PARALLELIZATION OF MANY BODY CODES IN PHYSICS

by

ISLEN VALLEJO HENAO

THESIS
for the degree of
MASTER OF SCIENCE

(Master in Computational Physics)

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

November 2009

Det matematisk- naturvitenskapelige fakultet
Universitetet i Oslo

Contents

Chapter 1

Introduction

Mathematical modelling is the tool preferred by scientists and engineers for describing the behaviour of physical systems. Often, the more degrees of freedom a model includes, the more likely it is that its predictive power improves. In turn, increasing the number of variables in the model reduces the probability to find analytical solutions of the resulting equations, and numerical methods making use of computer simulations become essential.

In quantum mechanics, the non-relativistic Schrödinger equation plays the role of Newton's laws of motion in classical mechanics. It can model a vast quantity of physical systems of interest both in science and technology. The lack of analytical solutions justifies the use of computational methods. Because of the number of degrees of freedom involved, the use of grid-based numerical methods are not suitable for this kind of problems. Presently, Monte Carlo methods belong to so-called ab initio methods able to solve, in principle exactly, systems with many interacting particles.

A challenge that computational physicists may face is the ability to combine powerful numerical methods with efficient use of computational resources and programming techniques. This is what precisely has motivated the formulation of this thesis. The fact that in recent years many scientists and engineers have migrated to slower interpreted high level languages like Matlab, because of the abundant disponibility of documentation, clear and compact syntaxis, interactive execution of commands and the integration of simulation and visualizations have made interpreted languages highly attractive.

Successfull attempts to integrate high and low level languages in simulators solving partial differential equations (PDEs) have been done in the recent years. Examples of mixed codes are the Python/C++ libraries Dolphin [?, ?], GETFEM++ [?], etc, for solving PDEs by the finite element methods. To the author's knowledge, there are not many cases making use of this programming approach for quantum mechanical problems. An exception is PyQuante [?] doing the Hartree-Fock method and Density Functional theory for quantum chemistry. There seems to be a gap in the development of plain high level and mixed codes for quantum Monte Carlo techniques.

In this context, this thesis examines three ways of implementing a Quantum Variational Monte Carlo (QVMC) simulator. The first two consider the potential use of Python and Python/C++ (mixed languages) in computing the ground state energy using a straigthfor-

ward algorithm. Benchmarks of both implementations and profiles of the pure Python code are provided and a discussion on the advantages and disadvantages is carried out. A third alternative shows how to exploit the object oriented capabilities of C++ to develop robust, efficient, flexible and a user friendly implementation for QVMC.

When constructing many-electron wave functions for QVMC simulations, the picture of a trial wave function expressed as the product of a Slater determinant times a correlation function provides an easy way to include many of the physical features of quantum mechanical systems. A problem with the Slater determinant is that it is computationally demanding. Algorithms for treating this problem, besides optimizing the Jastrow factor, have already been suggested.

However, a question pending is the reduction of the computational time and accuracy in computing the derivatives of the variational energy with respect to the variational parameters. On the one hand, analytical evaluations of this expression are practical only for very small systems. On the other hand, the use of finite difference methods to approximate the gradients and laplacians of the energy, besides introducing inaccuracies associated with the step size, is computationally expensive.

In this thesis we suggest a methodological proposal for the development of scientific code. In particular, we show how to take advantages of the characteristics of high level programming languages in order to prototype big implementations. The analysis of the performance of plain Python against C++ has been carried out. We also develop an optimized C++ code (about 8000 lines) capable of computing the ground state energy (in serie or in parallel) of several quantum mechanical systems using the Quantum Variational Monte Carlo method. Among the physical systems implemented are the helium and beryllium atoms and two-dimensional quantum dots for two and six electrons[1]. It is rather simple to simulate systems with more particles and single-particle orbits due to the object-oriented form of the code.

We have developed a new method for computing analytically the derivatives of the variational energy with respect to the variational parameters. Its main peculiarity is that in spite of being simple in terms of code development it is also accurate and computationally efficient