of Li are all undergoing the Metropolis algorithm. The acceptance ratios are averaged over blocks consisting of 30 MC steps.

that may be investigated is to treat the step lengths of the different electrons individually, requiring that all electrons each have the same acceptance ratios. An argument for allowing such a treatment is that since the wave-function is anti-symmetric with respect to interchange of two electrons, the probability distribution

$$\rho(\mathbf{X}) = \frac{|\Psi(\mathbf{X})|^2}{\int\limits_{\Omega} |\Psi(\mathbf{X})|^2 d\Omega},$$

is symmetric. Therefore, all the permutations of $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$ give the same contribution to the expectation values of physical observables. By this argument only one of the $N!$ subspaces, being perturbations of $\rho(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N)$, needs to be sampled. However, this approach needs a thorough investigation. Nevertheless, the implications of such a scheme is that it would significantly reduce auto-correlation effects.

There are several other ways to reduce the statistical auto-correlation present in the literature. Each step of the Metropolis algorithm, implemented in this thesis, is proposed without any knowledge of the wave-function. The proposed move of the walker isotropic; it is equally likely to move in any direction. This approach leads to a high rejection ratio. To reduce auto-correlation effects two criteria is essential: (i) increase the acceptance and (ii) increase the step length. The first criteria lessens the time it takes for the walker to move from one point in the phase space to the other, and the second criteria allow us to reach other parts of the configuration space more rapidly. Increasing the step length in the above approach decreases the acceptance, and vice versa. To obtain better efficiency the

way we choose our transition probabilities for the proposed moved needs to be changed. Figure 5.8 illustrates the basic principles behind such approaches. In these approaches the transition probabilities of the walkers are modified so that it is more likely to move to a region where $|\Psi|^2$ is large. To satisfy the requirement of detailed balance the acceptance ratios must be modified accordingly.

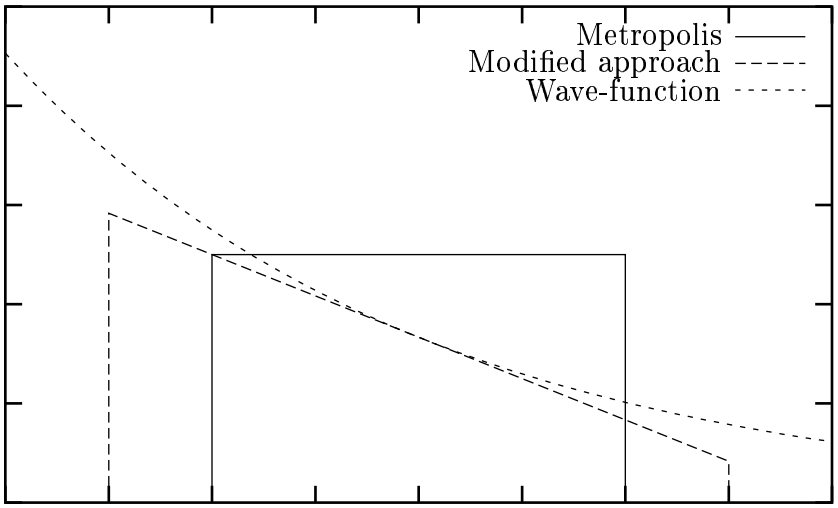

Figure 5.8: Illustration of two different approaches for selecting the transition probabilities for the Metropolis algorithm. In the modified version we are more likely to move toward a region where the value of the wave-function is larger, than to a region where the wave-function is smaller.

One such approach is the Fokker-Planck formalism where we move the walkers according to both diffusion and drift. The drift term is dependent on the gradient of the wave-function, and is implemented similar to the Diffusion Monte Carlo method. By this approach we both obtain higher acceptance and can perform larger step length. For details regarding the Fokker-Planck formalism consult for example ref. [15].

Chapter 6

Conclusion

In this thesis we have worked out the basic principles regarding the use of Monte Carlo integration to solve many-body quantum mechanical systems. A fast and reliable code has been developed, and principal advances for future development have been identified. In this section we summarize the insights gained through this work.

Quantum Monte Carlo methods provide powerful tools for solving the many-body Schrödinger equation. As a part of developing the QMC machinery, Variation Monte Carlo (VMC) is a natural starting point. Here the basics of the VMC method have been developed, and the method has been applied to the atomic problem. The results given in the previous chapter indicate that the main building blocks of the VMC method are very efficient. Furthermore, the principal limitations have been identified, and suggestions for future work have been outlined.

Several key factors regarding the program code need further development. The main factor is related to auto-correlation effects. The Metropolis algorithm needs to be sophisticated so that the walkers span the phase space more efficiently. We further identified the need for different step lengths at different length scales; the electrons close to the nucleus should be moved shorter than the electrons further away. A suggestion was made regarding possible labeling of the individual electrons, as the different electron permutations of the probability distribution simply resulted in symmetrical subspaces.

Another key factor for future development is to allow greater flexibility in the variational form of the trial wave-function; to allow electron-nucleus and electron-electron-nucleus correlations and also to allow linear combinations of Slater determinants.

An additional and natural extension is to implement the Diffusion Monte Carlo (DMC) method. This method produces excellent results for the electronic energies, and its foundations are laid when developing the VMC method. Here an ensemble of walkers is moved, and making many walkers is a simple matter. The diffusion, drift and branching terms should also be simple to implement.

The energy and variance optimization schemes have been studied, but we have omitted to mention effective procedures for optimization of *many* parameters simultaneously. High-dimensional optimization is a common procedure in many areas, and can be incorporated into the variance optimization scheme. The basic building blocks of the variance optimization scheme has been developed and can be used even for many-dimensional parameter optimization. However, such procedures requires high accuracy in the different variations. This emphasizes the importance of reducing auto-correlation effects and obtaining better trial wave-functions. Another easy and natural extension to the current program is to parallelize the code also for variance optimization.

The above extensions would result in a very efficient tool for many-body quantum mechanics, that can be applied to a wide variety of problems. For example for materials and molecules the Slater determinant can be reduced to a block diagonal matrix by identifying non-overlapping parts. Similarly, the terms in the Jastrow factor approach a constant with increasing distance, and such terms may also be removed. This procedure results in a linear dependency with the number of atoms involved. For large molecules an interesting application could be to apply the VMC method to Density Functional Theory (DFT) determinants.

The VMC method is one of the several quantum mechanical many-body methods. In particularly, the VMC method provides an easy and efficient tool for including particle-particle correlations based on physical principles. As one of the main building blocks of the QMC machinery it deserves further investigation!

Appendix A

Appendix

A.1 Program Code

A.1.1 main.cpp

```
#include <mpi.h>

using namespace std;

#include "Ref/Ref.h"
#include "Domain/Domain.h"
#include "Coor/Coor.h"
#include "Vmc/Vmc.h"
#include "SlaterDet/SlaterDet.h"
#include "FuncUpDown/FuncUpDown.h"

int main(int argn, char* args[]) {

  // ************************ MPI ***********************
  int rank, size;
  MPI::Init(argn, args);
  rank = MPI::COMM_WORLD.Get_rank();
  size = MPI::COMM_WORLD.Get_size();
  system("renice +19 -u simensr");

  // Defines the form of the SlaterDet template
  typedef SlaterDet<FuncUp<CoorSpinDiff>,
    FuncDown<CoorSpinDiff> > SlaterDeterminant;

  // ************************ Domain ********************
  Ref<Domain> domain;
  char*       initFileName = "init";
```

```
  domain        = new Domain(initFileName, rank, size);
  domain().init();


  // ************************ vmc ***********************
  Ref<Vmc<SlaterDeterminant> > vmc;
  vmc = new Vmc<SlaterDeterminant >(domain());
  vmc().run();

  // ************************ MPI ***********************
  MPI::Finalize();
  return 0;
}
```

A.1.2 Domain.h

```
#ifndef Domain_IS_INCLUDED
#define Domain_IS_INCLUDED

#include "../Paramizer/Paramizer.h"
#include "../Coor/Coor.h"
#include "../Random/Random.h"
#include "../Distance/Distance.h"
#include "../SpinFactors/SpinFactors.h"
#include "../Ref/Ref.h"
#include "../Random/Random.h"
#include "../fFunction/fFunction.h"

#include <fstream>
#include <string>
#include <iostream>
#include <cstdio>
using namespace std;

#ifndef copyArray_IS_INCLUDED
#define copyArray_IS_INCLUDED
// ************************** copyArray ***********************
inline void copyArray(double* fromArray, double* toArray, int dim) {
  double* _fromArray = fromArray - 1;
  double* _toArray = toArray - 1;
  for (int i=0; i<dim; i++)
    (*++_toArray) = (*++_fromArray);
}
#endif

#ifndef dotProduct_IS_INCLUDED
#define dotProduct_IS_INCLUDED
// ************************** dotProduct **********************
inline double dotProduct(double* array1, double* array2, int dim) {
  double* a1 = array1 - 1;
  double* a2 = array2 - 1;
  double product = 0;
  for (int i=0; i<dim; i++)
    product += (*++a1)*(*++a2);
  return product;
}
#endif

#ifndef odd_IS_INCLUDED
```

```
#define odd_IS_INCLUDED
// ***************************** odd ***************************
inline int odd(int number) {
  return (number - number/2);
}
// ********************* centerOfOddInteger ******************
inline int centerOfOddInteger(int oddInteger) {
  if (!odd(oddInteger)) cerr << "Error finding center of integer "
                             << "in Domain; integer is even (="
                             << oddInteger << ")!" << endl;
  else return ((oddInteger+1)/2);
}
// ****************** centerOfOddIntegerMinusOne **************
inline int centerOfOddIntegerMinusOne(int oddInteger) {
  return (centerOfOddInteger(oddInteger)-1);
}
#endif




// ************************************************************
// *                                                          *
// *                          DOMAIN                          *
// *                                                          *
// ************************************************************
class Domain {
 // ***********************************************************
 // *                PROTECTED ALGORITHMS                     *
 // ***********************************************************
 protected:
  // ************************* MPI ****************************
  int rank; // Prosess number
  int size; // Number of prosesses


  // ****************** Input and Output Files ****************
  char*    standardInput;
  char*    electronConfiguration;
  char*    randomConfig;
  char*    uniDirectionalConfig;
  char*    slaterParam;
  char*    fixedParamsUp;
  char*    fixedParamsDown;
```

```
char*     correlParam;
char*     output;
ofstream  outputFile;


// ********************* Standard Input **********************
int     numDimensions;
int     numThermalization;
int     numCycles;
double  stepLen;                // The step length of the Metropolis
   algorithm.
int     allowSpinFlip;          // Boolean turns on(1) spin-flipping
   routines.
int     quarterCusp;            // Boolean turns on(1) cusp=1/4 off (0)
   .
string  thermalizationType;     // Normal or adaptive.
double  soughtAcceptance;       // What acceptance ratio we wish to
   have.
int     numberSeekAcceptance;   // Number of updates of the step length
                                // to get sought acceptance.
int     varySeekWithRank;       // MPI option to vary the acceptance
                                // ratio with the rank (process number)
                                   .
double  varySeekWithRankStep;   // How much we vary the acceptance.
string  vmcType;                // One at a time or Some.
int     numberVmcRuns;          // Number of VMC runs.
int     varianceOptimization;   // Boolean 1=True, 0=False.
double  referenceEnergy;        // Reference energy for the variance-
                                // optimization. Updated between
                                   different VMC
                                // moves.
double  deltaE;                 // Quantity to allow both energy and
   variance
                                // optimization.
int     setWeightToUnity;       // Boolean 1=True, 0=False

// Accosiated parameters
int     twoD;


// ****************** Electron Configuration *****************
int numParticles;
int numParticlesSpinUp;
int numParticlesSpinDown;
// Booleans indicating which electrons are allowed
```

```
int up1s, up2s, up2px, up2py, up2pz;
int up3s, up3px, up3py, up3pz, up4s;
int down1s, down2s, down2px, down2py, down2pz;
int down3s, down3px, down3py, down3pz, down4s;


// ********************* Random Config ***********************
string randomGenerator; // Either Ran0 or Ran1
// int seed; The seed is given by the input file

// The different rendom generator seeds
int             initRanFlip;  // = seed - rank*3mill
int             initRanMove;  // = seed - rank*3mill - 1 mill
int             initRanMetro; // = seed - rank*3mill - 2 mill
// References to the random generator
Ref<Random2>    randomFlip;
Ref<Random2>    randomMove;
Ref<Random2>    randomMetro;


// *************** Uni-Directional Configuration *************
int    uniDirectionalMovement;        // Do we seek for minima in
   parameter
                                      // space (1=True, 0=False)?
int    continueAfterUniDirectional;   // Whem minima is found, do we
                                      // wish to make additional
                                      // sophisticated searches
                                      // (1=True, 0=False)?
int    numberOfUniDirectionalMoves;   // Total number of
   sophisticated
                                      // minima searches.
int    reduceLocalAreaBetweenMoves;   // Reduce the area in parameter
   -
                                      // space (1=True, 0=False)?
double reduceLocalAreaByFraction;     // Reduce the area in parameter
   -
                                      // space by fraction.
int    increaseNumCyclesBetweenMoves;// Increase the number of
                                      // cycles (1=True, 0=False)?
int    increaseNumCyclesByFactor;     // Increase by factor.
int    increaseNumThermalizationBetweenMoves; // Increase
   thermalization
                                             // (1=True, 0=False)?
int    increaseNumThermalizationByFactor;     // Increase by factor.
```

```
// Indicator specifying whether or not minima is found
int*   uniDirectionalIndicator;


// ************************* Other ****************************
int    numVariations;          // Total number of local variations
                               // numSlaterVariations*
                                  numCorrelVariations
double h;                      // Numberical difference (for
   calculating
                               // derivatives)
int*   paramIndex;             // Index used for initialization and
   updating
                               // of alphaParam and betaParam in the
                               // neighborhood of some central values
                                   for
                               // the parameters.


// *************** Slater-Determinant Parameters **************
string  orbitalType; // For example hydrogen, HF, etc.
int     numAlpha;    // Number of variational (Slater) parameters (
   min 1).
double* centralAlphaParams; // The central parameters (with no local
                            // variation this is the parameter we
                               use).
int*    numAlphaVar; // Number of local variations for each iniviual
                     // variational parameter.
double* alphaStep;   // The step (parameter difference) between
                     // each local variation.
double* alphaParam;  // Array containing all the different local
                     // configurations in (Slater) parameter space.

int     numSlaterVariations;  // Total number of local Slater
   variations.
int     centerSlater;         // Index indicating which of the
   Slater
                               // variations is the central one.
void    createSlaterParams(); // Routine to initialize the Slater
                               // determinant parameters.


// ****************** Correlation Parameters ******************
string  fBetaType;          // For example fNone, fBeta, fBetaLinear
   , etc.
```

```
int      numBeta;            // Number of variational (correlation)
                             // parameters (min 1).
double* centralBetaParams; // The central parameters (with no local
                             // variation this is the parameter we use
                                ).
int*     numBetaVar;         // Number of local variations for each
   iniviual
                             // variational parameter.
double* betaStep;            // The step (parameter difference)
   between
                             // each local variation.

double*     betaParam;              // Array containing all the
   different local
                                    // configurations in (correlation)
                                       parameter
                                    // space.
int         numCorrelVariations;    // Total number of local
   correlation
                                    // variations.
int         centerCorrel;           // Index indicating which of the
   correlation
                                    // variations is the central one.
fFunction* f;                       // Function set in accordance to
   the input
                                    // of fBetaType.
void        createCorrelParams(); // Routine to initialize the
   correlation
                                    // parameters.
void attachFParams(int _expBool);// Attach paramaters to given
   fFunction
// Routines to create different fFunctions
void createFBeta(int _expBool);
void createFBeta2(int _expBool);
void createFBeta3(int _expBool);

void createFBeta2r2(int _expBool);
void createFExtended(int _expBool);
void createFNone();


// ******************** Coordinates and Spin ********************
int           currentParticle;// The current particle (being moved)
int           otherParticle;  // The proposed particle to
   interchange spin
```

```
                                // with (if spin flip is allowed)
double*         coorArray;      // Cartesian coordinates of all coors.
double*         _coorArray;
double*         trialCoorArray; // Cartesian coordinates of trialCoor.
int             spinFlip;       // Boolean, is spin being proposed
   flipped.

int*            spinArray;      // Array containing the different
   spins.
CoorSpinDiff*   coors;          // The coordinate objects.
CoorSpinDiff*   trialCoor;      // The trial coordinate.
void            computeTrialPosition(); // Proposes a move of the
   current
                                        // particle.
double          coorStep();     // Random walk of one coordinate.
void            proposeFlip();  // Proposes an interchange of electron
    spin
                                // between current particle and (
                                   randomly)
                                // generated other particle.
void            acceptTrialPosition(); // Accept the proposed move.
void            rejectTrialPosition(); // Rejects the proposed move.
void            resetPtr();            // Sets coorArray = _coorArray.
void            setPositionToOrigin(); // Sets all coordinates equal
   zero.


// ****************** Inter-electronic distances *****************
Distance*       distance;     // Object that keeps track of the
   distances.
DistanceDiff*   distanceDiff; // Keeps track of the differentiated
   distances.
void            createDistanceAndDistanceDiff(); // Creates the above
   objects.


// ************************ Spin-Factors ************************
// Object to impose the different cusp conditions for like-spin and
   opposite-
// spin electrons, 1/4 and 1/2 respectively. Is turned on and off by
    the
// value of quarterCusp (on(1) cusp=1/4 off (0)).
SpinFactors* spinFactors;
void         createSpinFactors();
```

```
// ************************************************************
// *                      PUBLIC ALGORITHMS                   *
// ************************************************************
public:
 // Constructor: Input file name, prosess number, and number of MPI
 // processes
 Domain(char* _initFileName, int _rank, int _size);


 // ***************************** MPI ****************************
 int             getRank()                    {return rank;}
 int             getSize()                    {return size;}


 // ******************* Input and Output Files *******************
 ofstream*       getOutputFile()              {return &outputFile;}
 char*           getFixedParamsUp()           {return fixedParamsUp;}
 char*           getFixedParamsDown()         {return fixedParamsDown;}


 // ********************** Standard Input ************************
 int             getNumDimensions()           {return numDimensions;}
 int             getNumThermalization()       {return numThermalization;}
 int             getNumCycles()               {return numCycles;}
 double          getStepLen()                 {return stepLen;}

 int             getAllowSpinFlip()           {return allowSpinFlip;}
 string          getThermalizationType()      {return thermalizationType
    ;}
 double          getSoughtAcceptance()        {return soughtAcceptance;}
 int             getNumberSeekAcceptance()    {return
    numberSeekAcceptance;}

 int             getVarySeekWithRank()        {return varySeekWithRank;}
 double          getVarySeekWithRankStep()    {return
    varySeekWithRankStep;}
 string          getVmcType()                 {return vmcType;}
 int             getVarianceOptimization()    {return
    varianceOptimization;}

 double          getReferenceEnergy()         {return referenceEnergy;}
 double          getDeltaE()                  {return deltaE;}
```

```
int           getSetWeightToUnity()       {return setWeightToUnity;}
void          changeStepLen(int numSteps, int acceptedSteps,
                            double soughtAcceptance);

void          setReferenceEnergy(double E) {referenceEnergy=E;}


// ****************** Electron Configuration ******************
int getNumParticles()   {return numParticles;}
int getUp1s()           {return up1s;}
int getUp2s()           {return up2s;}
int getUp2px()          {return up2px;}

int getUp2py()          {return up2py;}
int getUp2pz()          {return up2pz;}
int getDown1s()         {return down1s;}
int getDown2s()         {return down2s;}

int getDown2px()        {return down2px;}
int getDown2py()        {return down2py;}
int getDown2pz()        {return down2pz;}


// ********************** Random Config ***********************
Ref<Random2> getRandomMetro() {return randomMetro;}
Ref<Random2> getRandomMove()  {return randomMove;}


// *************** Uni-Directional Configuration **************
void increaseNumCycles();
void increaseNumThermalization();
void reduceLocalArea();
int  getUniDirectionalMovement(){return uniDirectionalMovement;}

int  getNumberVmcRuns()          {return numberVmcRuns;}
int  getNumberOfUniDirectionalMoves() {return
   numberOfUniDirectionalMoves;}
void initUniDirectionalIndicator();
int  isMovementUniDirectional(double* alphaParams, double*
   betaParams);


// ************************** Other ****************************
int           getNumVariations()        {return numVariations;}
double        getH()                    {return h;}
```

```
// *************** Slater-Determinant Parameters ***************
string  getOrbitalType()          {return orbitalType;}
int     getNumAlpha()             {return numAlpha;}
int     getNumSlaterVariations()  {return numSlaterVariations;}
double* getAlphaParam(int i)      {return &(alphaParam[i*numAlpha]);}

int     getCentralSlater()        {return centerSlater;}
void    setCentralAlphaParam(double* param);
void    calculateAlphaParamArray();


// ******************* Correlation Parameters ******************
int        getNumBeta()             {return numBeta;}
double*    getBetaParam(int i)      {return &(betaParam[i*numBeta])
   ;}
void       setCentralBetaParam(double* param);
int        getCentralCorrel()       {return centerCorrel;}

fFunction* getF()                   {return f;}
void       calculateBetaParamArray();
int        getNumCorrelVariations() {return numCorrelVariations;}


// ******************** Coordinates and Spin *********************
double*       getCoorArray()              {return coorArray;}
CoorSpinDiff* getCoors();
CoorSpinDiff* getTrialCoor();
int*          getSpinArray()              {return spinArray;}

int*          getSpinFlip()               {return &spinFlip;}
int           getOtherParticle()          {return otherParticle;}
void          setCoorsToOrigon();
void          setCurrentParticle(int __currentParticle);

void          setOtherParticle(int __otherParticle);
void          setToNextParticle();
void          init();
void          initVMC();

void          suggestMove();
void          acceptMove();
void          rejectMove();
void          acceptThermalizedMove();
```

```
  void          rejectThermalizedMove();


  // ****************** Inter-electronic distances ****************
  Distance*     getDistance()              {return distance;}
  DistanceDiff* getDistanceDiff()          {return distanceDiff;}


  // ************************ Spin-Factors ************************
  SpinFactors*  getSpinFactors()           {return spinFactors;}


  // ********************** Potential Energy **********************
  double getNucleusElectronPotential();
  double getInterElectronicPot() {return distance->getPotential();}


  // ************************** Summary ***************************
  void    initSummary();
  void    Summary();
};

#endif
```

A.1.3 Domain.cpp

```
#include "Domain.h"

  // Constructor: Input file name, prosess number, and number of MPI
  // processes
Domain::Domain(char* initFileName, int _rank, int _size) {
  ifstream initIfile(initFileName); // Opening initIFile
  rank = _rank;
  size = _size;

  // ****************** Input and Output Files ******************
  standardInput         = new char[100];
  electronConfiguration = new char[100];
  randomConfig          = new char[100];
  uniDirectionalConfig  = new char[100];
  slaterParam           = new char[100];
  fixedParamsUp         = new char[100];
  fixedParamsDown       = new char[100];
  correlParam           = new char[100];
  output                = new char[100];
```

```
// Reading from initIFile
if (!(initIfile >> standardInput))
  cerr << "Error reading file: " << initFileName << endl;
if (!(initIfile >> electronConfiguration))
  cerr << "Error reading file: " << initFileName << endl;
if (!(initIfile >> randomConfig))
  cerr << "Error reading file: " << initFileName << endl;
if (!(initIfile >> uniDirectionalConfig))
  cerr << "Error reading file: " << initFileName << endl;
if (!(initIfile >> slaterParam))
  cerr << "Error reading file: " << initFileName << endl;
if (!(initIfile >> fixedParamsUp))
  cerr << "Error reading file: " << fixedParamsUp << endl;
if (!(initIfile >> fixedParamsDown))
  cerr << "Error reading file: " << fixedParamsDown << endl;
if (!(initIfile >> correlParam))
  cerr << "Error reading file: " << initFileName << endl;
if (!(initIfile >> output))
  cerr << "Error reading file: " << initFileName << endl;
// Opening the files
ifstream standardInputIfile(standardInput);
ifstream electronConfigurationIfile(electronConfiguration);
ifstream randomConfigIfile(randomConfig);
ifstream uniDirectionalConfigIfile(uniDirectionalConfig);
sprintf(output, "%s.run%d", output, rank);
outputFile.open(output);


// ********************** Standard Input ************************
if (!(standardInputIfile >> numDimensions))
  cerr << "Error reading file: " << standardInput
       << ". Couldn't read: numDimensions" << endl;
if (!(standardInputIfile >> numThermalization))
  cerr << "Error reading file: " << standardInput
       << ". Couldn't read: numThermalization" << endl;
if (!(standardInputIfile >> numCycles))
  cerr << "Error reading file: " << standardInput
       << ". Couldn't read: numCycles" << endl;
if (!(standardInputIfile >> stepLen))
  cerr << "Error reading file: " << standardInput
       << ". Couldn't read: stepLen" << endl;
if (!(standardInputIfile >> allowSpinFlip))
  cerr << "Error reading file: " << standardInput
       << ". Couldn't read: allowSpinFlip" << endl;
if (!(standardInputIfile >> quarterCusp))
```

```
    cerr << "Error reading file: " << standardInput
         << ". Couldn't read: quarterCusp" << endl;
  thermalizationType = new char[100];
  if (!(standardInputIfile >> thermalizationType))
    cerr << "Error reading file: " << standardInput
         << ". Couldn't read: thermalizationType" << endl;
  if (!(standardInputIfile >> soughtAcceptance))
    cerr << "Error reading file: " << standardInput
         << ". Couldn't read: soughtAcceptance" << endl;
  if (!(standardInputIfile >> numberSeekAcceptance))
    cerr << "Error reading file: " << standardInput
         << ". Couldn't read: numberSeekAcceptance" << endl;
  if (!(standardInputIfile >> varySeekWithRank))
    cerr << "Error reading file: " << standardInput
         << ". Couldn't read: varySeekWithRank" << endl;
  if (!(standardInputIfile >> varySeekWithRankStep))
    cerr << "Error reading file: " << standardInput
         << ". Couldn't read: varySeekWithRankStep" << endl;
  vmcType = new char[100];
  if (!(standardInputIfile >> vmcType))
    cerr << "Error reading file: " << standardInput
         << ". Couldn't read: vmcType" << endl;
  if (!(standardInputIfile >> numberVmcRuns))
    cerr << "Error reading file: " << standardInput
         << ". Couldn't read: numberVmcRuns" << endl;
  if (!(standardInputIfile >> varianceOptimization))
    cerr << "Error reading file: " << standardInput
         << ". Couldn't read: varianceOptimization" << endl;
  if (!(standardInputIfile >> referenceEnergy))
    cerr << "Error reading file: " << standardInput
         << ". Couldn't read: referenceEnergy" << endl;
  if (!(standardInputIfile >> deltaE))
    cerr << "Error reading file: " << standardInput
         << ". Couldn't read: deltaE" << endl;
  if (!(standardInputIfile >> setWeightToUnity))
    cerr << "Error reading file: " << standardInput
         << ". Couldn't read: setWeightToUnity" << endl;
  twoD                  = numDimensions*2;


  // ****************** Electron Configuration ********************
  if (!(electronConfigurationIfile >> numParticles))
    cerr << "Error reading file: " << electronConfiguration
         << ". Couldn't read: numParticles" << endl;
  if (!(electronConfigurationIfile >> numParticlesSpinUp ))
```

```
    cerr << "Error reading file: " << electronConfiguration
         << ". Couldn't read: numParticlesSpinUp" << endl;
  if (!(electronConfigurationIfile >> up1s >> up2s
        >> up2px >> up2py >> up2pz >> up3s >> up3px
        >> up3py >> up3pz >> up4s))
    cerr << "Error reading file: " << electronConfiguration
         << ". Couldn't read: up1s >> up2s >> up2px >> up2py >> up2pz"
             << endl;
  if (!(electronConfigurationIfile >> numParticlesSpinDown ))
    cerr << "Error reading file: " << electronConfiguration
         << ". Couldn't read: numParticlesSpinDown" << endl;
  if (!(electronConfigurationIfile >> down1s >> down2s
        >> down2px >> down2py >> down2pz >> down3s
        >> down3px >> down3py >> down3pz >> down4s))
    cerr << "Error reading file: " << electronConfiguration
         << ". Couldn't read: down1s >> down2s >> down2px >> down2py
             >> down2pz" << endl;


  // ********************** Random Config ***********************
  randomGenerator = new char[100];
  if (!(randomConfigIfile >> randomGenerator))
    cerr << "Error reading file: " << randomConfig
         << ". Couldn't read: randomGenerator" <<  endl;
  int seed;
  if (!(randomConfigIfile >> seed))
    cerr << "Error reading file: " << randomConfig
         << ". Couldn't read: seed" <<  endl;
  if ((!seed < 0) & (seed > -1000000))
    cerr << "Error! The seed should be in the range (-1 000 000, -1)"
        << endl;
  else {
    initRanMove  = seed - rank*3000000;
    initRanFlip  = seed - rank*3000000 - 1000000;
    initRanMetro = seed - rank*3000000 - 2000000;
  }
  if ( randomGenerator=="Ran0") {
    randomMove           = new Ran0(initRanMove);
    randomFlip           = new Ran0(initRanFlip);
    randomMetro          = new Ran0(initRanMetro);
  }
  else if ( randomGenerator=="Ran1") {
    randomMove           = new Ran1(initRanMove);
    randomFlip           = new Ran1(initRanFlip);
    randomMetro          = new Ran1(initRanMetro);
```

```
}
else cerr << "Error input randomGenerator, only Ran0 and Ran1
   implemented"
          << endl;


// *************** Uni-Directional Configuration **************
if (!(uniDirectionalConfigIfile >> uniDirectionalMovement))
  cerr << "Error reading file: " << uniDirectionalConfig
       << ". Couldn't read: uniDirectionalMovement(1=True,0=False)"
           << endl;
if (!(uniDirectionalConfigIfile >> numberOfUniDirectionalMoves))
  cerr << "Error reading file: " << uniDirectionalConfig
       << ". Couldn't read: numberOfUniDirectionalMoves" << endl;
if (!(uniDirectionalConfigIfile >> reduceLocalAreaByFraction))
  cerr << "Error reading file: " << uniDirectionalConfig
       << ". Couldn't read: reduceLocalAreaByFraction" << endl;
if (!(uniDirectionalConfigIfile >> increaseNumCyclesByFactor))
  cerr << "Error reading file: " << uniDirectionalConfig
       << ". Couldn't read: increaseNumCyclesByFactor" << endl;
if (!(uniDirectionalConfigIfile >> increaseNumThermalizationByFactor
   ))
  cerr << "Error reading file: " << uniDirectionalConfig
       << ". Couldn't read: increaseNumThermalizationByFactor" <<
          endl;

// ************************** Other ****************************
paramIndex = new int[10];


// ************** Slater-Determinant Parameters **************
createSlaterParams();


// ****************** Correlation Parameters ******************
createCorrelParams();


// ************************** Other ****************************
numVariations = numSlaterVariations * numCorrelVariations;
h = 0.0001;    // Numerical differentiate parameter


// ******************** Coordinates and Spin ********************
coorArray       = new double[numDimensions*numParticles];
```

```
  trialCoorArray = new double[numDimensions];
  spinArray      = new int[numParticles + 1];
  coors          = new CoorSpinDiff[numParticles];

  trialCoor      = new CoorSpinDiff[1];
  setCoorsToOrigon();
  for (int i = 0; i <= numParticles; i++) spinArray[i]=1;
  for (int i = 0; i < numParticles; i++)
    coors[i].attach((coorArray + i*numDimensions), numDimensions,
                    &spinArray[i], h);
  trialCoor[0].attach(trialCoorArray, numDimensions,
                      &spinArray[numParticles], h);


  // *************** Uni-Directional Configuration *************
  uniDirectionalIndicator = new int[2*(numAlpha+numBeta)];
  initUniDirectionalIndicator();


  // ********************** Spin-Factors ************************
  createSpinFactors();
  createDistanceAndDistanceDiff();
}



// ************************************************************
// *                        Standard Input                    *
// ************************************************************
//
// ********************** changeStepLen ***********************
void Domain::changeStepLen(int numSteps, int acceptedSteps,
                           double soughtAcceptance) {
  double acceptance = ((double) acceptedSteps)/numSteps;
  if (acceptance == 1) cerr << "Error in changing stepLen, either
     stepLen=0 or numThermalization small" << endl;
  else stepLen *= (1-soughtAcceptance)/(1-acceptance);
}



// ************************************************************
// *                  Uni-Directional Configuration           *
// ************************************************************
//
```

```
// ********************* increaseNumCycles ********************
void Domain::increaseNumCycles() {
  numCycles*=increaseNumCyclesByFactor;

}

// *****************  increaseNumThermalization ***************
void Domain::increaseNumThermalization() {
  numThermalization*=increaseNumThermalizationByFactor;

}

// ************************ reduceLocalArea *********************
void Domain::reduceLocalArea() {
  for (int i=0; i<numAlpha; i++)
    alphaStep[i]*=reduceLocalAreaByFraction;
  for (int i=0; i<numBeta; i++)
    betaStep[i]*=reduceLocalAreaByFraction;
}

/// ***************** initUniDirectionalIndicator **************
void Domain::initUniDirectionalIndicator() {
  int num = numAlpha + numBeta;
  for (int i=0; i<2*num; i++)
    uniDirectionalIndicator[i] = 1;
}

// ******************* isMovementUniDirectional ***************
int Domain::isMovementUniDirectional(double* alphaParams, double*
   betaParams) {
  // Update the uniDirectionalIndicator
  int num = numAlpha + numBeta;
  double small = 1e-15;
  for (int i=0; i<numAlpha; i++) {
    if ( alphaParams[i] <= centralAlphaParams[i] + small )
      uniDirectionalIndicator[i] = 0;
    if ( alphaParams[i] >= centralAlphaParams[i] - small)
      uniDirectionalIndicator[i+num] = 0;
  }
  for (int i=0; i<numBeta; i++) {
    if ( betaParams[i] <= centralBetaParams[i] + small)
      uniDirectionalIndicator[i+numAlpha] = 0;
    if ( betaParams[i] >= centralBetaParams[i] - small)
      uniDirectionalIndicator[i+numAlpha+num] = 0;
  }
```

```
  // Here the test is performed
  int uni = 0;
  for (int i=0; i<2*num; i++)
    if ( uniDirectionalIndicator[i] != 0 ) uni = 1;

  return uni;
}



// ************************************************************
// *                Slater-Determinant Parameters             *
// ************************************************************
//
// ******************** createSlaterParams *********************
void Domain::createSlaterParams() {
  // Opening slaterParamIfile
  ifstream slaterParamIfile(slaterParam);
  //    cerr << "Error! Could not open file: " << slaterParam << endl;
  orbitalType = new char[100];
  if (!(slaterParamIfile >> orbitalType))
    cerr << "Error reading file: " << slaterParam
         << ". Couldn't read: orbitalType" << endl;
  if (!(slaterParamIfile >> numAlpha))
    cerr << "Error reading file: " << slaterParam
         << ". Couldn't read: numAlpha" << endl;
  if (numAlpha<1) cerr << "numAlpha must be at least 1" << endl;
  centralAlphaParams  = new double[numAlpha];
  numAlphaVar         = new int[numAlpha];
  alphaStep           = new double[numAlpha];
  for (int i=0; i<numAlpha; i++ ) {
    if (!(slaterParamIfile >> centralAlphaParams[i] >>
          numAlphaVar[i] >> alphaStep[i])) {
      cerr << "Error reading file: " << slaterParam
           << ". Couldn't read: centralAlphaParams[i] >>"
           << " numAlphaVar[i] >> alphaStep[i] (i=" << i << ")" <<
              endl;
      cerr << "Possible missmach between numAlpha and"
           << " number of additional lines?" << endl;
    }
  }
  numSlaterVariations = 1;
  for (int i=0; i<numAlpha; i++)
    numSlaterVariations *= numAlphaVar[i];
```

```
  if ( (numSlaterVariations<1) ||
       ( !(odd(numSlaterVariations)) ) )
    cerr << "alpha#numVar must be odd and >0" << cerr << endl;
  alphaParam = new double[numAlpha*numSlaterVariations];
  calculateAlphaParamArray();
  centerSlater = centerOfOddIntegerMinusOne(numSlaterVariations);
}

// ****************** calculateAlphaParamArray ****************
void Domain::calculateAlphaParamArray() {
  for (int i=0; i<numAlpha; i++)
    centralAlphaParams[i]
      -= centerOfOddIntegerMinusOne(numAlphaVar[i]) * alphaStep[i];
  for (int i=0;i<numAlpha;i++) paramIndex[i]=0;
  bool increaseParam;
  int index;
  int j = 0;
  for (int i=0; i<numSlaterVariations; i++) {
    for (int k=0; k<numAlpha; k++) {
      alphaParam[j++] = centralAlphaParams[k];
    }
    index = 0;
    increaseParam = false;
    while (!increaseParam) {
      paramIndex[index] += 1;
      if (paramIndex[index] == numAlphaVar[index]) {
        centralAlphaParams[index] -= (numAlphaVar[index]-1)*alphaStep[
            index];
        paramIndex[index] = 0;
        index +=1;
      }
      else {
        centralAlphaParams[index] += alphaStep[index];
        increaseParam = true;
      }
    }
  }
  for (int i=0; i<numAlpha; i++)
    centralAlphaParams[i] +=
      centerOfOddIntegerMinusOne(numAlphaVar[i])*alphaStep[i];
}

// ******************* setCentralAlphaParam ******************
void Domain::setCentralAlphaParam(double* param) {
  copyArray(param, centralAlphaParams, numAlpha);
```

```
}



// ************************************************************
// *                    Correlation Parameters                *
// ************************************************************
//
// ******************** createCorrelParams *******************
void Domain::createCorrelParams() {
  // Opening correlParamIfile
  ifstream correlParamIfile(correlParam);
  //    cerr << "Error! Could not open file: " << correlParam << endl;
  fBetaType = new char[100];
  if (!(correlParamIfile >> fBetaType))
    cerr << "Error reading file: " << correlParam
         << ". Couldn't read: fBetaType" << endl;
  if (!(correlParamIfile >> numBeta))
    cerr << "Error reading file: " << correlParam
         << ". Couldn't read: numBeta" << endl;
  centralBetaParams = new double[numBeta];
  numBetaVar     = new int[numBeta];
  betaStep       = new double[numBeta];
  for (int i=0; i<numBeta; i++) {
    if (!(correlParamIfile >> centralBetaParams[i] >>
          numBetaVar[i] >> betaStep[i])) {
      cerr << "Error reading file: " << correlParam
           << ". Couldn't read: centralBetaParams[i] >>"
           << " numBetaVar[i] >> betaStep[i] (i=" << i << ")" << endl;
      cerr << "Possible missmach between numBeta and"
           << " number of additional lines?" << endl;
    }
  }

  numCorrelVariations = 1;
  for (int i=0; i<numBeta; i++)
    numCorrelVariations *= numBetaVar[i];
  if ( (numCorrelVariations<1) ||
       ( !(odd(numCorrelVariations)) ) )
    cerr << "beta#numVar must be odd and >0" << cerr << endl;

  betaParam = new double[numBeta*numCorrelVariations];
  calculateBetaParamArray();
  centerCorrel = centerOfOddIntegerMinusOne(numCorrelVariations);
```

```
  if      ( fBetaType == "fNone" )         createFNone();
  else if ( fBetaType == "fBeta" )         createFBeta(1);
  else if ( fBetaType == "fBetaLinear" )   createFBeta(0);
  else if ( fBetaType == "fBeta2" )        createFBeta2(1);
  else if ( fBetaType == "fBeta3" )        createFBeta3(1);
  else if ( fBetaType == "fBeta2r2" )      createFBeta2r2(1);
  else if ( fBetaType == "fExtended" )     createFExtended(1);
  else if ( fBetaType == "fBeta2Linear" ) createFBeta2(0);
  else    cerr << "fBetaType not identified!" << endl;
}

// ****************** calculateBetaParamArray ****************
void Domain::calculateBetaParamArray() {
  for (int i=0; i<numBeta; i++)
    centralBetaParams[i]
      -= centerOfOddIntegerMinusOne(numBetaVar[i]) * betaStep[i];
  for (int i=0;i<numBeta;i++) paramIndex[i]=0;
  bool increaseParam;
  int index;
  int j = 0;
  for (int i=0; i<numCorrelVariations; i++) {
    for (int k=0; k<numBeta; k++) {
      betaParam[j++] = centralBetaParams[k];
    }
    index = 0;
    increaseParam = false;
    while (!increaseParam) {
      paramIndex[index] += 1;
      if (paramIndex[index] == numBetaVar[index]) {
        centralBetaParams[index] -= (numBetaVar[index]-1)*betaStep[
            index];
        paramIndex[index] = 0;
        index +=1;
      }
      else {
        centralBetaParams[index] += betaStep[index];
        increaseParam = true;
      }
    }
  }
  for (int i=0; i<numBeta; i++)
    centralBetaParams[i] +=
      centerOfOddIntegerMinusOne(numBetaVar[i]) * betaStep[i];
}
```

```
// ******************** setCentralBetaParam ********************
void Domain::setCentralBetaParam(double* param) {
  copyArray(param, centralBetaParams, numBeta);
}

// ************************ attachFParams ************************
void Domain::attachFParams(int _expBool) {
  for (int i=0; i<numCorrelVariations; i++)
    f[i].attach(numBeta, &(betaParam[i*numBeta]),_expBool);
}

// ************************* createFNone *************************
void Domain::createFNone() {
  if (numBeta != 1) cerr << "Need numBeta=1 to create fNone!" << endl;
  numCorrelVariations = 1;
  f = new fNone[1];
  attachFParams(0);
}

// ************************* createFBeta *************************
void Domain::createFBeta(int _expBool) {
  if (numBeta != 1)
    cerr << "Could not create fBeta, numBeta must be 1!" << endl;
  else {
    f = new fBeta[numCorrelVariations];
    attachFParams(_expBool);
  }
}

// ************************ createFBeta2 *************************
void Domain::createFBeta2(int _expBool) {
  if (numBeta != 2)
    cerr << "Could not create fBeta2, numBeta must be 2!" << endl;
  else {
    f = new fBeta2[numCorrelVariations];
    attachFParams(_expBool);
  }
}

// ************************ createFBeta3 *************************
void Domain::createFBeta3(int _expBool) {
  if (numBeta != 3)
    cerr << "Could not create fBeta3, numBeta must be 3!" << endl;
  else {
    f = new fBeta3[numCorrelVariations];
```

```
    attachFParams(_expBool);
  }
}

// ********************** createFBeta2r2 **********************
void Domain::createFBeta2r2(int _expBool) {
  if (numBeta != 2)
    cerr << "Could not create fBeta2r2, numBeta must be 2!" << endl;
  else {
    f = new fBeta2r2[numCorrelVariations];
    attachFParams(_expBool);
  }
}

// ********************** createFExtended **********************
void Domain::createFExtended(int _expBool) {
  if (numBeta != 8)
    cerr << "Could not create fExtended, numBeta must be 8!" << endl;
  else {
    f = new fExtended[numCorrelVariations];
    attachFParams(_expBool);
  }
}



// ************************************************************
// *                  Inter-electronic distances              *
// ************************************************************
//
// ***************** createDistanceAndDistanceDiff **************
void Domain::createDistanceAndDistanceDiff() {
  distance         = new Distance;
  distance->attach(coors, trialCoor,
                   numParticles);
  distanceDiff     = new DistanceDiff[numDimensions];
  for (int i=0; i<numDimensions; i++)
      distanceDiff[i]
        .attach(coors, trialCoor, numParticles, h,i);

}



// ************************************************************
```

```
// *                    Coordinates and Spin                    *
// ************************************************************
//
// ******************** setCurrentParticle *******************
void Domain::setCurrentParticle(int __currentParticle) {
  currentParticle = __currentParticle;
  _coorArray      = (coorArray + currentParticle*numDimensions);
  spinFactors->setCurrentParticle(currentParticle);
  distance->setCurrentParticle(currentParticle);
  for (int i=0; i< numDimensions; i++)
    distanceDiff[i].setCurrentParticle(currentParticle);
}

// ********************** setOtherParticle *********************
void Domain::setOtherParticle(int __otherParticle) {
  otherParticle  = __otherParticle;
  spinFactors->setOtherParticle(otherParticle);
}

// ********************** setToNextParticle ********************
void Domain::setToNextParticle() {
  if (++currentParticle == numParticles) {
    currentParticle = 0;
    resetPtr();
  }
  else
    _coorArray += numDimensions;
  spinFactors->setToNextParticle();
  distance->setToNextParticle();
  for (int i=0; i< numDimensions; i++)
    distanceDiff[i].setToNextParticle();
}

// ***************************** init ***************************
void Domain::init() {
  setCurrentParticle(0);
  for (int i = 0; i <= numParticles; i++)
    spinArray[i]=1;

  for (int i = 0; i < numParticles; i++) {
    setPositionToOrigin();
    computeTrialPosition();
    acceptTrialPosition();
    if (currentParticle >= numParticlesSpinUp)
      coors[currentParticle].flipSpin();
```

```
    coors[currentParticle].calculateR();
    setToNextParticle();
  }
  spinFactors->init();
  spinFactors->setOtherParticle(0);
  distance->initialize();
}

// ************************* initVMC *************************
void Domain::initVMC() {
  setCurrentParticle(0);
  distance->initialize();
  for (int i = 0; i < numDimensions; i++)
    distanceDiff[i].initialize();
}

// ************************ suggestMove ************************
void Domain::suggestMove() {
  computeTrialPosition();
  distance->suggestMove();
  proposeFlip();
}

// ************************ acceptMove *************************
void Domain::acceptMove() {
  for (int i = 0; i < numDimensions; i++)
    distanceDiff[i].suggestMove();
  acceptThermalizedMove();
  for (int i = 0; i < numDimensions; i++)
    distanceDiff[i].acceptMove();
}

// ************************ rejectMove *************************
void Domain::rejectMove() {
  rejectThermalizedMove();
 for (int i = 0; i < numDimensions; i++)
    distanceDiff[i].rejectMove();
}

// ******************* acceptThermalizedMove *******************
void Domain::acceptThermalizedMove() {
  acceptTrialPosition();
  distance->acceptMove();
}
```

```
// ****************** rejectThermalizedMove ******************
void Domain::rejectThermalizedMove() {
  rejectTrialPosition();
  distance->rejectMove();
}

// ********************** proposeFlip **********************
void Domain::proposeFlip() {
  if (allowSpinFlip) {
    setOtherParticle ( (int) (randomFlip().getNum()*numParticles) );
    spinFlip = (*(spinArray+currentParticle) != *(spinArray+
        otherParticle));
  }
  if (quarterCusp) {
    if (spinFlip) spinFactors->calculateFlipFactors();
    else spinFactors->calculateNoFlipFactors();
  }
}

// ************************ resetPtr ************************
void Domain::resetPtr() {
  _coorArray = coorArray;
}

// ****************** computeTrialPosition ******************
void Domain::computeTrialPosition() {
  for (int i = 0; i < numDimensions; i++) {
    trialCoorArray[i] = _coorArray[i] + coorStep();
  }
  trialCoor[0].spin() = coors[currentParticle].spin();
  trialCoor[0].calculateR();
  trialCoor[0].calculateDiffs();
}

// ****************** acceptTrialPosition ******************
void Domain::acceptTrialPosition() {
  for (int i = 0; i < numDimensions; i++)
    _coorArray[i] = trialCoorArray[i];
  coors[currentParticle].copy(trialCoor);
}

// ****************** rejectTrialPosition ******************
void Domain::rejectTrialPosition() {
}
```

```
// ******************** setPositionToOrigin ********************
void Domain::setPositionToOrigin() {
   for (int i = 0; i < numDimensions; i++)
    _coorArray[i] = 0;
   coors[currentParticle].calculateR();
}

// ************************* getCoors *************************
CoorSpinDiff* Domain::getCoors() {
  return coors;
}

// ************************ getTrialCoors *********************
CoorSpinDiff* Domain::getTrialCoor() {
  return trialCoor;
}

// ************************** coorStep ************************
double Domain::coorStep() {
  return (stepLen * (randomMove().getNum() - 0.5)/ numParticles);
}


// ********************** setCoorsToOrigon ********************
void Domain::setCoorsToOrigon() {
  for (int i=0; i<numDimensions*numParticles; i++)
    coorArray[i]=0;

  for (int i=0; i<numDimensions; i++)
    trialCoorArray[i]=0;
}


// ************************************************************
// *                        Spin-Factors                      *
// ************************************************************
//
// ******************** createSpinFactors *********************
void Domain::createSpinFactors() {
  spinFactors = new SpinFactors;
  spinFlip    = 0;
  spinFactors->allocate(numParticles, spinArray);
  spinFactors->setQuarterCusp(quarterCusp);
  spinFactors->init();
```

```
}


// ************************************************************
// *                      Potential Energy                    *
// ************************************************************
//
// ***************** getNucleusElectronPotential **************
double Domain::getNucleusElectronPotential() {
  double nucleusElectronPotential = 0;
  for (int i=0; i<numParticles; i++)
    nucleusElectronPotential -= 1/coors[i].r();
  return ( (double) numParticles )*nucleusElectronPotential;
}


// ************************************************************
// *                          Summary                         *
// ************************************************************
//
// ************************ initSummary ***********************
void Domain::initSummary() {
  outputFile << "!!!!!!!!!!!!   INITIAL CONFIGURATION
       !!!!!!!!!!!!!!!!!!"
             << endl
             << "numDimensions          " << numDimensions << endl
             << "numParticles           " << numParticles << endl
             << "numParticlesSpinUp     " << numParticlesSpinUp <<
                endl
             << "spinUpConfig           [ "
             << up1s << " " << up2s << " " << up2px << " " << up2py
             << " " << up2pz << " ]" << endl
             << "numParticlesSpinDown   " << numParticlesSpinDown <<
                endl
             << "spinDownConfig         [ "
             << down1s << " " << down2s << " " << down2px << " " <<
                down2py
             << " " << down2pz << " ]" << endl
             << "randomGenerator        " << randomGenerator << endl
             << "initRanMove/Flip/Metro [ " << initRanMove << " "
             << initRanFlip << " " << initRanMetro << " ]\n"
             << "orbitalType            " << orbitalType << endl
             << "fBetaType              " << fBetaType << endl;
  if (quarterCusp)
    outputFile << "Quarter-cusp for like-spin particles turned on!\n";
```

```
  else
    outputFile << "Quarter-cusp for like-spin particles turned off!\n"
       ;
  if (allowSpinFlip)
    outputFile << "Spin-flip turned on!\n";
  else
    outputFile << "Spin-flip turned off!\n";
  outputFile << "Thermalization set to type '" << thermalizationType
             << "', and VMC type set\n" << "to '" << vmcType << "'.\n"
                ;
  if (uniDirectionalMovement)
    outputFile << "The minima in parameter space "
               << "will be sought (uniDirectionalMovement=1).\n"
               << "There will be " << numberOfUniDirectionalMoves
               << " unidirectional moves. "
               << "One unidirectional move will consist\n"
               << "of maximum " << numberVmcRuns << " VMC runs.\n"
               << "In-between each unidirectional move"
               << " the number of cycles will be increased by \n"
               << "a factor " << increaseNumCyclesByFactor
               << ", and the number"
               << " of thermalization by a factor "
               << increaseNumThermalizationByFactor << ". The "
               << "variational \nparameters step length will be
                  reduced"
               << " by fraction " << reduceLocalAreaByFraction << ".\n
                  ";
  else outputFile << numberVmcRuns << " VMC runs (with differing"
                  << " seeds) will be conducted.\n";
  outputFile << "
     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
             << endl;
}

// ************************** Summary ****************************
void Domain::Summary() {
  outputFile << "
     ********************************************************"
             << endl
             << "Summary of domain:" << endl
             << "     numThermalization  " << numThermalization <<
                endl
             << "     numCycles          " << numCycles << endl
             << "     stepLen            " << stepLen << endl
             << "     centralAlphaParams [ ";
```

```
  for (int i=0; i<numAlpha; i++)
    outputFile << centralAlphaParams[i]  << " ";

  outputFile << "]" << endl
             << "     numAlphaVar        [ ";

  for (int i=0; i<numAlpha; i++) outputFile << numAlphaVar[i]  << " ";
  outputFile << "]" << endl
             << "     alphaStep          [ ";
  for (int i=0; i<numAlpha; i++) outputFile << alphaStep[i]  << " ";
  outputFile << "]" << endl
             << "     centralBetaParams  [ ";
  for (int i=0; i<numBeta; i++) outputFile << centralBetaParams[i]
      << " ";
  outputFile << "]" << endl
             << "     numBetaVar         [ ";
  for (int i=0; i<numBeta; i++) outputFile << numBetaVar[i]  << " ";
  outputFile << "]" << endl
             << "     betaStep           [ ";
  for (int i=0; i<numBeta; i++) outputFile << betaStep[i]  << " ";
  outputFile << "]" << endl
             << "*************************************************************
                "
             << endl;
}
```

A.1.4 Coor.h

```
#ifndef Coor_IS_INCLUDED
#define Coor_IS_INCLUDED

#include <iostream>
#include <cmath>
using namespace std;


// ***************************************************************
// *                              COOR                           *
// ***************************************************************
class Coor {

 protected:
  double* x;     // Cartesian coordinates
  double* _x;    // Pointer to current coordinate
  int     len;   // Number of cartesian dimensions


 public:
  Coor(double* x, int _len);
  Coor() {}
  virtual void           attach(double* __x, int _len);
  inline virtual void    resetPtr()          {_x = x;}
  inline virtual double& operator()()        {return *_x;}
  inline virtual double& operator()(int num) {return x[num];}
  inline virtual void    operator++(int)     {_x++;}
  virtual int            getLen()            {return len;}
};


// ***************************************************************
// *                             COORR                           *
// ***************************************************************
class CoorR : public Coor {

 protected:
  double _r;
  int    rIsCalculated;

 public:
  CoorR(double* x, int _len);
  CoorR();
```

```
  virtual void    attach(double* x, int _len);
  inline virtual double& r()      {return _r;}
  inline virtual void    calculateR();
};


// ****************************************************************
// *                          COORSPIN                            *
// ****************************************************************
class CoorSpin : public CoorR {

 protected:
  int* _spin;

 public:
  CoorSpin(double* x, int _len, int* __spin);
  CoorSpin() {}
  virtual void attach(double* x, int _len, int* __spin);
  inline virtual int& spin()     {return *_spin;}
  inline virtual int  flipSpin() {return (*_spin *= -1);}

};


// ****************************************************************
// *                        COORSPINDIFF                          *
// ****************************************************************
class CoorSpinDiff : public CoorSpin {

 protected:
  // rDiff[0]             = sqrt( (x+h)^2 + y^2     + ... )
  // rDiff[1]             = sqrt( x^2     + (y+h)^2 + ... )
  // ...
  // rDiff[numDimensions] = sqrt( (x-h)^2 + y^2     + ... )
  // ...
  double* rDiff;
  double h, twoh;
  int twoLen;

 public:
  CoorSpinDiff(double* x, int _len, int* __spin, double _h);
  CoorSpinDiff() {}
  virtual void attach(double* x, int _len, int* __spin, double _h);
  virtual void calculateDiffs();
  // rdiff(0)             = sqrt( (x+h)^2 + y^2     + ... )
```

```
  // rdiff(1)              = sqrt( x^2      + (y+h)^2 + ... )
  // ...
  // rdiff(numDimensions) = sqrt( (x-h)^2 + y^2      + ... )
  // ...
  inline virtual double& r()            {return _r;}
  inline virtual double& r(int number) {return rDiff[number];}
  inline virtual double& operator()()   {return *_x;}
  inline virtual double& operator()(int num);
  inline virtual double  operator()(int num, int currentNr);
  inline virtual double* getX()         {return x;}
  inline virtual void    copy(CoorSpinDiff* copyCoor);
  inline virtual void    calculateR();
  virtual double rdiff(int currentNr)  {return rDiff[currentNr];}
};

#endif
```

A.1.5 Coor.cpp

```
#include "Coor.h"

// ***************************************************************
// *                              COOR                           *
// ***************************************************************
Coor::Coor(double* __x, int _len) : x(__x), len(_len) {
  resetPtr();
}

void Coor::attach(double* __x, int _len) {
  x   = __x;
  len = _len;
  resetPtr();
}


// ***************************************************************
// *                              COORR                          *
// ***************************************************************
CoorR::CoorR(double* __x, int _len) : Coor(__x, _len) {
  rIsCalculated = 0;
}

CoorR::CoorR() : Coor() {
  rIsCalculated = 0;
}
```

```
void CoorR::attach(double* __x, int _len) {
  Coor::attach(__x, _len);
}

void CoorR::calculateR() {
  _r = x[0]*x[0];
  for (int i = 1; i < len; i++)
    _r += (x[i]*x[i]);
  _r = sqrt(_r);
}


// ****************************************************************
// *                          COORSPIN                            *
// ****************************************************************
CoorSpin::CoorSpin(double* __x, int _len, int* __spin) : CoorR(__x,
    _len) {
  _spin = __spin;
}

void CoorSpin::attach(double* __x, int _len, int* __spin) {
  CoorR::attach(__x, _len);
  _spin = __spin;
}


// ****************************************************************
// *                        COORSPINDIFF                          *
// ****************************************************************
CoorSpinDiff::CoorSpinDiff(double* __x, int _len, int* __spin, double
    _h) : CoorSpin(__x, _len, __spin) {
  h = _h;
  twoh = 2*h;
  rDiff = new double[len*2];
}

void CoorSpinDiff::attach(double* __x, int _len, int* __spin, double
    _h) {
  CoorSpin::attach(__x, _len, __spin);
  h = _h;
  twoh = 2*h;
  twoLen = 2*len;
  rDiff = new double[twoLen + 1];
}
```

```
void CoorSpinDiff::calculateR() {
  _r = x[0]*x[0];
  for (int i = 1; i < len; i++)
    _r += (x[i]*x[i]);
  _r = sqrt(_r);
  rDiff[twoLen] = _r;
}

void CoorSpinDiff::calculateDiffs() {
  double rTemp;
  for (int i=0; i<len; i++) {

    x[i] += h;
    rTemp = x[0]*x[0];
    for (int j = 1; j < len; j++)
      rTemp += (x[j]*x[j]);
    rDiff[i] = sqrt(rTemp);

    x[i] -= twoh;
    rTemp = x[0]*x[0];
    for (int j = 1; j < len; j++)
      rTemp += (x[j]*x[j]);
    rDiff[i+len] = sqrt(rTemp);

    x[i] += h;
  }
}

double& CoorSpinDiff::operator()(int num) {
  return x[num];
}

double CoorSpinDiff::operator()(int num, int currentNumber) {
  return x[num] + ((currentNumber == num) - (currentNumber == num+len)
      )*h;
}

void CoorSpinDiff::copy(CoorSpinDiff* copyCoor) {
  _r = copyCoor->r();
  for (int i=0; i <= twoLen; i++)
    rDiff[i] = copyCoor->rdiff(i);
}
```

A.1.6 Distance.h

```
#ifndef Distance_IS_INCLUDED
#define Distance_IS_INCLUDED

#include "../Coor/Coor.h"
#include <blitz/array.h>
#include <cmath>
using namespace blitz;

#ifndef sqr_IS_INCLUDED
#define sqr_IS_INCLUDED
inline double sqr(double x) {return x*x;};
#endif

// ************************************************************
// *                          DISTANCE                        *
// ************************************************************
class Distance {

 protected:
  CoorSpinDiff* Coordinate;
  CoorSpinDiff* TrialCoordinate;
  int numParticles;      // Number of particles
  int Nm1;               // numParticles - 1
  int numDimensions;     // Number of dimension
  // Array of Upper matrix composed of the distance between
  // particle i and j
  Array<double, 2> interElectronicDistances;
  // The (new) distance from the current particle to the other
     particles
  Array<double, 1> trialColumn;
  Array<double, 1> trialRow;
  int currentParticle;  // This is the particle that currently is
                        // (proposed) moved

 public:
  Distance() {}
  void    attach(CoorSpinDiff* _Coordinate, CoorSpinDiff*
     _TrialCoordinate,
                 int _numParticles);
  void    initialize();
  void    setToNextParticle();
  void    setCurrentParticle(int _currentParticle);
  void    suggestMove();
```

```
  void     acceptMove();
  void     rejectMove() {}
  double   getPotential();
  Array<double, 2> getInterElectronicDistances()
    {return interElectronicDistances;}
  Array<double, 1> getTrialColumn() {return trialColumn;}
  Array<double, 1> getTrialRow()    {return trialRow;}
};

// ***************************************************************
// *                          DISTANCEDIFF                       *
// ***************************************************************
class DistanceDiff : public Distance {
 protected:
  double h;           // Differential parameter,
                      // dr/dx ~= ( r(x+h) - r(x-h) )/2h
  double twoh;        // 2*h
  int differentiate; // Which dimension (x=0, y=1 or z=2 in three dim
     .)
                      // to be differentiated


 public:
  DistanceDiff() {}
  void attach(CoorSpinDiff* _Coordinate, CoorSpinDiff*
     _TrialCoordinate,
              int _numParticles, double _h, int _differentiate);
  void initialize();
  void suggestMove();
  void acceptMove();
  void rejectMove() {}
};

#endif
```

A.1.7 Distance.cpp

```
#include "Distance.h"

// ***************************************************************
// *                            DISTANCE                         *
// ***************************************************************
// ************************ Distance ***********************
// Class to monitor inter-electronic distances:
//
// Here the interElectronicDistances matrix is given by:
```

```
//
//                    |  0   r01  r02  ...  r0(N-1)   |
//                    | r01   0   r12  ...  r1(N-1)   |
//                    | r02  r12   0   ...  r2(N-1)   |
//                    |  .          .       .         |
//                    |  .             .    .         |
//                    |  .                .  .         |
//                    |  .                r(N-2)(N-1) |
//                    | r0(N-1) .   .   .    0        |
//
void Distance::attach(CoorSpinDiff* _Coordinate,
                      CoorSpinDiff* _TrialCoordinate,
                      int _numParticles) {
  Coordinate      = _Coordinate;
  TrialCoordinate = _TrialCoordinate;
  numParticles    = _numParticles;
  Nm1             = numParticles - 1;
  numDimensions   = (*Coordinate).getLen();

  interElectronicDistances.resize(numParticles, numParticles);
  trialColumn.resize(numParticles);
  trialRow.resize(numParticles);
}

void Distance::initialize() {
  CoorSpinDiff* temp;
  temp = TrialCoordinate;
  setCurrentParticle(0);
  for (int i=0; i<numParticles; i++) {
    TrialCoordinate = &Coordinate[i];
    suggestMove();
    acceptMove();
    setToNextParticle();
  }
  TrialCoordinate = temp;
}


void Distance::setToNextParticle() {
 currentParticle++;
 if (currentParticle == numParticles) currentParticle=0;
}


void Distance::setCurrentParticle(int _currentParticle) {
```

```
  currentParticle = _currentParticle;
}


void Distance::suggestMove()
{
  for (int i=0; i<numParticles; i++) {
    double difference = sqr(Coordinate[i]()-(*TrialCoordinate)());
    for (int j=1; j<numDimensions; j++) {
      Coordinate[i]++; (*TrialCoordinate)++;
      difference += sqr(Coordinate[i]()-(*TrialCoordinate)());
    }
    Coordinate[i].resetPtr(); (*TrialCoordinate).resetPtr();
    trialColumn(i) = trialRow(i) = sqrt(difference);
  }
  trialColumn(currentParticle) = trialRow(currentParticle) = 0;

}

void Distance::acceptMove()
{
  Range N(0,Nm1);
  interElectronicDistances( currentParticle, N )  = trialColumn( N );
  interElectronicDistances( N , currentParticle ) = trialRow( N );
}

double Distance::getPotential()
{
  double potential = 0;
  for (int i=0; i<Nm1; i++)
    for (int j=i+1; j<numParticles; j++)
      potential += 1/interElectronicDistances(i, j);
  return potential;
}



// ***************************************************************
// *                          DISTANCEDIFF                       *
// ***************************************************************
//
//
// Class to monitor the values used for numerical derivation of the
// Jastrow-factor:
//                   dfij/dxi     ~= ( fij(xi+h) - fij(xi-h))/2h
```

```
// and:
//                    d^2fij/dxi^2 ~= ( fij(xi+h) + fij(xi-h) -2fij)/h
   ^2
//
// ie. we need the values rij(xi+h) and rij(xi-h). Here xi is the
   either
// x,y,z (for the three dimensional problem) of particle i.
// The relation:
//                    drij/dxi = - drij/dxj
// Implies:
//                    rij(xi+h) = rij(xj-h)
//
// Utilizing this relation, we need only to calculate one of the two.
// Here the rij matrix, given by:
//
//        | 0          r01(x1+h)  r02(x2+h)  ...  r0(N-1)(x(N-1)+h)    |
//        | r01(x0+h)     0       r12(x2+h)  ...  r1(N-1)(x(N-1)+h)    |
//        | r02(x0+h)  r12(x2+h)      0      ...  r2(N-1)(x(N-1)+h)    |
//        | .             .                              .             |
//        | .             .                              .             |
//        | .             .                              .             |
//        | .             .                   r(N-2)(N-1)(x(N-1)+h) |
//        | r0(N-1)(x0+h)           r(N-2)(N-1)(x(N-2)+h)     0       |
void DistanceDiff::attach(CoorSpinDiff* _Coordinate,
                          CoorSpinDiff* _TrialCoordinate,
                          int _numParticles, double _h,
                          int _differentiate) {
  Coordinate      = _Coordinate;
  TrialCoordinate = _TrialCoordinate;
  numParticles    = _numParticles;
  Nm1             = numParticles - 1;

  numDimensions   = (*Coordinate).getLen();
  h               = _h;
  twoh            = 2*h;
  differentiate   = _differentiate;

  interElectronicDistances.resize(numParticles, numParticles);
  trialColumn.resize(numParticles);
  trialRow.resize(numParticles);
}

void DistanceDiff::initialize() {
  CoorSpinDiff* temp;
  temp = TrialCoordinate;
```

```
  setCurrentParticle(0);
  for (int i=0; i<numParticles; i++) {
    TrialCoordinate = &Coordinate[i];
    suggestMove();
    acceptMove();
    setToNextParticle();
  }
  TrialCoordinate = temp;
}

void DistanceDiff::suggestMove()
{
  (*TrialCoordinate)(differentiate)-=h;
  for (int i=0; i<currentParticle; i++) {
    double difference = sqr(Coordinate[i]()-(*TrialCoordinate)());
    for (int j=1; j<numDimensions; j++) {
      Coordinate[i]++; (*TrialCoordinate)++;
      difference += sqr(Coordinate[i]()-(*TrialCoordinate)());
    }
    Coordinate[i].resetPtr(); (*TrialCoordinate).resetPtr();
    trialRow(i) = sqrt(difference);
  }
  for (int i=currentParticle+1; i<numParticles; i++) {
    double difference = sqr(Coordinate[i]()-(*TrialCoordinate)());
    for (int j=1; j<numDimensions; j++) {
      Coordinate[i]++; (*TrialCoordinate)++;
      difference += sqr(Coordinate[i]()-(*TrialCoordinate)());
    }
    Coordinate[i].resetPtr(); (*TrialCoordinate).resetPtr();
    trialRow(i) = sqrt(difference);
  }

  (*TrialCoordinate)(differentiate)+=twoh;
  for (int i=0; i<currentParticle; i++) {
    double difference = sqr(Coordinate[i]()-(*TrialCoordinate)());
    for (int j=1; j<numDimensions; j++) {
      Coordinate[i]++; (*TrialCoordinate)++;
      difference += sqr(Coordinate[i]()-(*TrialCoordinate)());
    }
    Coordinate[i].resetPtr(); (*TrialCoordinate).resetPtr();
    trialColumn(i) = sqrt(difference);
  }
  for (int i=currentParticle+1; i<numParticles; i++) {
    double difference = sqr(Coordinate[i]()-(*TrialCoordinate)());
    for (int j=1; j<numDimensions; j++) {
```

```
      Coordinate[i]++; (*TrialCoordinate)++;
      difference += sqr(Coordinate[i]()-(*TrialCoordinate)());
    }
    Coordinate[i].resetPtr(); (*TrialCoordinate).resetPtr();
    trialColumn(i) = sqrt(difference);
  }
  (*TrialCoordinate)(differentiate)-=h;
  trialRow(currentParticle) = 0;
  trialColumn(currentParticle) = 0;
}

void DistanceDiff::acceptMove()
{
  Range N(0,Nm1);
  interElectronicDistances( currentParticle, N )  = trialColumn( N );
  interElectronicDistances( N, currentParticle ) = trialRow( N );
}
```

A.1.8 Ref.h

```
#ifndef Ref_IS_INCLUDED
#define Ref_IS_INCLUDED

#ifdef _DEBUG_
#define REF(TYPE) RefFund<TYPE>
#endif

#ifndef _DEBUG_
#define REF(TYPE) TYPE*
#endif

// ****************************************************************
// *                              REF                             *
// ****************************************************************
template <class type>
class Ref {

 protected:
  type* item;

 public:
  Ref() {}
  Ref(type& _item)            {item = &_item;}
  void attach(type& _item)    {item = &_item;}
  void operator=(type& _item) {item = &_item;}
  void operator=(type* _item) {item =  _item;}
  inline type& operator()()   {return *item;}
  inline type* getPtr()       {return item;}
};

// ****************************************************************
// *                            REFFUND                           *
// ****************************************************************
template <class type>
class RefFund : public Ref<type> {

 public:
  void operator=(type& _item) {item = &_item;}
  void operator=(type* _item) {item =  _item;}
  inline type& operator()()   {return *item;}
  inline operator type& ()    {return *item;}

};
```

```
#endif
```

A.1.9 Random.h

```
#ifndef Random2_IS_INCLUDED
#define Random2_IS_INCLUDED

#include <iostream>
#include <cstdio>

// ****************************************************************
// *                              RANDOM2                         *
// ****************************************************************
class Random2 {
 protected:
  long dummy;

 public:
  Random2(long seed) : dummy(seed) {}
  virtual ~Random2() {}
  virtual double getNum(void) {return 0;}
  virtual void   demo(int num, FILE* stream);
};

// ****************************************************************
// *                               RAN0                           *
// ****************************************************************
class Ran0 : public Random2 {
 public:
  Ran0(long seed) : Random2(seed) {}
  double getNum(void);
};

// ****************************************************************
// *                               RAN1                           *
// ****************************************************************
class Ran1 : public Random2 {
 public:
  Ran1(long seed) : Random2(seed) {}
  double getNum(void);
};

#endif
```

A.1.10 Random.cpp

```
#include "Random.h"
```

```
// ***************************************************************
// *                          RANDOM2                            *
// ***************************************************************
void Random2::demo(int num, FILE* stream) {
  for (int i = 0; i < num; i++)
    fprintf(stream, "%f\n", getNum());
}

// ***************************************************************
// *                            RAN0                             *
// ***************************************************************
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define MASK 123459876
double Ran0::getNum(void) {
    long        k;
    double ans;

    dummy ^= MASK;
    k       = (dummy)/IQ;
    dummy   = IA*(dummy - k*IQ) - IR*k;
    if(dummy < 0) dummy += IM;
    ans     = AM*(dummy);
    dummy ^= MASK;
    return ans;
}
#undef IA
#undef IM
#undef AM
#undef IQ
#undef IR
#undef MASK

// ***************************************************************
// *                            RAN1                             *
// ***************************************************************
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
```

```
#define NDIV (1+(IM-1)/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)

double Ran1::getNum(void)
{
    int             j;
    long            k;
    static long     iy=0;
    static long     iv[NTAB];
    double          temp;

    if (dummy <= 0 || !iy) {
        if (-(dummy) < 1) dummy=1;
        else dummy = -(dummy);
        for(j = NTAB + 7; j >= 0; j--) {
            k     = (dummy)/IQ;
            dummy = IA*(dummy - k*IQ) - IR*k;
            if(dummy < 0) dummy += IM;
            if(j < NTAB) iv[j] = dummy;
        }
        iy = iv[0];
    }
    k     = (dummy)/IQ;
    dummy = IA*(dummy - k*IQ) - IR*k;
    if(dummy < 0) dummy += IM;
    j     = iy/NDIV;
    iy    = iv[j];
    iv[j] = dummy;
    if((temp=AM*iy) > RNMX) return RNMX;
    else return temp;
}
#undef IA
#undef IM
#undef AM
#undef IQ
#undef IR
#undef NTAB
#undef NDIV
#undef EPS
#undef RNMX
```

A.1.11 SlaterDet.h

```
#ifndef SlaterDet_IS_INCLUDED
#define SlaterDet_IS_INCLUDED

#include <iostream>
#include "../SlaterMatrix/SlaterMatrix.h"
#include "../Coor/Coor.h"
#include "../Domain/Domain.h"
#include "../Func/Func.h"
#include "../SingleParticleFuncs/SingleParticleFuncs.h"
#include "../Random/Random.h"
#include "../Ref/Ref.h"


// ****************************************************************
// *                          SLATERDET                           *
// ****************************************************************
template <class FuncUp, class FuncDown>
class SlaterDet {

 protected:

  SlaterMatrix_Det *smUp, *smDown;

  int           numParticles;
  int           numDimensions;
  int           dimUp, dimDown;
  double        *newValuesColumnUp, *newValuesColumnDown;
  int           *columnIndexOfParticle;

  Domain        *domain;
  CoorSpinDiff *coors;
  Ref<CoorSpinDiff> trialCoor;
  FuncUp        *funcsUp;
  FuncDown      *funcsDown;
  Ref<Random2> random;

  Ref<double>  ratioUp, ratioDown;
  double        ratio;
  double        dDiffRatio;
  double        *diffRatios, *_diffRatios;
  double        det;
  Ref<double>  detUp, detDown;
  double        trialDet;
```

```
  Ref<double>  trialDetUp, trialDetDown;

  int          *spinFlip;
  int          currentParticle;
  int          otherParticle;
  int          spin;
  int          currentColumn;
  int          otherColumn;
  int          moveAcceptance, flipAcceptance, moveFlipAcceptance;

#ifdef _DEBUG_
  int          moved, flipped, valueSidesCalculated;
#endif

 public:

  SlaterDet();
  ~SlaterDet();

  void       init(Domain* _domain, int alphaVar);
  void       initNewVmcRun();
  void       setToNextParticle();
  void       setCurrentParticle(int _currentParticle);

  void       suggestMove();
  void       suggestFlip();
  void       suggestMoveFlip();

  void       acceptMove();
  void       acceptFlip();
  void       acceptMoveFlip();

  void       rejectMove();
  void       rejectFlip();
  void       rejectMoveFlip();

  void       calcDiffRatios();
  void       calcDDiffRatio();

  void       setSpinFlip(int _sf)    {spinFlip=_sf;}
  int&       getSpinFlip()           {return spinFlip;}
  int&       getCurrentParticle()    {return currentParticle;}
  int&       getOtherParticle()      {return otherParticle;}
  double&    getDet()                {return det;}
  double&    getTrialDet()           {return trialDet;}
```

```
  double&   getRatio()                {return ratio;}
  double*   getDiffRatiosPtr()        {return diffRatios;}
  double&   getDDiffRatio()           {return dDiffRatio;}

  FuncUp*   getFuncsUpPtr()           {return funcsUp;}
  FuncDown* getFuncsDownPtr()         {return funcsDown;}
  int       getMoveAcceptance()       {return moveAcceptance;}
  int       getFlipAcceptance()       {return flipAcceptance;}
  int       getMoveFlipAcceptance()   {return moveFlipAcceptance;}
  void      resetAcceptances();

  void      summary();
  void      initDiff();

 protected:

#ifdef _DEBUG_
  int       checkIfValueSidesCalculated();
#endif
};

#include "SlaterDet.cpp"

#endif
```

A.1.12 SlaterDet.cpp

```
#ifndef SlaterDetCPP_IS_INCLUDED
#define SlaterDetCPP_IS_INCLUDED

#include "SlaterDet.h"

// ************************************************************
// *                          SLATERDET                       *
// ************************************************************
//
// ********************* SlaterDet ************************
template <class FuncUp, class FuncDown>
SlaterDet<FuncUp, FuncDown>::SlaterDet() {}


// ********************* ~SlaterDet ************************
template <class FuncUp, class FuncDown>
SlaterDet<FuncUp, FuncDown>::~SlaterDet()
{
  delete[] columnIndexOfParticle;
```

```
  delete[] newValuesColumnUp;
  delete[] newValuesColumnDown;
  delete[] diffRatios;
  delete[] funcsUp;
  delete[] funcsDown;
  delete[] smUp;
  delete[] smDown;
}


// ***************************** init ******************************
template<class FuncUp, class FuncDown>
void SlaterDet<FuncUp, FuncDown>::init(Domain* _domain, int alphaVar)
    {

  domain          = _domain;
  numParticles    = domain->getNumParticles();
  numDimensions   = domain->getNumDimensions();
  coors           = domain->getCoors();
  trialCoor       = domain->getTrialCoor();
  spinFlip        = domain->getSpinFlip();

  //Calculate the number of particles with spin up and down
  //and generate initial particle index to column index transform
     array
  columnIndexOfParticle = new int[numParticles];
  dimUp   = 0;
  dimDown = 0;
  for (int i = 0; i < numParticles; i++)
    if (coors[i].spin() > 0)
      columnIndexOfParticle[i] = dimUp++;
    else  // if (coors[i].spin() == -1)
      columnIndexOfParticle[i] = dimDown++;

  //Allocate newValueColumn arrays
  newValuesColumnUp   = new double[dimUp];
  newValuesColumnDown = new double[dimDown];

  //Allocate array for first derivative ratios
  diffRatios = new double[numParticles*numDimensions + 1];

  //Generate function objects
  funcsUp   = new FuncUp[dimUp];
  funcsDown = new FuncDown[dimDown];
  for (int i = 0; i < dimUp; i++) {
```

```
    funcsUp[i].init(trialCoor(), dimUp, domain, alphaVar);
    funcsUp[i].attachResult(newValuesColumnUp);
    funcsUp[i].attachDdiffResult(newValuesColumnUp);
  }
  for (int i = 0; i < dimDown; i++) {
    funcsDown[i].init(trialCoor(), dimDown, domain, alphaVar);
    funcsDown[i].attachResult(newValuesColumnDown);
    funcsDown[i].attachDdiffResult(newValuesColumnDown);
  }

  //Generate SlaterMatrix objects
  smUp   = new SlaterMatrix_Det[1];
  smDown = new SlaterMatrix_Det[1];
  smUp->redim(dimUp);
  smUp->setPtrToNewColumn(newValuesColumnUp);
  smDown->redim(dimDown);
  smDown->setPtrToNewColumn(newValuesColumnDown);
  ratioUp       = smUp->getRatio();
  ratioDown     = smDown->getRatio();
  detUp         = smUp->getDet();
  detDown       = smDown->getDet();
  trialDetUp    = smUp->getTrialDet();
  trialDetDown = smDown->getTrialDet();

  //Initialize SlaterMatrix objects
  for (int i = 0; i < dimUp; i++) {
    funcsUp[i].calcValueCenter(coors[i]);
    funcsUp[i].valuePt();
    smUp->calcRatio();
    smUp->importNewColumn();
    smUp->setToNextColumn();
  }
  for (int i = 0; i < dimDown; i++) {
    funcsDown[i].calcValueCenter(coors[i+dimUp]);
    funcsDown[i].valuePt();
    smDown->calcRatio();
    smDown->importNewColumn();
    smDown->setToNextColumn();
  }

#ifdef _DEBUG_
  moved = flipped = valueSidesCalculated = 0;
#endif

  setCurrentParticle(0);
```

```
  resetAcceptances();
  initDiff();
}


// ********************** initNewVmcRun *************************
template <class FuncUp, class FuncDown>
void SlaterDet<FuncUp, FuncDown>::initNewVmcRun() {

  //Calculate the number of particles with spin up and down
  //and generate initial particle index to column index transform
     array
  dimUp   = 0;
  dimDown = 0;
  for (int i = 0; i < numParticles; i++)
    if (coors[i].spin() > 0)
      columnIndexOfParticle[i] = dimUp++;
    else    // if (coors[i].spin() == -1)
      columnIndexOfParticle[i] = dimDown++;

  //Initialize SlaterMatrix objects
  smUp->init();
  for (int i = 0; i < dimUp; i++) {
    funcsUp[i].calcValueCenter(coors[i]);
    funcsUp[i].valuePt();
    smUp->calcRatio();
    smUp->importNewColumn();
    smUp->setToNextColumn();
  }

  smDown->init();
  for (int i = 0; i < dimDown; i++) {
    funcsDown[i].calcValueCenter(coors[i+dimUp]);
    funcsDown[i].valuePt();
    smDown->calcRatio();
    smDown->importNewColumn();
    smDown->setToNextColumn();
  }
  setCurrentParticle(0);
  resetAcceptances();
  initDiff();
}
template <class FuncUp, class FuncDown>
void SlaterDet<FuncUp, FuncDown>::suggestMove() {
#ifdef _DEBUG_
```

```
  if (moved == 1 || flipped == 1) {
    cerr << "\nCannot move. Change already suggested.\nAccept or
       reject first.\n";
    return;
  }
#endif

  currentColumn = columnIndexOfParticle[currentParticle];

  if (spin > 0) {
    funcsUp[currentColumn].calcValueCenter();
    funcsUp[currentColumn].valuePt();
    smUp->setCurrentColumn(currentColumn);
    smUp->calcRatio();
    ratio    = ratioUp();
    trialDet = trialDetUp();
  }
  else {
    funcsDown[currentColumn].calcValueCenter();
    funcsDown[currentColumn].valuePt();
    smDown->setCurrentColumn(currentColumn);
    smDown->calcRatio();
    ratio    = ratioDown();
    trialDet = trialDetDown();
  }

#ifdef _DEBUG_
  moved = 1;
  valueSidesCalculated = 0;
#endif
}


// ************************ suggestFlip ***************************
template <class FuncUp, class FuncDown>
void SlaterDet<FuncUp, FuncDown>::suggestFlip() {
#ifdef _DEBUG_
  if (moved == 1 || flipped == 1) {
    cerr << "\nCannot flip. Change already suggested.\nAccept or
       reject first.\n";
    return;
  }
#endif

  //Find a particle with a spin opposite of the current particle
```

```
  while ( coors[otherParticle = (int)(random().getNum() * numParticles
     )].spin() == spin );

  currentColumn = columnIndexOfParticle[currentParticle];
  otherColumn   = columnIndexOfParticle[otherParticle];

  if (spin > 0) {
    funcsUp[currentColumn].calcValueCenter(coors[otherParticle]);
    funcsDown[otherColumn].calcValueCenter(coors[currentParticle]);
    funcsUp[currentColumn].valuePt();
    funcsDown[otherColumn].valuePt();
    smUp->setCurrentColumn(currentColumn);
    smDown->setCurrentColumn(otherColumn);
  }
  else {
    funcsDown[currentColumn].calcValueCenter(coors[otherParticle]);
    funcsUp[otherColumn].calcValueCenter(coors[currentParticle]);
    funcsDown[currentColumn].valuePt();
    funcsUp[otherColumn].valuePt();
    smDown->setCurrentColumn(currentColumn);
    smUp->setCurrentColumn(otherColumn);
  }

  smUp->calcRatio();
  smDown->calcRatio();

  ratio     = ratioUp()     * ratioDown();
  trialDet = trialDetUp() * trialDetDown();

#ifdef _DEBUG_
  flipped = 1;
  valueSidesCalculated = 0;
#endif
}


// ********************* suggestMoveFlip ************************
template <class FuncUp, class FuncDown>
void SlaterDet<FuncUp, FuncDown>::suggestMoveFlip() {
#ifdef _DEBUG_
  if (moved == 1 || flipped == 1) {
    cerr << "\nCannot move-flip. Change already suggested.\nAccept or
       reject first.\n";
    return;
  }
```

```
#endif

  //Find a particle with a spin opposite of the current particle
  //while ( coors[otherParticle = (int)(random().getNum() *
     numParticles)].spin() == spin );

  // Chose otherParticle randomly, and if the spins differ flip spin
  // otherParticle = (int)(random().getNum() * numParticles);
  // spinFlip = (coors[otherParticle].spin() == spin);
  if (!spinFlip) suggestMove();
  else {
    otherParticle = domain->getOtherParticle();
    currentColumn = columnIndexOfParticle[currentParticle];
    otherColumn   = columnIndexOfParticle[otherParticle];

    if (spin > 0) {
      funcsUp[currentColumn].calcValueCenter(coors[otherParticle]);
      funcsDown[otherColumn].calcValueCenter();
      funcsUp[currentColumn].valuePt();
      funcsDown[otherColumn].valuePt();
      smUp->setCurrentColumn(currentColumn);
      smDown->setCurrentColumn(otherColumn);
    }
    else {
      funcsDown[currentColumn].calcValueCenter(coors[otherParticle]);
      funcsUp[otherColumn].calcValueCenter();
      funcsDown[currentColumn].valuePt();
      funcsUp[otherColumn].valuePt();
      smDown->setCurrentColumn(currentColumn);
      smUp->setCurrentColumn(otherColumn);
    }

    smUp->calcRatio();
    smDown->calcRatio();

    ratio     = ratioUp()     * ratioDown();
    trialDet = trialDetUp() * trialDetDown();

#ifdef _DEBUG_
  moved = flipped = 1;
  valueSidesCalculated = 0;
#endif
  }

}
```

```
// ************************ acceptMove ****************************
template <class FuncUp, class FuncDown>
void SlaterDet<FuncUp, FuncDown>::acceptMove() {
#ifdef _DEBUG_
  if (flipped == 1 || moved == 0) {
    cerr << "\nCannot accept move. A flip or a move-flip (or none) was
        suggested.\n";
    return;
  }
#endif

  if (spin > 0) {
    smUp->importNewColumn();
    det = detUp();
    //Calculate and store differentiated values of single orbital
    //functions. These values will be needed to calculate the first
       and
    //second derivatives later on.
    funcsUp[currentColumn].calcValueSides();
  }
  else {
    smDown->importNewColumn();
    det = detDown();
    //Calculate and store differentiated values of single orbital
    //functions. These values will be needed to calculate the first
    //and second derivatives later on.
    funcsDown[currentColumn].calcValueSides();
  }

  moveAcceptance++;

#ifdef _DEBUG_
  moved = valueSidesCalculated = 0;
#endif
}


// ************************ acceptFlip ****************************
template <class FuncUp, class FuncDown>
void SlaterDet<FuncUp, FuncDown>::acceptFlip() {
#ifdef _DEBUG_
  if (moved == 1 || flipped == 0) {
```

```
    cerr << "\nCannot accept flip. A move or a move-flip (or none) was
        suggested.\n";
    return;
  }
#endif

  if (spin > 0) {
    funcsUp[currentColumn].calcValueSides(coors[otherParticle]);
    funcsDown[otherColumn].calcValueSides(coors[currentParticle]);
  }
  else {
    funcsDown[currentColumn].calcValueSides(coors[otherParticle]);
    funcsUp[otherColumn].calcValueSides(coors[currentParticle]);
  }

  //Let the slater matrices import their new columns
  smUp->importNewColumn();
  smDown->importNewColumn();
  det = detUp() * detDown();

  //Flip the spin coordinates of the particles involved
  coors[currentParticle].flipSpin();
  coors[otherParticle].flipSpin();

  //Interchange indices in particle to column transform array
  columnIndexOfParticle[currentParticle] = otherColumn;
  columnIndexOfParticle[otherParticle]   = currentColumn;

  flipAcceptance++;

#ifdef _DEBUG_
  flipped = valueSidesCalculated = 0;
#endif
}


// ********************* acceptMoveFlip *************************
template <class FuncUp, class FuncDown>
void SlaterDet<FuncUp, FuncDown>::acceptMoveFlip() {
  if (!*spinFlip) acceptMove();
  else {
#ifdef _DEBUG_
    if (moved == 0 || flipped == 0) {
      cerr << "\nCannot accept move-flip. A move or a flip (or none)
         was suggested.\n";
```

```
      return;
    }
#endif
    if (spin > 0) {
      funcsUp[currentColumn].calcValueSides(coors[otherParticle]);
      funcsDown[otherColumn].calcValueSides();
    }
    else {
      funcsDown[currentColumn].calcValueSides(coors[otherParticle]);
      funcsUp[otherColumn].calcValueSides();
    }

    //Let the slater matrices import their new columns
    smUp->importNewColumn();
    smDown->importNewColumn();
    det = detUp() * detDown();

    //Flip the spin coordinates of the particles involved
    coors[currentParticle].flipSpin();
    coors[otherParticle].flipSpin();

    //Interchange indices in particle to column transform array
    columnIndexOfParticle[currentParticle] = otherColumn;
    columnIndexOfParticle[otherParticle]   = currentColumn;
  }

  moveFlipAcceptance++;

#ifdef _DEBUG_
  moved = flipped = valueSidesCalculated = 0;
#endif

}


// ************************ rejectMove ****************************
template <class FuncUp, class FuncDown>
void SlaterDet<FuncUp, FuncDown>::rejectMove() {
#ifdef _DEBUG_
  if (flipped == 1 || moved == 0) {
    cerr << "\nCannot reject move. A flip or a move-flip (or none) was
        suggested.\n";
    return;
  }
  moved = 0;
```

```
#endif

  if (spin > 0) {
    //Recalculate and store values of single orbital functions. These
    //values will be needed to calculate the first and second
    //derivatives later on.
    funcsUp[currentColumn].calcValueCenter(coors[currentParticle]);
  }
  else {
    //Calculate and store values of single orbital functions. These
    //values will be needed to calculate the first and second
    //derivatives later on.
    funcsDown[currentColumn].calcValueCenter(coors[currentParticle]);
  }

}


// ********************** rejectFlip ***************************
template<class FuncUp, class FuncDown>
void SlaterDet<FuncUp, FuncDown>::rejectFlip() {
#ifdef _DEBUG_
  if (moved == 1 || flipped == 0) {
    cerr << "\nCannot reject flip. A move or a move-flip (or none) was
        suggested.\n";
    return;
  }
  flipped = 0;
#endif

  if (spin > 0) {
    funcsUp[currentColumn].calcValueCenter(coors[currentParticle]);
    funcsDown[otherColumn].calcValueCenter(coors[otherParticle]);
  }
  else {
    funcsDown[currentColumn].calcValueCenter(coors[currentParticle]);
    funcsUp[otherColumn].calcValueCenter(coors[otherParticle]);
  }

}


// ********************** rejectMoveFlip ************************
template<class FuncUp, class FuncDown>
void SlaterDet<FuncUp, FuncDown>::rejectMoveFlip() {
```

```
  if (!*spinFlip) rejectMove();
  else {
#ifdef _DEBUG_
    if (moved == 0 || flipped == 0) {
      cerr << "\nCannot reject move-flip. A move or a flip (or none)
         was suggested.\n";
      return;
    }
    moved = flipped = 0;
#endif

    if (spin > 0) {
      funcsUp[currentColumn].calcValueCenter(coors[currentParticle]);
      funcsDown[otherColumn].calcValueCenter(coors[otherParticle]);
    }
    else {
      funcsDown[currentColumn].calcValueCenter(coors[currentParticle])
         ;
      funcsUp[otherColumn].calcValueCenter(coors[otherParticle]);
    }
  }
}


// ******************** setToNextParticle ************************
template <class FuncUp, class FuncDown>
void SlaterDet<FuncUp, FuncDown>::setToNextParticle() {
#ifdef _DEBUG_
  if (moved == 1 || flipped == 1) {
    cerr << "\nA move, flip or move-flip was suggested.\nAccept or
       reject before setting to next particle.\n";
    return;
  }
#endif

  if (++currentParticle == numParticles)
    currentParticle = 0;

  spin = coors[currentParticle].spin();
}


// ******************** setCurrentParticle ***********************
template <class FuncUp, class FuncDown>
```

```
void SlaterDet<FuncUp, FuncDown>::setCurrentParticle(int
    _currentParticle) {
#ifdef _DEBUG_
  if (moved == 1 || flipped == 1) {
    cerr << "\nA move, flip or move-flip was suggested.\nAccept or
       reject before setting particle index.\n";
    return;
  }
#endif

  currentParticle = _currentParticle;
  spin = coors[currentParticle].spin();
}


// ************************* initDiff ****************************
template <class FuncUp, class FuncDown>
void SlaterDet<FuncUp, FuncDown>::initDiff() {
  for (int i = 0; i < numParticles; i++)
    if (coors[i].spin() > 0) {
      funcsUp[columnIndexOfParticle[i]].calcValueCenter(coors[i]);
      funcsUp[columnIndexOfParticle[i]].calcValueSides(coors[i]);
    }
    else {
      funcsDown[columnIndexOfParticle[i]].calcValueCenter(coors[i]);
      funcsDown[columnIndexOfParticle[i]].calcValueSides(coors[i]);
   }

#ifdef _DEBUG_
  valueSidesCalculated = 1;
#endif
}


// ********************** calcDiffRatios ************************
template <class FuncUp, class FuncDown>
void SlaterDet<FuncUp, FuncDown>::calcDiffRatios() {
#ifdef _DEBUG_
  if (checkIfValueSidesCalculated() == 0) {
    cerr << "\nCannot calculate diffRatios.\n";
    exit(1);
  }
#endif

  _diffRatios = diffRatios;
```

```
  for (int i = 0; i < numParticles; i++)
    if (coors[i].spin() > 0) {
      smUp->setCurrentColumn(columnIndexOfParticle[i]);
      for (int j = 0; j < numDimensions; j++) {
        funcsUp[columnIndexOfParticle[i]].diff(j);
        smUp->calcRatio();
        *_diffRatios++ = ratioUp();
      }
    }
    else {
      smDown->setCurrentColumn(columnIndexOfParticle[i]);
      for (int j = 0; j < numDimensions; j++) {
        funcsDown[columnIndexOfParticle[i]].diff(j);
        smDown->calcRatio();
        (*_diffRatios++) = ratioDown();
      }
    }
}


// ********************** calcDDiffRatio ************************
template <class FuncUp, class FuncDown>
void SlaterDet<FuncUp, FuncDown>::calcDDiffRatio() {
#ifdef _DEBUG_
  if (checkIfValueSidesCalculated() == 0) {
    cerr << "\nCannot calculate dDiffRatio.\n";
    exit(1);
  }
#endif

  dDiffRatio = 0;
  for (int i = 0; i < dimUp; i++) {
    smUp->setCurrentColumn(i);
    funcsUp[i].ddiff();
    smUp->calcRatio();
    dDiffRatio += ratioUp();
  }
  for (int i = 0; i < dimDown; i++) {
    smDown->setCurrentColumn(i);
    funcsDown[i].ddiff();
    smDown->calcRatio();
    dDiffRatio += ratioDown();
  }
}
```

```
// ************************ summary ****************************
template <class FuncUp, class FuncDown>
void SlaterDet<FuncUp, FuncDown>::summary() {
  cerr << "\n\n-------------------------------------------------";
  cerr << "\n SUMMARY OF SlaterDet INSTANCE\n\n";

  smUp->summary();
  smDown->summary();
}


// ********************** resetAcceptances ***********************
template <class FuncUp, class FuncDown>
void SlaterDet<FuncUp, FuncDown>::resetAcceptances() {
  moveAcceptance = flipAcceptance = moveFlipAcceptance = 0;
}


#ifdef _DEBUG_
template <class FuncUp, class FuncDown>
int SlaterDet<FuncUp, FuncDown>::checkIfValueSidesCalculated() {
  /*
  if (valueSidesCalculated == 0) {
    cerr << "\nvalueSides not yet calculated.";
    return 0;
  }
  */
  return 1;
}
#endif

#endif
```

A.1.13 SlaterMatrix.h

```
#ifndef SlaterMatrix_IS_INCLUDED
#define SlaterMatrix_IS_INCLUDED

#include <string>
#include <iostream>
#include <cstdio>
#include <cstdlib>

using namespace std;

// ************************************************************
// *                          SLATERMATRIX                    *
// ************************************************************
class SlaterMatrix {

 protected:
  //Primary variables
  double* inverseMatrix;
  double* inverseMatrixCurrentRow;
  double* newColumn;
  double  ratio;
  int     dim;
  int     currentColumn;
  int     isAllocated;
#ifdef _DEBUG_
  int     ratioIsCalculated;
#endif

  //Iterating and secondary variables
  double* _inverseMatrix;
  double* _inverseMatrixCurrentRow;
  double* _newColumn;
  double  result;
  double  invRatio;

 public:
  SlaterMatrix();
  SlaterMatrix(int __dim);
  virtual        ~SlaterMatrix();
  void            init();
  void            redim(int __dim);
  void    setPtrToNewColumn(double* __newColumn);
  virtual void    setCurrentColumn(int __currentColumn);
```

```
  virtual void     setToNextColumn();
  virtual void     resetPtr();
  virtual void     calcRatio();
  virtual void     importNewColumn();
  void             importNewMatrixColumnwise(double* __matrix);
  virtual void     setMatrixToUnity();

  virtual void     summary();

  double&          getRatio()         {return ratio;}
  int              getDim()           {return dim;}
  double*          getInvMatrixPtr()  {return inverseMatrix;}

 protected:
  virtual int      allocateMemory(int __dim);
  virtual void     deleteMemory();
#ifdef _DEBUG_
  int      checkAllocation();
  int      checkIfRatioCalculated();
#endif

};


// ************************************************************
// *                        SLATERMATRIX_DET                  *
// ************************************************************
class SlaterMatrix_Det : public SlaterMatrix {

 protected:
  double trialDeterminant;
  double determinant;

 public:
  SlaterMatrix_Det();
  SlaterMatrix_Det(int __dim);
  virtual          ~SlaterMatrix_Det();
  inline void       importNewColumn();
  void              setMatrixToUnity();
  inline void       calcRatio();

  virtual void      summary();

  double&           getDet()          {return determinant;}
  double&           getTrialDet()     {return trialDeterminant;}
```

```
};

#endif
```

A.1.14 SlaterMatrix.cc

```
#define TRUE 1
#define FALSE 0
#define NULL 0

#include "SlaterMatrix.h"

// ************************************************************
// *                          SLATERMATRIX                    *
// ************************************************************
//
// ********************* SlaterMatrix ***********************
SlaterMatrix::SlaterMatrix() {
#ifdef _DEBUG_
  ratioIsCalculated = FALSE;
#endif

  isAllocated       = FALSE;
}

// ********************* SlaterMatrix ***********************
SlaterMatrix::SlaterMatrix(int __dim) {
  isAllocated       = FALSE;
#ifdef _DEBUG_
  ratioIsCalculated = FALSE;
#endif

  redim(__dim);
}


// ********************* ~SlaterMatrix ***********************
SlaterMatrix::~SlaterMatrix() {
  deleteMemory();
}


// ************************** init ****************************
void SlaterMatrix::init() {
  setMatrixToUnity();
  setCurrentColumn(0);
```

```
}


// ************************** redim ****************************
void SlaterMatrix::redim(int __dim) {
  deleteMemory();
  allocateMemory(__dim);
  init();
}


// ********************** setPtrToNewColumn ***********************
void SlaterMatrix::setPtrToNewColumn(double* __newColumn) {
#ifdef _DEBUG_
  if (newColumn != __newColumn)
    ratioIsCalculated = FALSE;
#endif

  newColumn = __newColumn;
}


// ********************** setCurrentColumn ************************
void SlaterMatrix::setCurrentColumn(int __currentColumn) {
#ifdef _DEBUG_
  if (checkAllocation() == 1) exit(1);
  if (currentColumn != __currentColumn)
    ratioIsCalculated = FALSE;
#endif

  currentColumn = __currentColumn;
  inverseMatrixCurrentRow = (inverseMatrix + dim*currentColumn);
}


// ********************** setToNextColumn ************************
void SlaterMatrix::setToNextColumn() {
#ifdef _DEBUG_
  if (checkAllocation() == 1) exit(1);
#endif

  if (++currentColumn == dim) {
    currentColumn = 0;
    resetPtr();
  }
```

```
  else
    inverseMatrixCurrentRow += dim;

#ifdef _DEBUG_
  ratioIsCalculated = FALSE;
#endif
}


// ************************* resetPtr *************************
void SlaterMatrix::resetPtr() {
  inverseMatrixCurrentRow = inverseMatrix;
}


// ************************* calcRatio *************************
void SlaterMatrix::calcRatio() {
#ifdef _DEBUG_
  if (checkAllocation() == 1) {
    cerr << "\nCannot calculate determinant ratio.";
    exit(1);
  }
#endif

  _newColumn                = newColumn;
  _inverseMatrixCurrentRow = inverseMatrixCurrentRow;

  ratio = 0;
  for (int i = 0; i < dim; i++)
    ratio += ( (*_newColumn++) * (*_inverseMatrixCurrentRow++));

#ifdef _DEBUG_
  if (ratio == 0)
    cerr << "\nRatio = 0.\n If new column is imported, slater matrix
       will not beinvertible.\n";
  ratioIsCalculated = TRUE;
#endif
}


// ********************** importNewColumn ***********************
void SlaterMatrix::importNewColumn() {
#ifdef _DEBUG_
  if (checkAllocation() == 1) return;
  if (checkIfRatioCalculated() == 1) {
```

```
    cerr << "\nCannot update inverse matrix. Determinant ratio not
       calculated.";
    exit(1);
  }
#endif

  _inverseMatrix            = inverseMatrix;
  _inverseMatrixCurrentRow = inverseMatrixCurrentRow + dim;
  invRatio = 1/ratio;

  for (int i = 0; i < dim; i++)
    if (i != currentColumn) {
      result      = 0;
      _newColumn = newColumn;

      for (int j = 0; j < dim; j++)
        result += ((*_newColumn++) * (*_inverseMatrix++));

      result *= invRatio;

      _inverseMatrix            -= dim;
      _inverseMatrixCurrentRow -= dim;
      for (int j = 0; j < dim; j++)
        (*_inverseMatrix++) -= ((*_inverseMatrixCurrentRow++) * result
           );
    }
    else
      _inverseMatrix += dim;

  _inverseMatrixCurrentRow -= dim;
  for (int i = 0; i < dim; i++)
    (*_inverseMatrixCurrentRow++) *= invRatio;

#ifdef _DEBUG_
  ratioIsCalculated = FALSE;
#endif

}


// ***************** importNewMatrixColumnwise ******************
void SlaterMatrix::importNewMatrixColumnwise(double* __matrix) {
#ifdef _DEBUG_
  if (checkAllocation() == 1) {
    cerr << "\nCannot import new matrix.";
```

```
    exit(1);
  }
#endif

  double* __newColumn      = newColumn;
  int     __currentColumn = currentColumn;

  setCurrentColumn(0);
  for (int i = 0; i < dim; i++) {
    setPtrToNewColumn(__matrix);
    calcRatio();
    importNewColumn();
    setToNextColumn();
    __matrix += dim;
  }

  setPtrToNewColumn(__newColumn);
  setCurrentColumn(__currentColumn);

#ifdef _DEBUG_
  ratioIsCalculated = FALSE;
#endif
}


// ********************* setMatrixToUnity ************************
void SlaterMatrix::setMatrixToUnity() {
#ifdef _DEBUG_
  if (checkAllocation() == 1) {
    cerr << "\nCannot set matrix to unity.";
    exit(1);
  }
#endif

  _inverseMatrix = inverseMatrix;

  for (int i = 0; i < dim; i++)
    for (int j = 0; j < dim; j++)
      *_inverseMatrix++ = ( j==i );

#ifdef _DEBUG_
  ratioIsCalculated = FALSE;
#endif
}
```

```
// ************************** summary ****************************
void SlaterMatrix::summary() {
  string result;
  char   buffer[50];

#ifdef _DEBUG_
  if (checkAllocation() == 1)
    exit(1);
#endif

  cerr << "\n--------------------------------------------------";
  cerr << "\nDimension: " << dim << "\n";
  cerr << "\nInverse slater matrix:\n\n";

  for (int i = 0; i < dim; i++) {
    for (int j = 0; j < dim; j++) {
      snprintf(buffer, 50, "%10.5f", inverseMatrix[i*dim + j]);
      result += buffer;
      if (j != (dim-1))
        result += ", ";
    }
    result += "\n";
  }

  cerr << result;

}


// ********************** allocateMemory ************************
int SlaterMatrix::allocateMemory(int __dim) {
  if (isAllocated) return 1;

  inverseMatrix = new double[__dim*__dim+1];
  dim = __dim;

  isAllocated = TRUE;

  return 0;
}


// ************************ deleteMemory ************************
void SlaterMatrix::deleteMemory() {
```

```
  if (isAllocated)
    delete[] inverseMatrix;
  isAllocated = FALSE;
}


#ifdef _DEBUG_


// ********************** checkAllocation ************************
int SlaterMatrix::checkAllocation() {
  if (!isAllocated) {
    cerr << "\nNo memory allocated for inverse slater matrix.";
    return 1;
  }

  return 0;
}


// ******************** checkIfRatioCalculated *******************
int SlaterMatrix::checkIfRatioCalculated() {
  if (!ratioIsCalculated) {
    cerr << "\nDeterminant ratio not yet calculated.";
    return 1;
  }

  return 0;
}

#endif


// ***************************************************************
// *                        SLATERMATRIX_DET                     *
// ***************************************************************
//
// ********************** SlaterMatrix_Det ************************
SlaterMatrix_Det::SlaterMatrix_Det() : SlaterMatrix() {
}


// ********************** SlaterMatrix_Det ************************
SlaterMatrix_Det::SlaterMatrix_Det(int __dim) : SlaterMatrix(__dim){
}
```

```
// ********************** ~SlaterMatrix_Det ***********************
SlaterMatrix_Det::~SlaterMatrix_Det() {
  deleteMemory();
}


// ************************** calcRatio ***************************
void SlaterMatrix_Det::calcRatio() {
  SlaterMatrix::calcRatio();
  trialDeterminant = ratio*determinant;
}


// ************************ importNewColumn ************************
void SlaterMatrix_Det::importNewColumn() {
  SlaterMatrix::importNewColumn();
  determinant = trialDeterminant;
}


// ********************** setMatrixToUnity ************************
void SlaterMatrix_Det::setMatrixToUnity() {
  SlaterMatrix::setMatrixToUnity();
  determinant = 1.;
}


// ************************** summary *****************************
void SlaterMatrix_Det::summary() {
  SlaterMatrix::summary();
  cerr << "\nDeterminant: " << determinant << endl;
}
```

A.1.15 Func.h

```
#ifndef Func_IS_INCLUDED
#define Func_IS_INCLUDED

#include <string>
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include "../Coor/Coor.h"
#include "../Functor/Functor.h"
#include "../Ref/Ref.h"
#include "../Domain/Domain.h"
#include "../SingleParticleFuncs/SingleParticleFuncs.h"

using namespace std;


// ****************************************************************
// *                              FUNC                            *
// ****************************************************************
template <class Param>
class Func {

 protected:
  Ref<Param>     coordinate;
  SingleParticleFunc<Param>** function;
  double         h, h_half, h_double, h_inv, h_inv2, h_invdouble;
  double*        values;
  double*        result;
  double*        diffResult;
  double*        ddiffResult;
  int            numDimensions;

 public:
  Func(Param& _coordinate);
  Func();
  virtual ~Func();
  virtual void   init();
  virtual void   init(Param& _coordinate);
  virtual void   attachNumDimensions(int _numDimensions)
    { numDimensions = _numDimensions; }
  virtual void   setCoordinate(Param& _coordinate);
  virtual void   calcValueCenter();
  virtual void   calcValueCenter(Param& _coordinate);
```

```
  virtual void   calcValueSides();
  virtual void   calcValueSides(Param& _coordinate);
  virtual double valuePt();
  virtual double diff();
  virtual double ddiff();
  virtual void   attachResult(double* __result);
  virtual void   attachDiffResult(double* __diffResult);
  virtual void   attachDdiffResult(double* __ddiffResult);
  virtual Param& getCoordinate() {return coordinate();}
  virtual void   summary();

 protected:
  virtual void   calcH();

};


// ************************************************************
// *                          FUNCSET                         *
// ************************************************************
template <class Param>
class FuncSet : public Func<Param> {

 protected:
  int          len;
  double*      valuesM;
  double*      valuesP;

 public:
  FuncSet(Param& _coordinate, int _len);
  FuncSet();
  virtual ~FuncSet();
  virtual void   init();
  virtual void   init(Param& _coordinate, int _len);
  virtual void   calcValueCenter();
  virtual void   calcValueCenter(Param& _coordinate);
  virtual void   calcValueSides();
  virtual void   calcValueSides(Param& _coordinate);
  virtual double valuePt();
  virtual double diff();
  virtual double ddiff();
  virtual void   summary();

};
```

```
// ************************************************************
// *                        FUNCSETMULTIVAR                   *
// ************************************************************
template <class Param>
class FuncSetMultivar : public FuncSet<Param> {
 protected:
  int       numVar;
  int       len_double;
  double*   _diffResult;

 public:
  FuncSetMultivar(Param& _coordinate, int _len);
  FuncSetMultivar();
  virtual ~FuncSetMultivar();
  virtual void   init();
  virtual void   init(Param& _coordinate, int _len);
  virtual void   calcValueSides();
  virtual void   calcValueSides(Param& _coordinate);
  virtual double diff();
  virtual double diff(int v);
  virtual double ddiff();
  virtual void   attachDiffResult(double* __diffResult);
  virtual void   summary();
};

#include "Func.cpp"

#endif
```

A.1.16 Func.cpp

```
#ifndef FuncCPP_IS_INCLUDED
#define FuncCPP_IS_INCLUDED

#include "Func.h"


// ************************************************************
// *                          FUNC                            *
// ************************************************************
//
// ************************** Func ****************************
template <class Param>
Func<Param>::Func(Param& _coordinate) {
  setCoordinate(_coordinate);
```

```
  calcH();
  numDimensions = 1;
}

template <class Param>
Func<Param>::Func() {
  calcH();
  numDimensions = 1;
}


// ************************** ~Func ******************************
template <class Param>
Func<Param>::~Func() {
  delete[] function[0];
  delete[] values;
}


// *************************** init ******************************
template <class Param>
void Func<Param>::init(Param& _coordinate) {
  setCoordinate(_coordinate);
  init();
}

template <class Param>
void Func<Param>::init() {
  function = new SingleParticleFunc<Param>*;
  values   = new double[3];
}


// ********************** setCoordinate **************************
template <class Param>
void Func<Param>::setCoordinate(Param& _coordinate) {
  coordinate = _coordinate;
}


// ************************** calcH ******************************
template <class Param>
void Func<Param>::calcH() {
  h           = 1e-4;
  h_half      = 0.5*h;
```

```
  h_double     = 2.0*h;
  h_inv        = 1./h;
  h_inv2       = 1./(h*h);
  h_invdouble = 1./(2.*h);
}


// ********************* calcValueCenter ************************
template <class Param>
void Func<Param>::calcValueCenter() {
  (*values) = (**function)(coordinate());
}

template <class Param>
void Func<Param>::calcValueCenter(Param& _coordinate) {
  (*values) = (**function)(_coordinate);
}


// ********************* calcValueSides *************************
template <class Param>
void Func<Param>::calcValueSides() {
  values[1] = (**function)(coordinate(), 0);
  values[2] = (**function)(coordinate(), 1);
}

template <class Param>
void Func<Param>::calcValueSides(Param& _coordinate) {
  values[1] = (**function)(_coordinate, 0);
  values[2] = (**function)(_coordinate, 1);
}


// ************************* valuePt ****************************
template <class Param>
double Func<Param>::valuePt() {
  return (*result = values[0]);
}


// ************************** diff ******************************
template <class Param>
double Func<Param>::diff() {
  *diffResult = (values[1] - values[2])*h_invdouble;
  return *diffResult;
```

```
}


// ************************** ddiff ******************************
template <class Param>
double Func<Param>::ddiff() {
  *ddiffResult = (values[1] + values[2] - 2*values[0])*h_inv2;
  return *ddiffResult;
}


// ********************** attachResult ***************************
template <class Param>
void Func<Param>::attachResult(double* __result) {
  result = __result;
}


// ********************* attachDiffResult ************************
template <class Param>
void Func<Param>::attachDiffResult(double* __diffResult) {
  diffResult = __diffResult;
}


// ********************* attachDdiffResult ***********************
template <class Param>
void Func<Param>::attachDdiffResult(double* __ddiffResult) {
  ddiffResult = __ddiffResult;
}


// ************************** summary ****************************
template <class Param>
void Func<Param>::summary() {
  cerr << "\n\n-------------------------------------------------\n";
  cerr << "Values:\n";
  cerr << " Center: " << values[0] << endl;
  cerr << " Minus:  " << values[1] << endl;
  cerr << " Plus:   " << values[2] << endl;
  cerr << "result:      "  << *result << endl;
  cerr << "diffResult:  "  << *diffResult << endl;
  cerr << "ddiffResult: "  << *ddiffResult << endl;
}
```

```
// ************************************************************
// *                          FUNCSET                         *
// ************************************************************
//
// ************************* FuncSet ****************************
template <class Param>
FuncSet<Param>::FuncSet(Param& _coordinate, int _len) :
  Func<Param>(_coordinate), len(_len) {}

template <class Param>
FuncSet<Param>::FuncSet() : Func<Param>() {}


// ************************* ~FuncSet ***************************
template <class Param>
FuncSet<Param>::~FuncSet() {
  for (int i=0; i<len; i++)
    delete[] function[i];
  delete[] values;
}


// ************************** init ******************************
template <class Param>
void FuncSet<Param>::init(Param& _coordinate, int _len) {
  len = _len;
  Func<Param>::init(_coordinate);
}

template <class Param>
void FuncSet<Param>::init() {
  values   = new double[3*len];
  valuesP  = values + len;
  valuesM  = values + len*2;
  function = new SingleParticleFunc<Param>*[len];
}


// ********************* calcValueCenter ************************
template <class Param>
void FuncSet<Param>::calcValueCenter() {
  for (int i = 0; i < len; i++)
```

```
    values[i] = function[i][0](coordinate());
}

template <class Param>
void FuncSet<Param>::calcValueCenter(Param& _coordinate) {
  for (int i = 0; i < len; i++)
    values[i] = function[i][0](_coordinate);
}


// ********************* calcValueSides *************************
template <class Param>
void FuncSet<Param>::calcValueSides() {

  for (int i = 0; i < len; i++) {
    valuesP[i] = function[i][0](coordinate(), 0);
    valuesM[i] = function[i][0](coordinate(), numDimensions );
  }
}

template <class Param>
void FuncSet<Param>::calcValueSides(Param& _coordinate) {

  for (int i = 0; i < len; i++) {
    valuesP[i] = function[i][0](_coordinate, 0);
    valuesM[i] = function[i][0](_coordinate, numDimensions );
  }
}


// ************************ valuePt ****************************
template <class Param>
double FuncSet<Param>::valuePt() {
  for (int i = 0; i < len; i++)
    result[i] = values[i];

  return 0;
}


// ************************** diff *****************************
template <class Param>
double FuncSet<Param>::diff() {
  for (int i = 0; i < len; i++)
    diffResult[i] = (valuesP[i]-valuesM[i])*h_invdouble;
```

```
  return 0;
}


// ************************** ddiff ******************************
template <class Param>
double FuncSet<Param>::ddiff() {
  for (int i = 0; i < len; i++)
    ddiffResult[i] = (valuesM[i]+valuesP[i]-2*values[i])*h_inv2;

  return 0;
}


// ************************** summary ****************************
template <class Param>
void FuncSet<Param>::summary() {
  string returnStr;
  char   buffer[50];

  cerr << "\n\n-----------------------------------------------------\n";
  cerr << "Values:\n\n";
  for (int i = 0; i < len; i++) {
    snprintf(buffer, 50, "%5.5f  %5.5f  %5.5f\n", values[i], values[i+
       len], values[i+2*len]);
    returnStr += buffer;
  }
  cerr << endl;

  returnStr = "";
  cerr << "Result:\n";
  for (int i = 0; i < len; i++) {
    snprintf(buffer, 50, "%5.5f", result[i]);
    returnStr += buffer;
    returnStr += "\n";
  }
  cerr << returnStr;
}


/*************************************************************/


template <class Param>
```

```
FuncSetMultivar<Param>::FuncSetMultivar(Param& _coordinate, int _len)
    : FuncSet<Param>(_coordinate, _len) {
  numVar     = coordinate.getLen();
  len_double = 2*_len;
}


template <class Param>
FuncSetMultivar<Param>::FuncSetMultivar() : FuncSet<Param>() {
}

template <class Param>
FuncSetMultivar<Param>::~FuncSetMultivar() {
  for (int i=0; i<len; i++)
    delete[] function[i];
  //delete[] function;
  delete[] values;
}


// ***************************** init ******************************
template <class Param>
void FuncSetMultivar<Param>::init() {
  values   = new double[(1+2*numVar)*len];
  valuesP  = values + len;
  valuesM  = values + len*2;
  function = new SingleParticleFunc<Param>*[len];
}

template <class Param>
void FuncSetMultivar<Param>::init(Param& _coordinate, int _len) {
  numVar     = _coordinate.getLen();
  len_double = 2*_len;
  FuncSet<Param>::init(_coordinate, _len);
}


// ********************* calcValueSides ************************
template <class Param>
void FuncSetMultivar<Param>::calcValueSides() {
  //  for (int i = (numVar-1); i >= 0; i--) {
  for (int i = 0; i < numVar; i++) {

    int iNum = i + numVar;
    for (int j = 0; j < len; j++) {
```

```
      valuesP[j] = function[j][0](coordinate(), i);
      valuesM[j] = function[j][0](coordinate(), iNum );
    }
    valuesP += (len_double);
    valuesM += (len_double);
  }
  valuesP = values + len;
  valuesM = values + len_double;
}

template <class Param>
void FuncSetMultivar<Param>::calcValueSides(Param& _coordinate) {
  for (int i = 0; i < numVar; i++) {
    int iNum = i + numVar;
    for (int j = 0; j < len; j++) {
      valuesP[j] = function[j][0](_coordinate, i);
      valuesM[j] = function[j][0](_coordinate, iNum );
    }
    valuesP += (len_double);
    valuesM += (len_double);
  }
  valuesP = values + len;
  valuesM = values + len_double;
}


// *************************** diff ******************************
template <class Param>
double FuncSetMultivar<Param>::diff() {
  for (int v = (numVar-1); v > 0; v--) {
    FuncSet<Param>::diff();
    diffResult += len;
    valuesM    += (len_double);
    valuesP    += (len_double);
  }
  FuncSet<Param>::diff();

  diffResult = _diffResult;
  valuesP = values + len;
  valuesM = values + len_double;


  return 0;
}
```

```
template <class Param>
double FuncSetMultivar<Param>::diff(int v) {
  //Calculate the first derivative with respect to the variable
     indexed
  //v of all the functions and place the result in the result array (!
     !!)
#ifdef _DEBUG_
  if (v >= numVar) {
    cerr << "\nCannot calculate first derivative. Variable index out
       of bound.";
    exit(1);
  }
#endif

  diffResult  = result;
  valuesM     += (len_double*v);
  valuesP     += (len_double*v);
  FuncSet<Param>::diff();

  diffResult = _diffResult;
  valuesP    = values + len;
  valuesM    = values + len_double;

  return 0;
}


// ************************** ddiff ******************************
template <class Param>
double FuncSetMultivar<Param>::ddiff() {
  for (int i = 0; i < len; i++)
    ddiffResult[i] = -(2*numVar*values[i]);
  for (int v = (numVar-1); v > 0; v--) {
    for (int i = 0; i < len; i++)
      ddiffResult[i] += (valuesM[i] + valuesP[i]);
    valuesP += (len_double);
    valuesM += (len_double);
  }
  for (int i = 0; i < len; i++) {
    ddiffResult[i] += (valuesM[i] + valuesP[i]);
    ddiffResult[i] *= h_inv2;
  }

  valuesP = values + len;
  valuesM = values + len_double;
```

```
  return 0;
}


// ********************* attachDiffResult ************************
template <class Param>
void FuncSetMultivar<Param>::attachDiffResult(double* __diffResult) {
  FuncSet<Param>::attachDiffResult(__diffResult);
  _diffResult = __diffResult;
}


// ************************* summary *****************************
template <class Param>
void FuncSetMultivar<Param>::summary() {
  string returnStr;
  char   buffer[50];

  cerr << "\n\n-------------------------------------------------\n";
  cerr << "Values:\n\n";
  for (int i = 0; i < len; i++) {
    snprintf(buffer, 50, "%5.5f  ", values[i]);
    returnStr += buffer;
    for (int j = 0; j < numVar; j++) {
      snprintf(buffer, 50, "  %5.5f  %5.5f", valuesM[i+j*2*len],
          valuesP[i+j*2*len]);
      returnStr += buffer;
    }
    returnStr += "\n";
  }
  cerr << returnStr << endl;

  returnStr = "";
  cerr << "Result:\n";
  for (int i = 0; i < len; i++) {
    snprintf(buffer, 50, "%5.5f", result[i]);
    returnStr += buffer;
    returnStr += "\n";
  }

  cerr << returnStr;

}
```

```
#endif
```

A.1.17 FuncUpDown.h

```
#ifndef FuncUpDown_IS_INCLUDED
#define FuncUpDown_IS_INCLUDED

#include <cmath>
#include <iostream>
#include <fstream>
#include "../Coor/Coor.h"
#include "../SingleParticleFuncs/SingleParticleFuncs.h"
#include "../Domain/Domain.h"
#include "../Func/Func.h"

// ************************************************************
// *                          FUNCUP                          *
// ************************************************************
template <class Param>
class FuncUp : public FuncSetMultivar<Param> {
 protected:
  char* fixedParamsUp;

 public:
  FuncUp() {};
  virtual ~FuncUp()
    {
      for (int i=0; i<len; i++)
        delete function[i];
      delete function;
      delete[] values;
    }

  FuncUp(Param& coordinate, int __len) :
    FuncSetMultivar<Param>(coordinate, __len) {}

  virtual void init(Param& coordinate, int __len, Domain* domain,
                    int alphaVar)
    {
      fixedParamsUp    = domain->getFixedParamsUp();
      FuncSetMultivar<Param>::init(coordinate, __len);
      init(domain, alphaVar);
    }

  virtual void init() {
    FuncSetMultivar<Param>::init();
  }
```

```
  virtual void init(Domain* domain, int alphaVar) {
    int index = 0;
    ifstream ifile;
    ifile.open(fixedParamsUp);
    if (domain->getOrbitalType() == "Hydrogen") {
      if (domain->getUp1s())  {function[index]=new Hydr1s<Param>;
          index++;}
      if (domain->getUp2s())  {function[index]=new Hydr2s<Param>;
          index++;}
      if (domain->getUp2px()) {function[index]=new Hydr2px<Param>;
         index++;}
      if (domain->getUp2py()) {function[index]=new Hydr2py<Param>;
         index++;}
      if (domain->getUp2pz()) {function[index]=new Hydr2pz<Param>;
         index++;}
      if (index != len )
        cerr << "Error creating FuncUp!\nNot matching dimensions of
           number "
             << "particles spin up (" << len
             << ") and number of input functions\n";
      for (int i=0; i<len; i++) {
        function[i]->attach(domain->getNumAlpha(),
                            domain->getAlphaParam(alphaVar));
      }
    }
    else if (domain->getOrbitalType() == "HartreeFock") {
      for (int i=0; i<len; i++) {
        function[i] = new HF<Param>;
        function[i]->readHFparams(ifile);
      }
    }
    for (int i=0; i<len; i++)
      function[i]->setNumberDimensions(domain->getNumDimensions());
    ifile.close();
  }
};


// ****************************************************************
// *                          FUNCDOWN                            *
// ****************************************************************
template <class Param>
class FuncDown : public FuncSetMultivar<Param> {
 protected:
```

```
  char* fixedParamsDown;

public:
 FuncDown() {};
 virtual ~FuncDown()
   {
     for (int i=0; i<len; i++)
       delete[] function[i];
     delete[] values;
   }
 FuncDown(Param& coordinate, int __len) :
   FuncSetMultivar<Param>(coordinate, __len) {}

 virtual void init(Param& coordinate, int __len,
                   Domain* domain, int alphaVar)
   {
     fixedParamsDown = domain->getFixedParamsDown();
     FuncSetMultivar<Param>::init(coordinate, __len);
     init(domain, alphaVar);
   }

 virtual void init() {
   FuncSetMultivar<Param>::init();
 }

 virtual void init(Domain* domain, int alphaVar) {
   int index = 0;
   ifstream ifile;
   ifile.open(fixedParamsDown);
   if (domain->getOrbitalType() == "Hydrogen") {
     if (domain->getDown1s())  {function[index]=new Hydr1s<Param>;
         index++;}
     if (domain->getDown2s())  {function[index]=new Hydr2s<Param>;
         index++;}
     if (domain->getDown2px()) {function[index]=new Hydr2px<Param>;
        index++;}
     if (domain->getDown2py()) {function[index]=new Hydr2py<Param>;
        index++;}
     if (domain->getDown2pz()) {function[index]=new Hydr2pz<Param>;
        index++;}
     if (index != len ) cerr << "Error creating FuncUp!\nNot matching
        "
                             << " dimensions of number particles spin
                                up ("
```

```
                                << len << ") and number of input
                                   functions\n";
      for (int i=0; i<len; i++)
        function[i]->attach(domain->getNumAlpha(),
                            domain->getAlphaParam(alphaVar));
    }
    else if (domain->getOrbitalType() == "HartreeFock") {
      for (int i=0; i<len; i++) {
        function[i] = new HF<Param>;
        function[i]->readHFparams(ifile);
      }
    }
    for (int i=0; i<len; i++)
      function[i]->setNumberDimensions(domain->getNumDimensions());
    ifile.close();
  }
};

#endif
```

A.1.18 SingleParticleFuncs.h

```
#ifndef SingleParticleFuncs_IS_INCLUDED
#define SingleParticleFuncs_IS_INCLUDED

#include <cmath>
#include <iostream>
#include <fstream>
#include "../Ref/Ref.h"
#include "../Coor/Coor.h"
#include "../Domain/Domain.h"
#include "../SolidHarmonics/SolidHarmonics.h"
#include "../STOBasis/STOBasis.h"

// ************************************************************
// *                        SINGELPARTICLEFUNC                *
// ************************************************************
template <class Param>
class SingleParticleFunc {

 protected:
  double*  param;
  virtual  inline double phi(Param& coordinate) {return 0;}
  int      centerNr;
  int      currentNr;

 public:
  SingleParticleFunc() {}
  virtual ~SingleParticleFunc() {}
  virtual void   attach(int _numParams, double* _param)
    { param=_param;}
  virtual void   setNumberDimensions(int numDimensions)
    { centerNr = 2*numDimensions;}
  virtual double operator()(Param& coordinate)
    { currentNr = centerNr;
    return phi(coordinate);}
  virtual double operator()(Param& coordinate, int nr)
    { currentNr = nr;
    return phi(coordinate);}
  virtual void readHFparams(ifstream& ifile) {;}
};


//************************************************************************
```

```
//*                  3-dimensional hydrogen orbitals
                       *
//*********************************************************************

//********** Hydr1s *********
template <class Param>
class Hydr1s : public SingleParticleFunc<Param> {
 protected:
  virtual inline double phi(Param& coordinate) {
    return exp(- param[0] * coordinate.r(currentNr));
  }
 public:
  Hydr1s() {}
  virtual ~Hydr1s() {}
};
//********** Hydr2s *********
template <class Param>
class Hydr2s : public SingleParticleFunc<Param> {
 protected:
  virtual inline double phi(Param& coordinate) {
    return ((2 - param[0]*coordinate.r(currentNr)) * exp(-0.5*
        coordinate.r(currentNr)));
  }
 public:
  Hydr2s() {}
  virtual ~Hydr2s() {}
};
//********** Hydr2px *********
template <class Param>
class Hydr2px : public SingleParticleFunc<Param> {
 protected:
  virtual inline double phi(Param& coordinate) {
    return param[0]*coordinate(0, currentNr) * exp(-0.5*coordinate.r(
        currentNr)*param[0]);
  }
 public:
  Hydr2px() {}
  virtual ~Hydr2px() {}
};
//********** Hydr2py *********
template <class Param>
class Hydr2py : public SingleParticleFunc<Param> {
 protected:
  virtual inline double phi(Param& coordinate) {
```

```
    return param[0]*coordinate(1, currentNr) * exp(-0.5*coordinate.r(
        currentNr)*param[0]);
  }
 public:
  Hydr2py() {}
  virtual ~Hydr2py() {}
};
//********** Hydr2pz *********
template<class Param>
class Hydr2pz : public SingleParticleFunc<Param> {
 protected:
  virtual inline double phi(Param& coordinate) {
    return param[0]*coordinate(2, currentNr) * exp(-0.5*coordinate.r(
        currentNr)*param[0]);
  }
 public:
  Hydr2pz() {}
  virtual ~Hydr2pz() {}
};


//*************************************************************************

//*                3-dimensional Hartree-Fock orbitals
                     *
//*************************************************************************

//
template<class Param>
class HF : public SingleParticleFunc<Param> {
 protected:
  double*                          coefficients;
  int                              numCoeff;
  int                              numParams;
  Ref<SolidHarmonics<Param> > solidHarmonics;
  Ref<STOBasis<Param> >        STOfuncs;

  virtual inline double z1(Param& coordinate) {return 0;}
  virtual inline double z2(Param& coordinate) {return 0;}
  virtual inline double z3(Param& coordinate) {return 0;}
  virtual inline double z4(Param& coordinate) {return 0;}
  virtual inline double z5(Param& coordinate) {return 0;}
  virtual inline double z6(Param& coordinate) {return 0;}
  virtual inline double z7(Param& coordinate) {return 0;}
  virtual inline double z8(Param& coordinate) {return 0;}
```

```
  virtual inline double z9(Param& coordinate) {return 0;}
  virtual inline double z10(Param& coordinate) {return 0;}

  virtual inline double phi2(Param& coordinate) {
    return param[0]*z1(coordinate) + param[1]*z2(coordinate) +
      param[2]*z3(coordinate) + param[3]*z4(coordinate) +
      param[4]*z5(coordinate) + param[5]*z6(coordinate);
  }
  virtual inline double phi(Param& coordinate) {
    double result = 0;
    for (int i=0; i<numCoeff; i++)
      result += coefficients[i] * solidHarmonics()(coordinate,i,
         currentNr)
        * STOfuncs()(coordinate, i, currentNr);
    return result;
  }
public:
 HF() {}

  virtual void readHFparams(ifstream& ifile)
    {
      ifile >> numCoeff;

      int*    OrbN = new int[numCoeff];
      int     OrbL;
      int     OrbM;
      double* Xsi  = new double[numCoeff];
      coefficients = new double[numCoeff];

      for (int i=0; i<numCoeff; i++)
        ifile >> OrbN[i];
      ifile >> OrbL;
      ifile >> OrbM;
      for (int i=0; i<numCoeff; i++)
        ifile >> Xsi[i];
      for (int i=0; i<numCoeff; i++)
        ifile >> coefficients[i];

      solidHarmonics = new SolidHarmonics<Param>;
      STOfuncs       = new STOBasis<Param>;
      solidHarmonics().init(numCoeff, OrbL, OrbM);
      STOfuncs().init(numCoeff, OrbN, OrbL, Xsi);
      delete OrbN;
      delete Xsi;
    }
```

```
};

#endif
```

A.1.19 STOBasis.h

```
#ifndef STOBasis_IS_INCLUDED
#define STOBasis_IS_INCLUDED

#include "../STOBasisFuncs/STOBasisFuncs.h"

// ************************************************************
// *                          STOBASIS                        *
// ************************************************************
template <class Param>
class STOBasis {

 protected:
  STOBasisFuncs<Param>* STOfuncs;
  double constant, expConst;

 public:
  STOBasis() {}
  void init(int num, int* orbN, int orbL, double* Xsi)
    {
      STOfuncs     = new STOBasisFuncs<Param>[num];
      for (int i=0; i<num; i++)
        STOfuncs[i].initConsts(orbN[i], orbL, Xsi[i]);
    }

  virtual double operator ()(Param& coordinate, int num, int
     currentNumber)
    { return STOfuncs[num](coordinate, currentNumber); }

};

#endif
```

A.1.20 STOBasisFuncs.h

```
#ifndef STOBasisFuncs_IS_INCLUDED
#define STOBasisFuncs_IS_INCLUDED

// ************************************************************
// *                       STOBASISFUNCS                      *
// ************************************************************
template <class Param>
class STOBasisFuncs {

 protected:
```

```
  double constant, expConst, rExp, r;

 public:
  STOBasisFuncs() {}
  virtual void initConsts(int n, int l, double xsi)
    {
      rExp = n-l-1;
      expConst = -xsi;
      double faculty = 1;
      for (int i=1; i<2*n+1; i++) faculty*=i;
      constant = pow(2*xsi, n+0.5)/sqrt(faculty);
    }
  virtual double operator ()(Param& coordinate, int currentNumber)
    {
      r = coordinate.r(currentNumber);
      return constant*pow(r,rExp)*exp(expConst*r);
    }

};

#endif
```

A.1.21 SolidHarmonics.h

```
#ifndef SolidHarmonics_IS_INCLUDED
#define SolidHarmonics_IS_INCLUDED

#include "../Domain/Domain.h"
#include <iostream>
#include <iomanip>
#include "../SolidHarmonicsFuncs/SolidHarmonicsFuncs.h"

#ifndef sqr_IS_INCLUDED
#define sqr_IS_INCLUDED
inline double sqr(double x) {return x*x;};
#endif

// ************************************************************
// *                        SOLIDHARMONICS                    *
// ************************************************************
template <class Param>
class SolidHarmonics {

 protected:
  SolidHarmonicsFuncs<Param>** SHfuncs;

 public:
  SolidHarmonics() {}
  void init(int num, int orbL, int orbM)
    {
      SHfuncs     = new SolidHarmonicsFuncs<Param>*[num];
      for (int i=0; i<num; i++) {
        if ( (orbL == 0)&(orbM == 0) )
          SHfuncs[i] = new SH00<Param>;
        else if ( (orbL == 1)&(orbM == -1) )
          SHfuncs[i] = new SH1m1<Param>;
        else if ( (orbL == 1)&(orbM == 0) )
          SHfuncs[i] = new SH10<Param>;
        else if ( (orbL == 1)&(orbM == 1) )
          SHfuncs[i] = new SH1p1<Param>;
        else if ( (orbL == 2)&(orbM == -2) )
          SHfuncs[i] = new SH2m2<Param>;
        else if ( (orbL == 2)&(orbM == -1) )
          SHfuncs[i] = new SH2m1<Param>;
        else if ( (orbL == 2)&(orbM == 0) )
          SHfuncs[i] = new SH20<Param>;
        else if ( (orbL == 2)&(orbM == 1) )
```

```
          SHfuncs[i] = new SH2p1<Param>;
        else if ( (orbL == 2)&(orbM == 2) )
          SHfuncs[i] = new SH2p2<Param>;
      }
    }
  virtual double operator ()(Param& coordinate, int num, int
     currentNumber)
    { return SHfuncs[num]->SH(coordinate, currentNumber); }

};

#endif
```

A.1.22 SolidHarmonicsFuncs.h

```
#ifndef SolidHarmonicsFuncs_IS_INCLUDED
#define SolidHarmonicsFuncs_IS_INCLUDED

#include "../Domain/Domain.h"
#include <iostream>
#include <iomanip>

#ifndef sqr_IS_INCLUDED
#define sqr_IS_INCLUDED
inline double sqr(double x) {return x*x;};
#endif


// ************************************************************
// *                        SOLIDHARMONICSFUNCS                *
// ************************************************************
template <class Param>
class SolidHarmonicsFuncs {
 protected:
  double constant;
 public:
  SolidHarmonicsFuncs() {}
  virtual double SH(Param& coordinate, int currentNumber) {return 0;}
};

// ***************************** SH00 ***************************
template <class Param>
class SH00 : public SolidHarmonicsFuncs<Param> {
 public:
  SH00() {}
  virtual double SH(Param& coordinate, int currentNumber) {return 1;}
```

```
};

// ************************** SH1m1 **************************
template <class Param>
class SH1m1 : public SolidHarmonicsFuncs<Param> {
 public:
  SH1m1() {}
  virtual double SH(Param& coordinate, int currentNumber)
    {return coordinate(0, currentNumber);}
};

// ************************** SH10 **************************
template <class Param>
class SH10 : public SolidHarmonicsFuncs<Param> {
 public:
  SH10() {}
  virtual double SH(Param& coordinate, int currentNumber)
    {return coordinate(2, currentNumber);}
};

// ************************** SH1p1 **************************
template <class Param>
class SH1p1 : public SolidHarmonicsFuncs<Param> {
 public:
  SH1p1() {}
  virtual double SH(Param& coordinate, int currentNumber)
    {return coordinate(1, currentNumber);}
};

// ************************** SH2m2 **************************
template <class Param>
class SH2m2 : public SolidHarmonicsFuncs<Param> {
 public:
  SH2m2() {constant = 1/2*sqrt(3.);}
  virtual double SH(Param& coordinate, int currentNumber)
    {return constant*(sqr(coordinate(0, currentNumber)) - sqr(
       coordinate(1, currentNumber)));}
};

// ************************** SH2m1 **************************
template <class Param>
class SH2m1 : public SolidHarmonicsFuncs<Param> {
 public:
  SH2m1() {constant = sqrt(3.);}
  virtual double SH(Param& coordinate, int currentNumber)
```

```
    {return constant*coordinate(0, currentNumber)*coordinate(2,
       currentNumber);}
};

// ***************************** SH20 ***************************
template<class Param>
class SH20 : public SolidHarmonicsFuncs<Param> {
 public:
  SH20() {}
  virtual double SH(Param& coordinate, int currentNumber)
    {return (3*sqr(coordinate(2, currentNumber)) - sqr(coordinate.r(
       currentNumber)))/2;}
};

// ***************************** SH2p1 **************************
template<class Param>
class SH2p1 : public SolidHarmonicsFuncs<Param> {
 public:
  SH2p1() {constant = sqrt(3.);}
  virtual double SH(Param& coordinate, int currentNumber)
    {return constant*coordinate(1, currentNumber)*coordinate(2,
       currentNumber);}
};

// ***************************** SH2p2 **************************
template<class Param>
class SH2p2 : public SolidHarmonicsFuncs<Param> {
 public:
  SH2p2() {constant = sqrt(3.);}
  virtual double SH(Param& coordinate, int currentNumber)
    {return  constant*coordinate(0, currentNumber)*coordinate(1,
       currentNumber);}
};

#endif
```

A.1.23 Jastrow.h

```
#ifndef Jastrow_IS_INCLUDED
#define Jastrow_IS_INCLUDED

#include "./../fFunction/fFunction.h"
#include "./../Distance/Distance.h"
#include "../SpinFactors/SpinFactors.h"
#include "../Domain/Domain.h"
#include "../Coor/Coor.h"
#include <blitz/array.h>

#include <iostream>
#include <cmath>
using namespace std;

class Jastrow {

 protected:
  Distance* distance;
  SpinFactors* spinFactors;

  fFunction* f;
  Domain* domain;

  CoorSpinDiff* coors;
  CoorSpinDiff* trialCoor;

  // Array of Upper matrix composed of the values of the function fij
  Array<double, 2> jastrowMatrix;
  // The (new) values
  Array<double, 1> trialColumn, trialRow;
  Array<double, 1> trialDistanceColumn, trialDistanceRow;



  int numParticles;          // Number of particles.
  int Nm1;                   // numParticles - 1.
  int currentParticle;       // Particle proposed moved.
  double jastrowian;         // Value of J, G=J or G=exp(J).
  double difference;         // Value of deltaJ = Jnew - Jold.

  int* spinFlip;             // Boolean 0=noflip, 1=flip.
  int otherParticle;         // Particle to exchange spin with.
  double* newFactor;         // Values of either 1 or 1/2 to impose
```

```
                          // different cusp conditions.
  double* oldFactor;      // Same as above.
  double* otherDifference; // newFactor - oldFactor.

  void suggestMoveNoFlip();
  void calculateNoFlipDifference();
  void suggestMoveFlip();
  void calculateFlipDifference();

 public:
  Jastrow() {}
  void attach(Domain* _domain, fFunction* _f);
  void initialize();
  void setToNextParticle();
  void setCurrentParticle(int _currentParticle);
  void setOtherParticle(int _otherParticle) {otherParticle=
     _otherParticle;}
  void suggestMove();
  void acceptMove();
  void rejectMove() {}
  double& operator()();
  double getDifference();
  Array<double, 2> getJastrowMatrix() {return jastrowMatrix;}
  Array<double, 1> getTrialColumn()   {return trialColumn;}
  Array<double, 1> getTrialRow()      {return trialRow;}
};


class JastrowDiff{
  // The 2*d array, fij(xi_d+h) and fij(xi_d-h)
  DistanceDiff* distanceDiff;
  SpinFactors* spinFactors;
  fFunction* f;
  Domain* domain;
  CoorSpinDiff* coors;
  CoorSpinDiff* trialCoor;

  Array<double, 2> jastrowMatrixPlus, jastrowMatrixMinus;
  // The (new) values
  Array<double, 1> trialColumnPlus, trialRowPlus;
  Array<double, 1> trialColumnMinus, trialRowMinus;
  Array<double, 1> trialDistanceColumn,  trialDistanceRow;

  int numParticles;     // Number of particles.
  int Nm1;              // numParticles - 1.
```

```
  int numDimensions;    // Number of dimensions.
  int currentParticle; // Particle proposed moved.

  int diffPlus;         // Index indicating with respect to which
                        // Cartesian coordinate we wish to
                        // perform positive differentiation
                        // (x(d=0), y(d=1), etc.) fij(xi_d+h).
                        // diffPlus = d.
  int diffMinus;        // Index indicating with respect to which
                        // Cartesian coordinate we wish to
                        // perform nagative differentiation
                        // (x(d=0), y(d=1), etc.) fij(xi_d-h).
                        // diffPlus = d + numDimensions.

 public:
  JastrowDiff() {}
  void attach(Domain* _domain, fFunction* _f, int _differentiate);
  void initialize();
  void setToNextParticle();
  void setCurrentParticle(int _currentParticle);
  void suggestMove();
  void acceptMove();
  void rejectMove() {}
  Array<double, 2> getJastrowMatrixPlus()  {return jastrowMatrixPlus;}
  Array<double, 2> getJastrowMatrixMinus() {return jastrowMatrixMinus
     ;}
};

#include "InlineJastrow.h"

#endif
```

A.1.24 Jastrow.cpp

```
#include "Jastrow.h"

// *************************************************************
// *                          JASTROW                          *
// *************************************************************
//
// ************************** attach ***************************
void Jastrow::attach(Domain* _domain, fFunction* _f) {
  domain           = _domain;
  spinFactors      = domain->getSpinFactors();
  distance         = domain->getDistance();
  f                = _f;
```

```
  numParticles      = domain->getNumParticles();
  Nm1               = numParticles - 1;

  jastrowMatrix.resize(numParticles, numParticles);
  trialColumn.resize(numParticles);
  trialRow.resize(numParticles);

  trialDistanceColumn.resize(numParticles);
  trialDistanceRow.resize(numParticles);

  spinFlip          = domain->getSpinFlip();
  newFactor         = spinFactors->getNewFactor();
  oldFactor         = spinFactors->getOldFactor();
  otherDifference   = spinFactors->getOtherDifference();

  coors             = domain->getCoors();
  trialCoor         = domain->getTrialCoor();
}


// ************************ initialize ***************************
void Jastrow::initialize() {

  Array<double, 2> interElectronicDistances
    = distance->getInterElectronicDistances();
  for(int i=0; i<numParticles; i++)
    for (int j=0; j<numParticles; j++)
      jastrowMatrix(i,j) = (*f)( interElectronicDistances(i,j),
                                 coors[i].r(), coors[j].r());

  spinFactors->calculateMatrix();
  jastrowian = 0;
  int k=0;
  for(int i=0; i<numParticles-1; i++) {
    for (int j=i+1; j<numParticles; j++)
      jastrowian += jastrowMatrix(i,j) * spinFactors->getMatrix()[k
          ++];
  }
  setCurrentParticle(0);
  difference = 0;
}


// ********************** setToNextParticle **********************
void Jastrow::setToNextParticle() {
```

```
  currentParticle++;
  if (currentParticle==numParticles) currentParticle=0;
  difference = 0;
}


// ******************** setCurrentParticle ***********************
void Jastrow::setCurrentParticle(int _currentParticle) {
  currentParticle = _currentParticle;
  difference = 0;
}


// ************************ suggestMove ****************************
void Jastrow::suggestMove() {
  if (*spinFlip) suggestMoveFlip();
  else suggestMoveNoFlip();
}


// ********************** suggestMoveNoFlip ************************
void Jastrow::suggestMoveNoFlip() {
  trialDistanceColumn = distance->getTrialColumn();
  trialDistanceRow    = distance->getTrialRow();
  for ( int i=0; i < currentParticle; i++ ) {
    trialColumn(i) = trialRow(i) =
      (*f)( trialDistanceColumn(i), coors[i].r(), trialCoor->r());
  }
  for ( int i=currentParticle+1; i < numParticles; i++ ) {
    trialColumn(i) = trialRow(i) =
      (*f)( trialDistanceColumn(i), trialCoor->r(), coors[i].r());
  }
  trialColumn(currentParticle) = trialRow(currentParticle) = 0;
  calculateNoFlipDifference();
}


// ****************** calculateNoFlipDifference ******************
void Jastrow::calculateNoFlipDifference() {
  difference = 0;
  for (int i = 0; i<currentParticle; i++)
    difference +=
      newFactor[i] * ( trialColumn(i) - jastrowMatrix(i,
         currentParticle) );
  for (int i = currentParticle+1; i < numParticles; i++ )
```

```
    difference +=
      newFactor[i-1] * ( trialColumn(i) - jastrowMatrix(i,
         currentParticle) );
}


// ********************** suggestMoveFlip ************************
void Jastrow::suggestMoveFlip() {
  trialDistanceColumn = distance->getTrialColumn();
  trialDistanceRow    = distance->getTrialRow();
  otherParticle = domain->getOtherParticle();
  for ( int i=0; i < currentParticle; i++ ) {
    trialColumn(i) = trialRow(i) =
      (*f)(trialDistanceColumn(i), coors[i].r(), trialCoor->r());
  }
  for ( int i=currentParticle+1; i < numParticles; i++ ) {
    trialColumn(i) = trialRow(i) =
      (*f)( trialDistanceColumn(i), trialCoor->r(), coors[i].r());
  }
  trialColumn(currentParticle) = trialRow(currentParticle) = 0;
  calculateFlipDifference();
}


// ******************* calculateFlipDifference *******************
void Jastrow::calculateFlipDifference() {
  difference = 0;
  for (int i = 0; i<currentParticle; i++)
    difference +=
      newFactor[i] * trialColumn(i)
      - oldFactor[i]*jastrowMatrix(i, currentParticle)
      + otherDifference[i] * jastrowMatrix(i, otherParticle);
  for (int i = currentParticle+1; i < numParticles; i++)
    difference +=
      newFactor[i-1] * trialColumn(i)
      - oldFactor[i] * jastrowMatrix(i, currentParticle)
      + otherDifference[i] * jastrowMatrix(i, otherParticle);
}


// ************************ acceptMove ***************************
void Jastrow::acceptMove() {
  Range N(0,Nm1);
  jastrowMatrix( currentParticle, N )  = trialColumn( N );
  jastrowMatrix( N , currentParticle ) = trialRow( N );
```

```
  jastrowian+=difference;
}


// ************************ operator ****************************
double& Jastrow::operator()() {
  return jastrowian;
}

// ************************ getDifference ************************
double Jastrow::getDifference() {
  return difference;
}



// ************   CLASS JastrowPropose   ************

// ***************************************************************
// *                          JASTROWDIFF                        *
// ***************************************************************
//
// ************************** attach *****************************
void JastrowDiff::attach(Domain* _domain, fFunction* _f,
                         int _differentiate) {
  domain           = _domain;
  int differentiate    = _differentiate;
  distanceDiff     = &(domain->getDistanceDiff()[differentiate]);
  spinFactors      = domain->getSpinFactors();
  f                = _f;
  numParticles     = domain->getNumParticles();
  numDimensions    = domain->getNumDimensions();
  Nm1              = numParticles - 1;

  jastrowMatrixPlus.resize(numParticles, numParticles);
  jastrowMatrixMinus.resize(numParticles, numParticles);
  trialColumnPlus.resize(numParticles);
  trialRowPlus.resize(numParticles);
  trialColumnMinus.resize(numParticles);
  trialRowMinus.resize(numParticles);

  trialDistanceColumn.resize(numParticles);
  trialDistanceRow.resize(numParticles);
```

```
  coors             = domain->getCoors();       // HER
  trialCoor         = domain->getTrialCoor(); // HER
  diffPlus          = differentiate;
  diffMinus         = differentiate + numDimensions;
}


// ************************ initialize ***************************
void JastrowDiff::initialize() {
  Array<double, 2> interElectronicDistances(numParticles,numParticles)
     ;
  interElectronicDistances  = distanceDiff->
     getInterElectronicDistances();
  for(int i=0; i<numParticles; i++)
    for (int j=0; j<numParticles; j++)
      jastrowMatrixPlus(i,j)
        = (*f)( interElectronicDistances(i,j),
                coors[i].r(diffPlus), coors[j].r(diffPlus));
  for(int i=0; i<numParticles; i++)
    for (int j=0; j<numParticles; j++) {
      jastrowMatrixMinus(i,j)
        = (*f)( interElectronicDistances(j,i),
                coors[i].r(diffMinus), coors[j].r(diffMinus));
    }
  setCurrentParticle(0);
}


// ********************** setToNextParticle ***********************
void JastrowDiff::setToNextParticle() {
  currentParticle++;
  if (currentParticle==numParticles) currentParticle=0;
}


// ********************** setCurrentParticle ***********************
void JastrowDiff::setCurrentParticle(int _currentParticle) {
  currentParticle = _currentParticle;
}


// ************************ suggestMove ***************************
void JastrowDiff::suggestMove() {
  trialDistanceColumn = distanceDiff->getTrialColumn();
  trialDistanceRow    = distanceDiff->getTrialRow();
```

```
  for ( int i=0; i < currentParticle; i++ ) {
    trialColumnPlus(i) =
      (*f)( trialDistanceColumn(i), coors[i].r(diffPlus)
            , trialCoor->r());
    trialRowPlus(i) =
      (*f)( trialDistanceRow(i), coors[i].r()
            , trialCoor->r(diffPlus));

    trialColumnMinus(i) =
      (*f)( trialDistanceRow(i), coors[i].r(diffMinus)
            , trialCoor->r());
    trialRowMinus(i) =
      (*f)( trialDistanceColumn(i), coors[i].r()
            , trialCoor->r(diffMinus));
  }
  for ( int i=currentParticle+1; i < numParticles; i++ ) {
    trialColumnPlus(i) =
      (*f)( trialDistanceColumn(i), trialCoor->r()
            , coors[i].r(diffPlus) );
    trialRowPlus(i) =
      (*f)( trialDistanceRow(i), trialCoor->r(diffPlus),
            coors[i].r() );

    trialColumnMinus(i) =
      (*f)( trialDistanceRow(i), trialCoor->r(),
            coors[i].r(diffMinus) );
    trialRowMinus(i) =
      (*f)( trialDistanceColumn(i) , trialCoor->r(diffMinus),
            coors[i].r());
  }
  trialColumnPlus(currentParticle)  = trialRowPlus(currentParticle)
      = 0;
  trialColumnMinus(currentParticle) = trialRowMinus(currentParticle)
      = 0;

}


// ************************ acceptMove ***************************
void JastrowDiff::acceptMove() {
  Range N(0,Nm1);
  jastrowMatrixPlus( currentParticle, N )   = trialColumnPlus( N );
  jastrowMatrixPlus( N , currentParticle )  = trialRowPlus( N );

  jastrowMatrixMinus( currentParticle, N )  = trialColumnMinus( N );
```

```
  jastrowMatrixMinus( N , currentParticle ) = trialRowMinus( N );
}
```

A.1.25 Correlation.h

```
#ifndef Correlation_IS_INCLUDED
#define Correlation_IS_INCLUDED

#include "../Domain/Domain.h"
#include "../Jastrow/Jastrow.h"
#include "../Distance/Distance.h"
#include "../fFunction/fFunction.h"
#include "../SpinFactors/SpinFactors.h"
#include <blitz/array.h>

// ************************************************************
// *                          CORREL                          *
// ************************************************************
class Correl {

 protected:

  Domain*        domain;
  int            numParticles;     // Number of particles.
  int            Nm1;              // numParticles - 1.
  int            numDimensions;    // Number of dimension.

  int            currentParticle; // This is the particle that
                                  // currently is (proposed) moved.
  int            otherParticle;   // Particle to exchange spin with.

  Distance*      distance;
  DistanceDiff*  distanceDiff;
  Jastrow*       jastrow;
  JastrowDiff*   jastrowDiff;

  int            expBool;         // 0 G=J, 1 G=exp(J)
  double*        beta;            // Variational parameters
  double         ratio;           // Gnew/Gold
  fFunction*     f;               // J=sum(i<j) f(r_ij, r_i, r_j)
  double         h, hDoubleInverse, hSquaredInverse; // For numerical
                                                     // derivation
  double*        gradJastrow;     // The gradient of G divided by G
  double         laplacian;       // The Laplacian of G divided by G
  double         correlation;     // The value of G

  SpinFactors*   spinFactors;
  double*        spinFactorMatrix;
```

```
  Array<double, 2> jastrowMatrix;
  Array<double, 2> *jastrowMatrixPlus, *jastrowMatrixMinus;


 public:
  Correl();
  void attach(Domain& _domain);
  void attachFFunction(fFunction* _f);
  void createJastrowAndJastrowDiff();

  void setToNextParticle();
  void setCurrentParticle(int _currentParticle);
  void setOtherParticle(int _otherParticle)
    {otherParticle=_otherParticle;}

  // Algorithms for proposing a Metropolis step
  void suggestMove();
  // Thermalization spesific algorithms; prior to samle
  void initializeThermalization();
  void acceptThermalizedMove();
  void rejectThermalizedMove();
  // VMC spesific algorithms
  void initializeVMC();
  void acceptMove();
  void rejectMove();
  // Only applicabel prior to accepting move;
  // returns either Jnew/Jold or exp(Jnew)/exp(Jold)
  void calculateRatio();
  double* getRatioPtr()                {return &ratio;}

  // Only applicable after accepting move;
  // returns either J or exp(J)
  void    calculateGradAndLaplacianRatios();
  double* getGradRatio()               {return gradJastrow;}
  double* getLaplaceRatio()            {return &laplacian;}
  void    calculateCorrelation();
  double* getCorrelationPtr()          {return &correlation;}
  double  operator()();
};


#endif
```

A.1.26 Correlation.cpp

```
#include "Correlation.h"

//*****************************************************************
//*                              CORREL                           *
//*****************************************************************
//
//************************** Correl *******************************
Correl::Correl() {}


//************************** attach *******************************
void Correl::attach(Domain& _domain) {
  domain           = &_domain;
  numParticles     = domain->getNumParticles();
  numDimensions    = domain->getNumDimensions();
  Nm1              = numParticles-1;
  f                = domain->getF();

  spinFactors      = domain->getSpinFactors();
  spinFactorMatrix= spinFactors->getMatrix();

  gradJastrow                    = new double[numParticles*numDimensions
      ];

  distance         = domain->getDistance();
  distanceDiff     = domain->getDistanceDiff();
  h                = domain->getH();
  hDoubleInverse   = 1./2./h;
  hSquaredInverse = 1./h/h;
}


//********************** attachFFunction ************************
void Correl::attachFFunction(fFunction* _f) {
  f = _f;
  expBool=f->getExpBool();
}


//***************** createJastrowAndJastrowDiff *****************
// Create jastrow and jastrowDiff
// NB: This must be preformed AFTER create/
   attachDistanceAndDistanceDiff
//     AND AFTER createFBeta/attachFFunction
void Correl::createJastrowAndJastrowDiff() {
```

```
  // Create jastrow and jastrowDiff
  jastrow            = new Jastrow[1];
  jastrow->attach(domain, f);
  jastrowDiff        = new JastrowDiff[ numDimensions ];
  for (int i=0; i<numDimensions; i++)
      jastrowDiff[i].attach(domain, f, i);
  jastrowMatrix.resize(numParticles, numParticles);
  jastrowMatrixPlus  = new Array<double, 2>[3];
  jastrowMatrixMinus = new Array<double, 2>[3];
  for (int i=0; i<numDimensions; i++) {
    jastrowMatrixPlus[i].resize(numParticles, numParticles);
    jastrowMatrixMinus[i].resize(numParticles, numParticles);
  }
}



//********************* setToNextParticle ***********************
void Correl::setToNextParticle() {
  currentParticle++;
  if (currentParticle==numParticles) currentParticle=0;
  jastrow->setToNextParticle();
  for (int i=0; i<numDimensions; i++)
    jastrowDiff[i].setToNextParticle();
}


//******************** setCurrentParticle ***********************
void  Correl::setCurrentParticle(int _currentParticle){
  currentParticle=_currentParticle;
  jastrow->setCurrentParticle(currentParticle);
  for (int i=0; i<numDimensions; i++)
    jastrowDiff[i].setCurrentParticle(currentParticle);
}


//************************ suggestMove ****************************
void Correl::suggestMove() {
  jastrow->suggestMove();
  calculateRatio();
}


//****************** initializeThermalization *******************
void Correl::initializeThermalization() {
```

```
  jastrow->initialize();
  currentParticle=0;
}


//******************** acceptThermalizedMove **********************
void Correl::acceptThermalizedMove() {
  jastrow->acceptMove();
}


//******************** rejectThermalizedMove **********************
void Correl::rejectThermalizedMove() {
  jastrow->rejectMove();
}


//************************ initializeVMC *************************
void Correl::initializeVMC() {
  for (int i=0; i<numDimensions; i++)
    jastrowDiff[i].initialize();
  spinFactors->calculateMatrix();
  setCurrentParticle(0);
}


//************************** acceptMove **************************
void Correl::acceptMove() {
  for (int i=0; i<numDimensions; i++)
    jastrowDiff[i].suggestMove();
  acceptThermalizedMove();
  for (int i=0; i<numDimensions; i++)
    jastrowDiff[i].acceptMove();
}


//************************** rejectMove **************************
void Correl::rejectMove() {
  rejectThermalizedMove();
  for (int i=0; i<numDimensions; i++)
    jastrowDiff[i].rejectMove();
}


//************** calculateGradAndLaplacianRatios *****************
```

```
void Correl::calculateGradAndLaplacianRatios() {

  laplacian = 0.0;
  int gjIndex = 0;
  double J = 1;
  if (!expBool) J = (*jastrow)();
  for (int i=0; i<numParticles; i++) {
    for (int k=0; k<numDimensions; k++) {
      jastrowMatrixPlus[k]  = jastrowDiff[k].getJastrowMatrixPlus();
      jastrowMatrixMinus[k] = jastrowDiff[k].getJastrowMatrixMinus();

      double gradPlus = 0, gradMinus = 0;
      int t=Nm1;
      int l=i-1;
      for (int j=0; j<i; j++) {
        gradPlus  += spinFactorMatrix[l]*jastrowMatrixPlus[k](j,i);
        gradMinus += spinFactorMatrix[l]*jastrowMatrixMinus[k](j,i);
        l += --t;
      }
      for (int j=i+1; j<numParticles; j++) {
        l++;
        gradPlus  += spinFactorMatrix[l]*jastrowMatrixPlus[k](j,i);
        gradMinus += spinFactorMatrix[l]*jastrowMatrixMinus[k](j,i);
      }
      gradJastrow[gjIndex++] = (gradMinus-gradPlus)*hDoubleInverse/J;
      laplacian += gradPlus + gradMinus;
    }
  }

  jastrowMatrix = jastrow->getJastrowMatrix();

  double grad = 0;
  for (int i=0; i<numParticles; i++) {
    int k=Nm1;
    int l=i-1;
    for (int j=0; j<i; j++) {
      grad  += spinFactorMatrix[l]*jastrowMatrix(j,i);
      l += --k;
    }
    for (int j=i+1; j<numParticles; j++) {
      l++;
      grad  += spinFactorMatrix[l]*jastrowMatrix(j,i);
    }
  }
```

```
  laplacian -= numDimensions*2.*grad;
  laplacian *= hSquaredInverse/J;

  if (expBool)
    for (int i=0; i<numParticles*numDimensions; i++)
        laplacian += sqr( gradJastrow[i] );

}


//********************** calculateRatio ************************
// Only applicabel prior to accepting move
void Correl::calculateRatio() {
  if (expBool) { ratio = exp( (*jastrow).getDifference() );}
  else { ratio =  ((*jastrow)()-(*jastrow).getDifference())/(*jastrow)
     (); }
}


//******************** calculateCorrelation *********************
// To be used to get direct access to the value of the correlation
void Correl::calculateCorrelation() {
  if (expBool) {correlation = exp( (*jastrow)() );}
  else {correlation = (*jastrow)(); }
}


//************************** operator ****************************
// Only applicable after accepting move
double Correl::operator()() {
  if (expBool) {return correlation = exp( (*jastrow)() );}
  else {return correlation =  (*jastrow)(); }
}
```

A.1.27 fFunction.h

```
#ifndef fFunction_IS_INCLUDED
#define fFunction_IS_INCLUDED

#include <string>
#include <iostream>
#include <cstdio>
using namespace std;

// ****************************************************************
// *                          FFUNCTION                           *
// ****************************************************************
class fFunction {
protected:
  int     numberParams;  // Number of parameters
  double* params;        // Array of parameters
  int     expBool;       // If correlation G=J;           expBool=0
                         // Else if correlation G=exp(J); expBool=1

  virtual inline double   f(double rij, double ri, double rj)
    {return 0;}
  virtual inline double   R(double r, int paramNumber)
    {return 0;}
  double R_IJ, R_I, R_J;

public:
  fFunction() {}
  virtual void    attach(int _numberParams, double* _params,
                         int _expBool) {
    numberParams = _numberParams;
    params       = _params;
    expBool      = _expBool;
  }
  virtual int     getExpBool()           {return expBool;}
  virtual double* getParams()            {return params;}
  virtual int     getNumberParams()      {return numberParams;}
  virtual double  operator()(double rij, double ri, double rj)
    {return f(rij, ri, rj);}
};

//************************** fNone ******************************
class fNone : public fFunction {
protected:
```

```
  virtual inline double f(double rij, double ri, double rj) {return
      1.;}

public:
  fNone() {}
};

//*************************** fBeta ******************************
class fBeta: public fFunction {
protected:
  virtual inline double   f(double rij, double ri, double rj)
    { return 0.5*rij/(1+params[0]*rij);}

public:
  fBeta() {}
};

//*************************** fBeta2 *****************************
class fBeta2: public fFunction {
protected:
  virtual inline double   f(double rij, double ri, double rj)
    {return params[1]*rij/(1+params[0]*rij);}
public:
  fBeta2() {}
};

//*************************** fBeta3 *****************************
class fBeta3: public fFunction {
protected:
  virtual inline double   f(double rij, double ri, double rj)
    {return params[1]*rij/(1+params[0]*rij+params[2]*rij*rij);}
public:
  fBeta3() {}
};

//************************** fBeta2r2 ****************************
class fBeta2r2: public fFunction {
protected:
  virtual inline double f(double rij, double ri, double rj)
    {return rij/(1+params[0]*rij+params[1]*rij*rij);}

public:
  fBeta2r2() {}
};
```

```
//************************ fExtended ****************************
class fExtended: public fFunction {

protected:
  virtual inline double R(double r, int paramNumber)
    {
      double pr = params[paramNumber]*r;
      return r/(1+pr);
    }

  virtual inline double f(double rij, double ri, double rj)
    {
      R_IJ = R(rij, 0); R_I = R(ri, 1); R_J = R(rj, 1);
      return
        // e-e:                                        // m n o
        params[2]*R_IJ                                 // 0 0 1
        + params[3]*R_IJ*R_IJ                          // 0 0 2

        // e-n:                                        // m n o
        + params[4]*(R_I+R_J)                          // 1 0 0
        + params[5]*(R_I*R_I+R_J*R_J)                  // 2 0 0

        // e-e-n:                                      // m n o
        + params[6]*R_I*R_I*R_J*R_J                    // 2 2 0
        + params[7]*R_IJ*R_IJ*(R_I*R_I+R_J*R_J)        // 2 0 2
        ;
    }

public:
  fExtended() {}
};

#endif
```

A.1.28 LocalWaveFunction.h

```
#ifndef LocalWaveFunction_IS_INCLUDED
#define LocalWaveFunction_IS_INCLUDED

#include "../Domain/Domain.h"
#include <iostream>
#include <iomanip>

// ****************************************************************
// *                        LOCALWAVEFUNCTION                     *
// ****************************************************************
class LocalWaveFunction {

 protected:

  ofstream *outputFile;

  int      numParticles;      // Number of particles.
  int      numDimensions;     // Number of dimension.
  int      prodPartDim;       // numParticles * numDimensions.

  double* centralSlater;      // Pointer to the value of the central
                              // Slater-determinant.
  double* localSlater;        // Pointer to the value of the local
                              // Slater-determinant.
  double* localDiffSlater;    // Pointer to the gradient of the Slater-
                              // determinent to the Slater-deterimant.
  double* localDDiffSlater; // Pointer to the Laplacian of the Slater-
                              // determinent to the Slater-deterimant.
  int      numAlpha;          // Number of Slater-determinant parameters
     .
  double* alphaParams;        // Pointer to the local Slater-determinant
                              // parameters.

  double* centralCorrelation;     // Pointer to the value of the
                                  // central correlation G.
  double* localCorrelation;       // Pointer to the value of the local
                                  // correlation G.
  double* localDiffCorrelation;   // Pointer to the gradient of the
                                  // correlation to the correlation.
  double* localDDiffCorrelation; // Pointer to the Laplacian of the
                                  // correlation to the correlation.
  int      numBeta;               // Number of correlation parameters.
  double* betaParams;             // Pointer to the local  correlation
```

```
                                // parameters.

double  norm;                   // localSlater*localCorrelation/
                                // (centralSlater*centralCorrelation)
                                   .
double  kineticEnergy;          // The kinetic energy.

int     numSamples;             // Keep track of the number of
   samples.
double  accumulatedEnergy;      // Sum of local energies E_L
double  accumulatedEnergySquared; // Sum of E_L*E_L
double  accumulatedNorm;        // Sum of the norms
double  accumulatedVariance;    // Sum of (E_L - E_ref - deltaE)^2

// Blocks are included to give an estimate of the true standard
// deviation of the energy. This approch is due to auto-correltion
// effects. The results should be tested by for example producing
// several independent VMC runs, and finding their standard
// deviation.
int     numBlocks;              // The actual number of blocks.
int     maxNumBlocks;           // Number of blocks allocated.
int*    blockSize;              // Number of samples per block.
int*    currentBlockNum;        // Index used for performing block
                                // sampling.
int*    numSamplesBlock;        // The number of samples for in each
                                // of the different blocks.
double* accumulatedEnergyBlock;// The sum of the energy samples for
                                // the individual blocks.
double* accumulatedEnergySquaredBlock; // The sum of the square of
   the
                                      // block-energies.
double* accumulatedNormBlock;   // Sum of the norms for each block.
double* tempEnergyBlock;        // Accumulates energies for a sample.
double* tempNormBlock;          // Accumulates norms to a sample.

int     varianceOptimization;   // Boolean determining whether we
                                // optimize with respect to energy(0)
                                // or variance(1).
int     setWeightToUnity;       // Boolean true(1) weights (the norms
   )
                                // are set to unity.
double  referenceEnergy;        // Reference energy, E_ref (used for
                                // variance optimization).
double  deltaE;                 // Small energry displacement (used
                                // for variance optimization).
```

```
  void     calculateNorm();
  void     calculateKineticEnergy();
  void     allocateBlocks();
  double   getAverageEnergyBlock(int block);
  double   getStandardDeviationBlock(int block);
  void     addBlock(int _blockSize);
  void     deleteBlocks();

 public:
  LocalWaveFunction() {}
  void     init();
  void     initialize(Domain* domain);
  void     attachCentralSlaterDet(double* _centralSlaterDet);
  void     attachCentralCorrelation(double* _centralCorrelation);
  void     attachSlater(double* _localSlater,
                        double* _localDiffSlater,
                        double* _localDDiffSlater,
                          double* _alphaParams);
  void     attachCorrelation(double* _localCorrelation,
                             double* _localDiffCorrelation,
                             double* _localDDiffCorrelation,
                             double* _betaParams);
  void     sample(double potentialEnergy);
  void     summary();

  double   getAverageEnergy();
  double   getStandardDeviation();
  void     setReferenceEnergy(double E) {referenceEnergy=E;}
  double   getVariance();

  double* getAlphaParams() {return alphaParams;}
  double* getBetaParams()  {return betaParams;}
};

#endif
```

A.1.29 LocalWaveFunction.cpp

```
#include "LocalWaveFunction.h"

// ****************************************************************
// *                        LOCALWAVEFUNCTION                     *
// ****************************************************************
//
// Keeps track of one local variation
```

```
//
// ************************** init ******************************
void LocalWaveFunction::init() {
  numSamples                = 0;
  accumulatedEnergy         = 0;
  accumulatedEnergySquared  = 0;
  accumulatedNorm           = 0;
  accumulatedVariance       = 0;
}


// ************************ initialize **************************
void LocalWaveFunction::initialize(Domain* domain) {
  numParticles              = domain->getNumParticles();
  numDimensions             = domain->getNumDimensions();
  prodPartDim               = numParticles*numDimensions;
  numAlpha                  = domain->getNumAlpha();
  numBeta                   = domain->getNumBeta();
  outputFile                = domain->getOutputFile();

  setWeightToUnity          = domain->getSetWeightToUnity();
  varianceOptimization      = domain->getVarianceOptimization();
  referenceEnergy           = domain->getReferenceEnergy();
  deltaE                    = domain->getDeltaE();

  init();
  deleteBlocks();
  maxNumBlocks              = 15;
  allocateBlocks();

  addBlock(2);
  addBlock(5);
  addBlock(10);
  addBlock(20);
  addBlock(25);

  addBlock(50);
  addBlock(100);
  addBlock(200);
  addBlock(500);
  addBlock(1000);

  addBlock(2000);
  addBlock(5000);
  addBlock(10000);
```

```
  addBlock(20000);
  addBlock(50000);
}


// ********************** allocateBlocks ************************
void LocalWaveFunction::allocateBlocks() {
  blockSize                      = new int[maxNumBlocks];
  currentBlockNum                = new int[maxNumBlocks];
  numSamplesBlock                = new int[maxNumBlocks];
  accumulatedEnergyBlock         = new double[maxNumBlocks];

  accumulatedEnergySquaredBlock = new double[maxNumBlocks];
  accumulatedNormBlock           = new double[maxNumBlocks];
  tempEnergyBlock                = new double[maxNumBlocks];
  tempNormBlock                  = new double[maxNumBlocks];
}


// ******************** attachCentralSlaterDet *******************
// Central == the origin of the local variation
// centralSlaterDet = pointer to the value of the central SD
void LocalWaveFunction::attachCentralSlaterDet(double* _centralSlater)
    {
  centralSlater    = _centralSlater;
}


// ***************** attachCentralCorrelation *******************
// Central == the origin of the local variation
// centralCorrelation = pointer to the value of the central
// correlation
void LocalWaveFunction::attachCentralCorrelation(double*
   _centralCorrelation) {
  centralCorrelation = _centralCorrelation;
}


// ********************** attachSlater ************************
// These are the values of the (varied) SD
void LocalWaveFunction::attachSlater(double* _localSlater,
                                     double* _localDiffSlater,
                                     double* _localDDiffSlater,
                                     double* _alphaParams) {
  localSlater       = _localSlater;
```

```
  localDiffSlater  = _localDiffSlater;
  localDDiffSlater = _localDDiffSlater;
  alphaParams      = _alphaParams;
}


// ******************** attachCorrelation ************************
void LocalWaveFunction::attachCorrelation(double* _localCorrelation,
                                          double*
                                             _localDiffCorrelation,
                                          double*
                                             _localDDiffCorrelation,
                                          double* _betaParams) {
  localCorrelation      = _localCorrelation;
  localDiffCorrelation  = _localDiffCorrelation;
  localDDiffCorrelation = _localDDiffCorrelation;
  betaParams            = _betaParams;
}


// ************************ calculateNorm ************************
void LocalWaveFunction::calculateNorm() {
  if (setWeightToUnity) norm=1;
  else {
    norm =  (*localSlater)*(*localCorrelation)
      /(*centralSlater)/(*centralCorrelation);
    norm *= norm;
  }
}


// ******************** calculateKineticEnergy *******************
void LocalWaveFunction::calculateKineticEnergy() {
  kineticEnergy = -0.5*( (*localDDiffSlater) + (*localDDiffCorrelation
     ) )
    - dotProduct(localDiffSlater, localDiffCorrelation, prodPartDim);
}


// ************************** sample *****************************
void  LocalWaveFunction::sample(double potentialEnergy ) {
  calculateNorm();
  calculateKineticEnergy();
  numSamples                += 1;
  double localEnergy       = kineticEnergy + potentialEnergy;
```

```
  double localEnergyNorm   = localEnergy*norm;
  double localEnergy2norm = localEnergyNorm*localEnergy;

  accumulatedEnergy           += localEnergyNorm;
  accumulatedEnergySquared  += localEnergy2norm;
  accumulatedNorm             += norm;
  accumulatedVariance
    += pow(localEnergy-referenceEnergy-deltaE, 2)*norm;
  for (int i=0; i<numBlocks; i++) {
    tempEnergyBlock[i]          += localEnergy;
    tempNormBlock[i]            += norm;
    if (++currentBlockNum[i]==blockSize[i]) {
      currentBlockNum[i]=0;
      numSamplesBlock[i]+=1;
      double averageNorm    =tempNormBlock[i]/blockSize[i];
      accumulatedNormBlock[i] +=averageNorm;
      double averageEnergy = tempEnergyBlock[i]/blockSize[i];
      accumulatedEnergyBlock[i] +=averageEnergy*averageNorm;
      accumulatedEnergySquaredBlock[i]
        +=averageEnergy*averageEnergy*averageNorm;
      tempEnergyBlock[i]=tempNormBlock[i]=0;
    }
  }
}


// ********************* getAverageEnergy ************************
double LocalWaveFunction::getAverageEnergy() {
  return accumulatedEnergy/accumulatedNorm;
}


// ******************** getStandardDeviation *********************
double LocalWaveFunction::getStandardDeviation() {
  return sqrt ( (accumulatedEnergySquared/accumulatedNorm
                 - pow( accumulatedEnergy/accumulatedNorm, 2) ) /
                    numSamples);
}


// ************************ getVariance ************************
double LocalWaveFunction::getVariance() {
  return sqrt ( (accumulatedVariance/accumulatedNorm)
                 / numSamples);
}
```

```
// ******************** getAverageEnergyBlock ********************
double LocalWaveFunction::getAverageEnergyBlock(int block) {
  return accumulatedEnergyBlock[block]/accumulatedNormBlock[block];
}


// ******************* getStandardDeviationBlock ******************
double LocalWaveFunction::getStandardDeviationBlock(int block) {
  return sqrt ( (accumulatedEnergySquaredBlock[block]
                 /accumulatedNormBlock[block]
                - pow( accumulatedEnergyBlock[block]
                       /accumulatedNormBlock[block], 2) )
                / numSamplesBlock[block]);
}


// ************************* addBlock ****************************
void LocalWaveFunction::addBlock(int _blockSize) {
  if (numBlocks<maxNumBlocks) {
    blockSize[numBlocks]                    = _blockSize;
    currentBlockNum[numBlocks]              = 0;
    accumulatedEnergyBlock[numBlocks]       = 0;
    accumulatedEnergySquaredBlock[numBlocks] = 0;
    accumulatedNormBlock[numBlocks]         = 0;
    tempEnergyBlock[numBlocks]              = 0;
    tempNormBlock[numBlocks]                = 0;
    numSamplesBlock[numBlocks]              = 0;
    numBlocks++;
  }
  else
    *outputFile << "Could not create an additional block in
       LocalWaveFunction.\n"
                << "Maximun number of blocks set to " << maxNumBlocks
                << " (in code).\n";
}


// ********************** deleteBlocks ***************************
void LocalWaveFunction::deleteBlocks() {
  numBlocks=0;
}
```

```
// ************************** summary ***************************
void LocalWaveFunction::summary() {
  *outputFile << "*** Alpha = [ ";
  for (int i=0; i<numAlpha; i++)
    *outputFile << setw(5) << alphaParams[i] << " ";
  *outputFile << "]  Beta = [ ";
  for (int i=0; i<numBeta; i++)
    *outputFile << setw(5) << betaParams[i] << " ";
  *outputFile << "]" <<  " Energy = " << setw(12) << getAverageEnergy
     ()
              << " " << setw(12) << getStandardDeviation();
  if (varianceOptimization) *outputFile << " Variance = " << setw(12)
                                        << getVariance();
  *outputFile << " ***"<<endl;
  *outputFile << "Blocksize ";
  for (int i=0; i<numBlocks; i++)
    *outputFile << setw(9) << blockSize[i] << " ";
  *outputFile << "\n";
  *outputFile << "S.d.         [ ";
  for (int i=0; i<numBlocks; i++)
    *outputFile << setw(9) << getStandardDeviationBlock(i) << " ";
  *outputFile << "]\n";
}
```

A.1.30 Variations.h

```
#ifndef Variations_IS_INCLUDED
#define Variations_IS_INCLUDED

#include "../Domain/Domain.h"
#include "../Correlation/Correlation.h"
#include "../SlaterDet/SlaterDet.h"
#include "../Ref/Ref.h"
#include "../fFunction/fFunction.h"
#include "../LocalWaveFunction/LocalWaveFunction.h"
#include "../SpinFactors/SpinFactors.h"

// ************************************************************
// *                          VARIATIONS                      *
// ************************************************************
template <class SlaterDeterminant>
class Variations {

 protected:

  ofstream* outputFile;
  fFunction*  f;

  int       numSlaterVariations;   // Number of variations in the
                                   // Slater-determinant parameter
                                   // subspace.
  int       numCorrelVariations;   // Number of variations in the
                                   // correlation parameter subspace.
  int       numVariations;         // Total number of variations.
  int       numParticles;          // Number of particles.

  int       centralSlater;         // Index specifying which of the
                                   // Slater-variations we vary about.
  int       centralCorrel;         // Index specifying which of the
                                   // correlation-variations we vary
                                   // about.

  int       allowSpinFlip;         // Boolean indicating whether we
                                   // allow(1) spin-flip or not(0).
  int       varianceOptimization;  // Boolean indicating whether we
                                   // optimize the paramters with
                                   // respect to energy or variance
                                   // minimization.
```

```
  Ref<Domain>              domain;
  Ref<Correl>              correlation;
  Ref<SlaterDeterminant> slaterDeterminant;

  LocalWaveFunction*       localWaveFunctions;
  Correl*                  correl;
  SlaterDeterminant*       slater;

  void                     createCorrel();
  void                     createSlaterDet();
  void                     createLocalSurface();

public:
  Variations(Domain& _domain);
  void    setToNextParticle();
  void    setCurrentParticle(int _currentParticle);

  void    initializeThermalization(); // Called prior to starting the
                                      // thermalization procedure.
  void    initializeVMC();            // Called in-between
     thermalization
                                      // and the actual VMC.
  void    initNewVmcRun();            // Called when subsequent VMC
     runs
                                      // are performed (after one VMC
                                         run
                                      // is finished and before next
                                      // thermalization).

  void    suggestMove();
  void    acceptThermalization();
  void    rejectThermalization();
  void    acceptMove();
  void    rejectMove();

  void    sample();                   // One sample of the energy.
  void    summary();                  // Prints all the different local
                                      // variations to outputFile.
  void    summaryLowest();            // Prints all the lowest local
                                      // variation to outputFile (with
                                      // lowest we mean either lowest
                                      // energy or variance).

  SlaterDeterminant& getSlaterDet()        {return slaterDeterminant()
     ;}
```

```
  Correl&            getCorrelation()      {return correlation();}

  int     findLowestWaveFunction();   // Finds the index corresponding
                                      // to the lowest local variation.
  double getLowestEnergy();           // Find the energy corresponding
                                      // to the lowest local variation.
  double getLowestVariance();         // Find the variance
     corresponding
                                      // to the lowest local variation.
  double *getLowestAlphaParams();     // Find the Slater-paramters
                                      // corresponding to the lowest
                                      // local variation.
  double *getLowestBetaParams();      // Find the correlation-paramters
                                      // corresponding to the lowest
                                      // local variation.
  void setReferenceEnergy(double E); // Changes reference energy to E.

};

#include "Variations.cpp"


#endif
```

A.1.31 Variations.cpp

```
#ifndef VariationCPP_IS_INCLUDED
#define VariationCPP_IS_INCLUDED

#include "Variations.h"



// ****************************************************************
// *                          VARIATIONS                          *
// ****************************************************************
//
// ************************* Vatiations ***************************
template <class SlaterDeterminant>
Variations<SlaterDeterminant>::Variations(Domain& _domain) : domain(
   _domain) {
  numParticles          = domain().getNumParticles();
  numSlaterVariations   = domain().getNumSlaterVariations();
  numCorrelVariations   = domain().getNumCorrelVariations();
  allowSpinFlip         = domain().getAllowSpinFlip();
  outputFile            = domain().getOutputFile();
```

```
  varianceOptimization = domain().getVarianceOptimization();

  createSlaterDet();
  createCorrel();
  createLocalSurface();
}


// ********************** createSlaterDet ************************
template <class SlaterDeterminant>
void Variations<SlaterDeterminant>::createSlaterDet() {
  slater = new SlaterDeterminant[numSlaterVariations];
  for (int i=0; i< numSlaterVariations; i++)
    slater[i].init(&domain(), i);
  centralSlater = domain().getCentralSlater();
  slaterDeterminant = &(slater[centralSlater]);
}


// ************************* createCorrel ************************
template <class SlaterDeterminant>
void Variations<SlaterDeterminant>::createCorrel() {
  correl = new Correl[numCorrelVariations];
  for (int i=0; i<numCorrelVariations; i++) {
    correl[i].attach(domain());
    correl[i].attachFFunction(&(domain().getF()[i]));
    correl[i].createJastrowAndJastrowDiff();
  }
  centralCorrel = domain().getCentralCorrel();
  correlation = &(correl[centralCorrel]);
}


// ********************** setToNextParticle ************************
template <class SlaterDeterminant>
void Variations<SlaterDeterminant>::setToNextParticle() {
  for (int i=0; i<numCorrelVariations; i++)
    correl[i].setToNextParticle();
  for (int i=0; i< numSlaterVariations; i++)
    slater[i].setToNextParticle();
}


// ********************** setCurrentParticle ************************
template <class SlaterDeterminant>
```

```
void Variations<SlaterDeterminant>::setCurrentParticle(int
    _currentParticle) {
  for (int i=0; i<numCorrelVariations; i++)
    correl[i].setCurrentParticle(_currentParticle);
  for (int i=0; i< numSlaterVariations; i++)
    slater[i].setCurrentParticle(_currentParticle);
}


// ****************** initializeThermalization ******************
// Called prior to starting the thermalization procedure.
template <class SlaterDeterminant>
void Variations<SlaterDeterminant>::initializeThermalization() {
  correlation().initializeThermalization();
  setCurrentParticle(0);
}


// ********************** initializeVMC ************************
// Called inbetween VMC and thermalization
template <class SlaterDeterminant>
void Variations<SlaterDeterminant>::initializeVMC() {
  for (int i=0; i<numCorrelVariations; i++) {
    correl[i].initializeThermalization();
    correl[i].initializeVMC();
  }
  for (int i=0; i< numSlaterVariations; i++)
    slater[i].initDiff();

  for (int i=0; i<numVariations; i++)
    localWaveFunctions[i].init();
  setCurrentParticle(0);
}


// ************************ initNewVmcRun ************************
template <class SlaterDeterminant>
void Variations<SlaterDeterminant>::initNewVmcRun() {
  for ( int i=0; i<numSlaterVariations; i++ ) {
    slater[i].initNewVmcRun();
  }
}


// ********************** createLocalSurface **********************
```

```
template <class SlaterDeterminant>
void Variations<SlaterDeterminant>::createLocalSurface() {
  numVariations        = numSlaterVariations*numCorrelVariations;
  double* _centralSlaterDet    = &slaterDeterminant().getDet();
  double* _centralCorrelation = correlation().getCorrelationPtr();
  localWaveFunctions = new LocalWaveFunction[numVariations];
  for ( int i=0; i<numVariations; i++ ) {
    localWaveFunctions[i].initialize( &domain() );
    localWaveFunctions[i].attachCentralSlaterDet(_centralSlaterDet);
    localWaveFunctions[i].attachCentralCorrelation(_centralCorrelation
        );
  }

  int k=0;
  for ( int i=0; i<numSlaterVariations; i++ ) {
    for ( int j=0; j<numCorrelVariations; j++ ) {
      localWaveFunctions[k]
        .attachSlater(&slater[i].getDet(),
                      slater[i].getDiffRatiosPtr(),
                      &slater[i].getDDiffRatio(),
                      domain().getAlphaParam(i));
      localWaveFunctions[k]
        .attachCorrelation(correl[j].getCorrelationPtr(),
                           correl[j].getGradRatio(),
                           correl[j].getLaplaceRatio(),
                           domain().getBetaParam(j));
      k++;
    }
  }
}


// ************************ suggestMove *************************
template <class SlaterDeterminant>
void Variations<SlaterDeterminant>::suggestMove() {
  correlation().suggestMove();
  if (allowSpinFlip) slaterDeterminant().suggestMoveFlip();
  else slaterDeterminant().suggestMove();
}


// ******************** acceptThermalization ********************
template <class SlaterDeterminant>
void Variations<SlaterDeterminant>::acceptThermalization() {
  correlation().acceptThermalizedMove();
```

```
  if (allowSpinFlip) {
    for (int i=0; i<numSlaterVariations; i++)
      if (i!=centralSlater)
        slater[i].suggestMoveFlip();
    for (int i=0; i<numSlaterVariations; i++)
      slater[i].acceptMoveFlip();
  }
  else {
    for (int i=0; i< numSlaterVariations; i++)
      if (i!=centralSlater)
        slater[i].suggestMove();
    for (int i=0; i< numSlaterVariations; i++)
      slater[i].acceptMove();
  }
}


// ********************* rejectThermalization *********************
template <class SlaterDeterminant>
void Variations<SlaterDeterminant>::rejectThermalization() {
  correlation().rejectThermalizedMove();
  if (allowSpinFlip) {
    for (int i=0; i<numSlaterVariations; i++)
      if (i!=centralSlater)
        slater[i].suggestMoveFlip();
    for (int i=0; i< numSlaterVariations; i++)
      slater[i].rejectMoveFlip();
  }
  else {
    for (int i=0; i< numSlaterVariations; i++)
      if (i!=centralSlater)
        slater[i].suggestMove();
    for (int i=0; i< numSlaterVariations; i++)
      slater[i].rejectMove();
  }
}


// ************************* acceptMove *************************
template <class SlaterDeterminant>
void Variations<SlaterDeterminant>::acceptMove() {
  for (int i=0; i<numCorrelVariations; i++)
    if (i!=centralCorrel)
      correl[i].suggestMove();
  for (int i=0; i<numCorrelVariations; i++) {
```

```
    correl[i].acceptMove();
  }
  if (allowSpinFlip) {
    for (int i=0; i<numSlaterVariations; i++)
      if (i!=centralSlater)
        slater[i].suggestMoveFlip();
    for (int i=0; i<numSlaterVariations; i++)
      slater[i].acceptMoveFlip();
  }
  else {
    for (int i=0; i< numSlaterVariations; i++)
      if (i!=centralSlater)
        slater[i].suggestMove();
    for (int i=0; i< numSlaterVariations; i++)
      slater[i].acceptMove();
  }
}


// ************************* rejectMove *************************
template <class SlaterDeterminant>
void Variations<SlaterDeterminant>::rejectMove() {

  for (int i=0; i<numCorrelVariations; i++) {
    correl[i].rejectMove();
  }
  if (allowSpinFlip) {
    for (int i=0; i<numSlaterVariations; i++)
      if (i!=centralSlater)
        slater[i].suggestMoveFlip();
    for (int i=0; i< numSlaterVariations; i++)
      slater[i].rejectMoveFlip();
  }
  else {
    for (int i=0; i< numSlaterVariations; i++)
      if (i!=centralSlater)
        slater[i].suggestMove();
    for (int i=0; i< numSlaterVariations; i++)
      slater[i].rejectMove();
  }
}


// ************************** sample ****************************
template <class SlaterDeterminant>
```

```
void Variations<SlaterDeterminant>::sample() {
  for (int i=0; i<numCorrelVariations; i++) {
    correl[i].calculateCorrelation();
    correl[i].calculateGradAndLaplacianRatios();
  }

  for (int i=0; i< numSlaterVariations; i++ ) {
    slater[i].calcDDiffRatio();
    slater[i].calcDiffRatios();
  }
  double potentialEnergy = domain().getNucleusElectronPotential()
    + domain().getInterElectronicPot();
  for (int i=0; i<numVariations; i++) {
    localWaveFunctions[i].sample(potentialEnergy);
  }
}


// ************************** summary ****************************
template <class SlaterDeterminant>
void Variations<SlaterDeterminant>::summary() {
  for (int i=0; i<numVariations; i++)
    localWaveFunctions[i].summary();
}


// ************************ summaryLowest ************************
template <class SlaterDeterminant>
void Variations<SlaterDeterminant>::summaryLowest() {
  if (varianceOptimization)  *outputFile << "Lowest Variance for: ";
  else  *outputFile << "Lowest Energy for: ";
  localWaveFunctions[findLowestWaveFunction()].summary();
}


// ******************* findLowestWaveFunction ******************
// Returns the integer value of the lovalWaveFunction with the lowest
// energy or variance
template <class SlaterDeterminant>
int Variations<SlaterDeterminant>::findLowestWaveFunction() {
  if (varianceOptimization) {
    int lowest=-1;
    double minVariance = 999;
    for (int i=0; i<numVariations; i++) {
      double variance = localWaveFunctions[i].getVariance();
```

```
      if ( variance < minVariance) {
        minVariance = variance;
        lowest = i;
      }
    }
    if (lowest ==-1) {
      *outputFile << "Error finding variance minimum in class
         Variation"
                  << endl << "Minimum variance >= 999!" << endl;
      return -1;
    }
    else return lowest;
  }
  else {
    int lowest=-1;
    double minEnergy = 0;
    for (int i=0; i<numVariations; i++) {
      double energy = localWaveFunctions[i].getAverageEnergy();
      if ( energy < minEnergy) {
        minEnergy = energy;
        lowest = i;
      }
    }
    if (lowest ==-1) {
      *outputFile << "Error finding energy minimum in class Variation"
                  << endl << "Minimum energy >= 0!" << endl;
      return -1;
    }
    else return lowest;
  }
}


// ********************** getLowestEnergy ************************
template <class SlaterDeterminant>
double Variations<SlaterDeterminant>::getLowestEnergy() {
  return localWaveFunctions[findLowestWaveFunction()].getAverageEnergy
     ();
}


// ********************** getLowestVariance ************************
template <class SlaterDeterminant>
double Variations<SlaterDeterminant>::getLowestVariance() {
  return localWaveFunctions[findLowestWaveFunction()].getVariance();
```

```
}


// ******************** getLowestAlphaParams ********************
template <class SlaterDeterminant>
double* Variations<SlaterDeterminant>::getLowestAlphaParams() {
  return localWaveFunctions[findLowestWaveFunction()].getAlphaParams()
      ;
}


// ******************** getLowestBetaParams ********************
template <class SlaterDeterminant>
double* Variations<SlaterDeterminant>::getLowestBetaParams() {
  return localWaveFunctions[findLowestWaveFunction()].getBetaParams();
}


// ******************** setReferenceEnergy ********************
template <class SlaterDeterminant>
void Variations<SlaterDeterminant>::setReferenceEnergy(double E) {
  for (int i=0; i<numVariations; i++)
    localWaveFunctions[i].setReferenceEnergy(E);
}

#endif
```

A.1.32 Vmc.h

```
#ifndef Vmc_IS_INCLUDED
#define Vmc_IS_INCLUDED

#include "../Domain/Domain.h"
#include "../Correlation/Correlation.h"
#include "../SlaterDet/SlaterDet.h"
#include "../Ref/Ref.h"
#include "../Variations/Variations.h"
#include "../Walker/Walker.h"

// ************************************************************
// *                          VMC                             *
// ************************************************************
template <class SlaterDeterminant>
class Vmc {
 protected:

  Ref<Domain>                              domain;
  Ref<SlaterDeterminant>                   slaterDeterminant;
  Ref<Correl>                              correlation;
  Ref<Walker<SlaterDeterminant> >          walker;
  Ref<Variations<SlaterDeterminant> >      wf;
  Ref<Random2>                             randomMetro;

  int       numThermalization;  // Number of thermalization steps
  int       numCycles;          // Number of Monte Carlo cycles

  ofstream *outputFile;         // Outputfile
  string    thermalizationType; // Determines way to perform
                                // the thermalization(s).
  string    vmcType;            // Determines way to perform
                                // the VMC run(s).
  int       uniDirectionalMovement; // Boolean determining whether
                                // we to search for a minima until
                                // movement in parameter space is
                                // no longer uni-directional(1) or
                                // not(0).
  int       numberVmcRuns;      // Either the number of VMC runs
                                // performed or the maximum number
                                // of VMC runs we allow when
                                // performing a (uni-directional)
                                // search for minima.
```

```
  int       numberOfUniDirectionalMoves; // Number of uni-directional
                                         // moves.
  int       rank;                        // The rank or prosess number of the
                                         // MPI run.
  double    centerRank;                  // (Number of ranks + 1)/2.

 public:
  Vmc(Domain& _domain);
  void thermalization();
  void adaptiveStepThermalization();
  void vmcSome();
  void vmcOneParticleAtATime();
  void initNextRun();
  void run();

};

#include "Vmc.cpp"

#endif
```

A.1.33 Vmc.cpp

```
#ifndef VmcCPP_IS_INCLUDED
#define VmcCPP_IS_INCLUDED

#include "Vmc.h"



// ************************************************************************
// *                                  VMC                                 *
// ************************************************************************
//
// ***************************** Vmc ******************************
template <class SlaterDeterminant>
Vmc<SlaterDeterminant>::Vmc(Domain& _domain) : domain(_domain) {

  wf                = new Variations<SlaterDeterminant>( domain() );
  slaterDeterminant = &(wf().getSlaterDet());
  correlation       = &(wf().getCorrelation());
  randomMetro       = domain().getRandomMetro();
  walker            = new Walker<SlaterDeterminant>( domain(),
                                                     slaterDeterminant
                                                         (),
                                                     randomMetro(),
```

```
                                                    correlation(),
                                                    wf() );

  outputFile             = domain().getOutputFile();
  numThermalization      = domain().getNumThermalization();
  numCycles              = domain().getNumCycles();
  rank                   = domain().getRank();
  centerRank             = (domain().getSize() - 1.0) / 2.0;

  thermalizationType              = domain().getThermalizationType();
  vmcType                         = domain().getVmcType();
  uniDirectionalMovement          = domain().getUniDirectionalMovement()
      ;
  numberVmcRuns                   = domain().getNumberVmcRuns();
  numberOfUniDirectionalMoves     = domain().
      getNumberOfUniDirectionalMoves();
  *outputFile << endl;
}


// ********************** thermalization ************************
template <class SlaterDeterminant>
void Vmc<SlaterDeterminant>::thermalization() {
  *outputFile << " Thermalization: " << numThermalization;
  wf().initializeThermalization();
  for (int j = 0; j < numThermalization; j++) {
    walker().doThermalizationStep();
  }
}


// ***************** adaptiveStepThermalization ******************
template <class SlaterDeterminant>
void Vmc<SlaterDeterminant>::adaptiveStepThermalization() {
  *outputFile << " Thermalization: " << numThermalization;
  wf().initializeThermalization();
  int frac = domain().getNumberSeekAcceptance();
  double soughtAcceptance = domain().getSoughtAcceptance();
  if (domain().getVarySeekWithRank())
    soughtAcceptance -=
      (centerRank - rank) * domain().getVarySeekWithRankStep();
  int numThermFrac = numThermalization/frac;
  if (numThermFrac>0) {
    for (int i=0; i<frac; i++) {
      int acceptance = 0;
```

```
      for (int j = 0; j < numThermFrac; j++) {
        acceptance += walker().doThermalizationStep();
      }
      domain().changeStepLen(numThermFrac, acceptance,
          soughtAcceptance);
    }
  }
  *outputFile << " StepLenght changed to: " <<  domain().getStepLen();
}


// ************************* vmcSome ****************************
template <class SlaterDeterminant>
void Vmc<SlaterDeterminant>::vmcSome() {
  int numParticles = domain().getNumParticles();
  domain().initVMC();
  wf().initializeThermalization();
  wf().initializeVMC();
  walker().resetAcceptance();
  *outputFile << " VMC run: " << numCycles;
  int acceptance = 0;
  for (int i = 0; i < numCycles; i++) {
    for (int j = 0; j < numParticles; j++) {
      acceptance += walker().doRandomStep();
    }
    wf().sample();
  }
  *outputFile << " Acceptance: "
              << (((double)acceptance) / (numCycles) / numParticles)
              << endl << endl;
  wf().summary();
  *outputFile << endl;
  wf().summaryLowest();
}


// ******************* vmcOneParticleAtATime *********************
template <class SlaterDeterminant>
void Vmc<SlaterDeterminant>::vmcOneParticleAtATime() {
  domain().initVMC();
  wf().initializeThermalization();
  wf().initializeVMC();
  walker().resetAcceptance();
  *outputFile << " VMC run: " << numCycles;
  int acceptance = 0;
```

```
  for (int i = 0; i < numCycles; i++) {
    acceptance += walker().doRandomStep();
    wf().sample();
  }

  *outputFile << " Acceptance: " <<
    (((double)acceptance) / (numCycles))    << endl << endl;
  wf().summary();
  *outputFile << endl;
  wf().summaryLowest();
}


// ************************ initNextRun ***************************
template <class SlaterDeterminant>
void Vmc<SlaterDeterminant>::initNextRun() {
  domain().setReferenceEnergy(wf().getLowestEnergy());

  domain().setCentralAlphaParam(wf().getLowestAlphaParams());
  domain().calculateAlphaParamArray();

  domain().setCentralBetaParam(wf().getLowestBetaParams());
  domain().calculateBetaParamArray();

  domain().init();
  wf().initNewVmcRun();
}


// ***************************** run ******************************
template <class SlaterDeterminant>
void Vmc<SlaterDeterminant>::run() {

  domain().initSummary();

  for (int i=0; i<numberOfUniDirectionalMoves; i++) {

    int numRuns=0;
    int uniDirection=1;
    *outputFile << "::::::::::::::::::::::::::::::::"
                << "::::::::::::::::::::::::::::::::" << endl;

    while ( (uniDirection) & (numRuns<numberVmcRuns) ) {
      domain().Summary();
      *outputFile << endl;
```

```
    // Thermalization
    if (thermalizationType == "thermalization" ) thermalization();
    else if (thermalizationType == "adaptiveStepThermalization" )
      adaptiveStepThermalization();
    else *outputFile
      << "Error in Vmc::run. Could not perform thermalization."
      << endl << "  Input thermalizationType not known." << endl;

    // VMC
    if (vmcType == "vmcSome" ) vmcSome();
    else if (vmcType == "vmcOneParticleAtATime" )
       vmcOneParticleAtATime();
    else *outputFile << "Error in Vmc::run. Could not perform vmc."
                     << endl << "  Input vmcType not known." << endl
                        ;

    *outputFile << "One VMC run finished." << endl;

    if (uniDirectionalMovement)
      uniDirection = domain().isMovementUniDirectional
        (wf().getLowestAlphaParams(), wf().getLowestBetaParams() );
    if (uniDirection) initNextRun();
    numRuns++;
  }

  if ( (uniDirectionalMovement) & (numRuns==numberVmcRuns) )
    *outputFile << "Error in Vmc:run! Could not find minima in
       numRuns = "
                << numberVmcRuns << " runs." << endl;

  if (uniDirectionalMovement) domain().initUniDirectionalIndicator()
     ;

  *outputFile << endl << endl;

  domain().increaseNumCycles();
  numCycles = domain().getNumCycles();
  domain().increaseNumThermalization();
  numThermalization = domain().getNumThermalization();
  domain().reduceLocalArea();
  initNextRun();

}
```

```
}

#endif
```

A.1.34 Walker.h

```
#include "../Variations/Variations.h"

// ************************************************************
// *                          WALKER                          *
// ************************************************************
template <class SlaterDeterminant>
class Walker {
 protected:

  int              accepted;          // The number of accepted steps.
  double           ratio;             // The product of slaterRatio
                                      // and correlationRatio.
  RefFund<double> slaterRatio;        // The slaterRatio.
  RefFund<double> correlationRatio; // The correlationRatio.

  Ref<Random2>                              random;
  Ref<Domain>                               domain;
  Ref<SlaterDeterminant>                    slaterDeterminant;
  Ref<Correl>                               correlation;
  Ref<Variations<SlaterDeterminant> > variations;

 public:
  Walker(Domain& _domain,
         SlaterDeterminant& _slaterDeterminant,
         Random2& _random, Correl& _correlation,
         Variations<SlaterDeterminant>& _variations);

  int                 doThermalizationStep();
  int                 doRandomStep();
  void                resetAcceptance();
  int                 getAcceptance();
  SlaterDeterminant* getSlaterDeterminantPtr() {return &(
     slaterDeterminant());}
};

#include "Walker.cpp"

#endif
```

A.1.35 Walker.cpp

```
#ifndef WalkerCPP_IS_INCLUDED
#define WalkerCPP_IS_INCLUDED
```

```
#include "Walker.h"


// ************************** Walker ****************************
template <class SlaterDeterminant>
Walker<SlaterDeterminant>::Walker(Domain& _domain,
                                  SlaterDeterminant&
                                      _slaterDeterminant,
                                  Random2& _random, Correl&
                                      _correlation,
                                  Variations<SlaterDeterminant>&
                                      _variations) {
  random             = _random;
  domain             = _domain;
  slaterDeterminant  = _slaterDeterminant;
  correlation        = _correlation;
  slaterRatio        =&slaterDeterminant().getRatio();
  correlationRatio   = correlation().getRatioPtr();
  variations         = _variations;
}


// ******************** doThermalizationStep *********************
template <class SlaterDeterminant>
int Walker<SlaterDeterminant>::doThermalizationStep() {
  domain().suggestMove();
  //domain().proposeFlip();
  variations().suggestMove();
  ratio = slaterRatio()*correlationRatio();

  // THE METROPOLIS TEST!
  if ( (ratio*ratio) > random().getNum()) {
    domain().acceptThermalizedMove();
    variations().acceptThermalization();
    accepted = 1;
  }
  else {
    domain().rejectThermalizedMove();
    variations().rejectThermalization();
    accepted = 0;
  }
  variations().setToNextParticle();
  domain().setToNextParticle();
  return accepted;
}
```

```
// ********************** doRandomStep ************************
template <class SlaterDeterminant>
int Walker<SlaterDeterminant>::doRandomStep() {
  domain().suggestMove();
  //domain().proposeFlip();
  variations().suggestMove();
  ratio = slaterRatio()*correlationRatio();
  // THE METROPOLIS TEST!
  if ( (ratio*ratio) > random().getNum()) {
    domain().acceptMove();
    variations().acceptMove();
    accepted = 1;
  }
  else {
    //domain().rejectTrialPosition();
    domain().rejectMove();
    variations().rejectMove();
    accepted = 0;
  }
  variations().setToNextParticle();
  domain().setToNextParticle();
  return accepted;
}


// ********************** resetAcceptance ************************
template <class SlaterDeterminant>
void Walker<SlaterDeterminant>::resetAcceptance() {
  slaterDeterminant().resetAcceptances();
}


// ********************** getAcceptance ************************
template <class SlaterDeterminant>
int Walker<SlaterDeterminant>::getAcceptance() {
  return slaterDeterminant().getMoveAcceptance();
}

#endif
```

A.1.36 SpinFactors.h

```
#ifndef SpinFactors_IS_INCLUDED
#define SpinFactors_IS_INCLUDED

#include <iostream>
using namespace std;

#ifndef dotProduct_IS_INCLUDED
#define dotProduct_IS_INCLUDED

inline double dotProduct(double* array1, double* array2, int dim) {
  double* a1 = array1 - 1;
  double* a2 = array2 - 1;
  double product = 0;
  for (int i=0; i<dim; i++)
    product += (*++a1)*(*++a2);
  return product;
}
#endif

class SpinFactors {

protected:
  int     numParticles;     // Number of particles.
  int     Nmatrix;          // Number of elements in the upper-
                            // triangular matrices.
  int     currentParticle; // This is the particle that
                            // currently is (proposed) moved.
  int     otherParticle;    // Particle to exchange spin with.

  int*    spinArray;        // Array directly accesing the
                            // particle spins.
  int*    _spinArray;       // spinArray pointer.
  int*    currentSpin;      // Spin of current particle.
  int*    otherSpin;        // Spin of other particle.

  double* newFactor;        // Array containing new factors.
  double* oldFactor;        // Array containing old factors.
  double* otherDifference; // Distance between new and old factors.
  double* matrix;           // Upper triangular matrix containing
                            // the spin factors (1/2 for like spins,
                            // 1 for anti-parllel spin).
  double* _newFactor;       // newFactor pointer.
  double* _oldFactor;       // oldFactor pointer.
```

```
  double* _otherDifference;// otherDifference pointer.
  double* _matrix;         // matrix pointer.

  int     quarterCusp;     // Boolean (1 = quarter cusp for like
                           // spin electrons, 0 = cusp 1/2 for all
                           // electrons).
  int     spinFlip;        // Boolen determining wether we allow spin
                           // flip.

  void    dumpNewFactor();
  void    dumpOldFactor();
  void    dumpOtherDifference();
  void    setNewFactor(double value);
public:
  SpinFactors() {}
  void    allocate(int _numParticles, int* _spinArray);
  void    init();
  // Call this routine if no spins are flipped
  // (OR the two spins are equal)
  void    calculateNoFlipFactors();
  // Call this routine ONLY if the two spins differ
  void    calculateFlipFactors();
  void    setToNextParticle();
  void    setCurrentParticle(int _currentParticle)
    {currentParticle = _currentParticle;}
  void    setOtherParticle(int _otherParticle)
    {otherParticle= _otherParticle;}
  void    setSpinFlip(int _spinFlip)       {spinFlip = _spinFlip;}
  void    setQuarterCusp(int _quarterCusp) {quarterCusp = _quarterCusp
     ;}
  double* getNewFactor()       {return newFactor;}
  double* getOldFactor()       {return oldFactor;}
  double* getOtherDifference() {return otherDifference;}
  double* getMatrix()          {return matrix;}
  int*    getSpinFlip()        {return &spinFlip;}
  int*    getOtherParticle()   {return &otherParticle;}
  void    calculateMatrix();
  void    summaryNoFlip();
  void    summaryFlip();
  void    summayMatrix();
};

#endif
```

A.1.37 SpinFactors.cpp

```
#include "SpinFactors.h"

// ************************ allocate ****************************
void SpinFactors::allocate(int _numParticles, int* _spinArray) {
  numParticles     = _numParticles;
  Nmatrix          = (numParticles*(numParticles-1))/2;
  spinArray        = _spinArray;
  newFactor        = new double[numParticles - 1];
  oldFactor        = new double[numParticles - 1];
  otherDifference = new double[numParticles - 1];
  matrix           = new double[Nmatrix];
  init();
}

// ************************** init ******************************
void SpinFactors::init() {
  setCurrentParticle(0);
  setOtherParticle(0);
  setSpinFlip(0);
  setNewFactor(1);
}


// ****************** calculateNoFlipFactors ********************
// Call this routine if no spins are flipped
// (OR the two spins are equal)
void SpinFactors::calculateNoFlipFactors() {
  _spinArray  = spinArray;
  currentSpin = _spinArray + currentParticle;
  _newFactor  = newFactor  - 1;
  for (int i=0; i<currentParticle; i++)
    if ( (*_spinArray++)==(*currentSpin) ) (*++_newFactor)=0.5;
    else (*++_newFactor)=1;
  for (int i=currentParticle+1; i<numParticles; i++)
    if ( (*++_spinArray)==(*currentSpin) ) (*++_newFactor)=0.5;
    else (*++_newFactor)=1;
}

// ******************* calculateFlipFactors *********************
// Call this routine ONLY if the two spins differ
void SpinFactors::calculateFlipFactors() {
  _spinArray  = spinArray;
  currentSpin = spinArray + currentParticle;
  _newFactor  = newFactor - 1;
  _oldFactor  = oldFactor - 1;
```

```
  for (int i=0; i<currentParticle; i++)
    if ( (*_spinArray++)==(*currentSpin) ) {
      (*++_newFactor)=1;
      (*++_oldFactor)=0.5;
    }
    else {
      (*++_newFactor)=0.5;
      (*++_oldFactor)=1;
    }
  for (int i=currentParticle+1; i<numParticles; i++)
    if ( (*++_spinArray)==(*currentSpin) ) {
      (*++_newFactor)=1;
      (*++_oldFactor)=0.5;
    }
    else {
      (*++_newFactor)=0.5;
      (*++_oldFactor)=1;
    }
  *(newFactor+otherParticle-(otherParticle>currentParticle)) = 1;

  _spinArray       = spinArray;
  otherSpin        = spinArray + otherParticle;
  _otherDifference = otherDifference - 1;
  for (int i=0; i<otherParticle; i++)
    if ( (*_spinArray++)==(*otherSpin) ) (*++_otherDifference)=0.5;
    else (*++_otherDifference)=-0.5;
  for (int i=otherParticle+1; i<numParticles; i++)
    if ( (*_spinArray++)==(*otherSpin) ) (*++_otherDifference)=0.5;
    else (*++_otherDifference)=-0.5;
  *(otherDifference+currentParticle-(currentParticle>otherParticle))
      = 0;
}


// ********************* setToNextParticle ************************
void SpinFactors::setToNextParticle() {
  currentParticle++;
  if (currentParticle == numParticles) setCurrentParticle(0);
}


// ********************* calculateMatrix ************************
void SpinFactors::calculateMatrix() {
  _matrix = matrix-1;
  for (int i=0; i<numParticles-1; i++)
```

```
    for (int j=i+1; j<numParticles; j++)
      if ( ( spinArray[i]==spinArray[j] ) & (quarterCusp) )
        (*++_matrix) = 0.5;
      else (*++_matrix) = 1;
}


// ********************** setNewFactor ************************
void SpinFactors::setNewFactor(double value) {
  _newFactor = newFactor - 1;
  for (int i=0; i<numParticles-1; i++)
    (*++_newFactor) = value;
}


// ********************** dumpNewFactor ************************
void SpinFactors::dumpNewFactor() {
  cerr << "Current Particle=" << currentParticle << " newFactor=[ ";
  _newFactor = newFactor - 1;
  for (int i=0; i<numParticles-1;i++)
    cerr << (*++_newFactor) << " ";
  cerr << " ]" << endl;
}


// ********************** dumpOldFactor ************************
void SpinFactors::dumpOldFactor() {
  cerr << "Current Particle=" << currentParticle << " oldFactor=[ ";
  _oldFactor = oldFactor - 1;
  for (int i=0; i<numParticles-1;i++)
    cerr << (*++_oldFactor) << " ";
  cerr << " ]" << endl;
}


// ******************** dumpOtherDifference ********************
void SpinFactors::dumpOtherDifference() {
  cerr << "Current Particle=" << currentParticle << "Other Particle="
      << otherParticle << " otherDifference=[ ";
  _otherDifference = otherDifference - 1;
  for (int i=0; i<numParticles-1;i++)
    cerr << (*++_otherDifference) << " ";
  cerr << " ]" << endl;
}
```

```
// ********************** summaryNoFlip ************************
void SpinFactors::summaryNoFlip() {
  dumpNewFactor();
}


// ************************ summaryFlip ************************
void SpinFactors::summaryFlip() {
  dumpNewFactor();
  dumpOldFactor();
  dumpOtherDifference();
}


// ************************ summayMatrix ************************
void SpinFactors::summayMatrix() {
  cerr << "Matrix =" << endl;
  _matrix = matrix-1;
  for (int i=0; i<numParticles-1; i++) {
    for (int j=0; j<i+1; j++)
      cerr << 0 << " ";
    for (int j=i+1; j<numParticles; j++)
      cerr << (*++_matrix) << " ";
  cerr << endl;
  }
}
```

Bibliography

[1] T. Helgaker, P. Jørgensen, and J. Olsen. *Molecular Electronic-Structure Theory.* (John Wiley & Sons, Inc., 2002).

[2] R. Shankar. *Principles of Quantum Mechanics, Second Edition.* (Plenum Press, 1994).

[3] P. C. Hemmer. *Kvantemekanikk.* (Tapir, 1980).

[4] P. W. Atkins and R. S. Freidman. *Molecular Quantum Mechanics.* (Oxford Univerity Press, 2003).

[5] B. H. Bransden and C. J. Joachain. *Physics of Atoms and Molecules.* (Longman Group Ltd, 1983).

[6] K. Rottmann. *Matematisk formelsamling.* (Spektrum forlag, 2003).

[7] J. W. Rohlf. *Modern Physics from α to Z^0.* (John Wiley & Sons, Inc., 1994).

[8] S. A. Alexander and R. L. Coldwell. Atomic wave-function forms. *Int. J. Quant Chem*, **63**:1001, 1997.

[9] E. Clementi and D. L. Raimondi. Atomic screening constants from scf functions. *J. Chem. Phys.*, **38**:2686, 1963.

[10] P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Phys. Rev. B*, **136**:864, 1964.

[11] P. Fulde. *Electron Correlations in Molecules and Solids.* (Springer, 1995).

[12] P. R. C. Kent. *Techniques and Applications of Qunatum Monte Carlo.* PhD thesis, 1999).

[13] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys.*, **21**:1087, 1953.

[14] T. Kato. *Comm. Pure Appl. Math*, **10**:151, 1957.

[15] B. L. Hammond, W. A. Lester Jr., and P. J. Reynolds. *Monte Carlo Methods in Ab Initio Quantumn Chemistry.* (World Scientific, 1994).

[16] E. Clementi and C. ROETTI. Roothaan-hartree-fock atomic wavefunctions. *Atomic Data and Nuclear Data Tables*, **14**:177, 1974.

[17] K. E. Schmidt and C. J. Umrigar. *J. Chem. Phys.*, **105**:4172.

[18] V. Popsueva. Quantum dots - a monte carlo simulation. Master's thesis, 2004.

[19] M. Dineykhan and R. G. Nazmitdinov. Two-electron quantum dot in a magnetic field: Analytical results. *Phys. Rev. B*, **55**:13707, 1997.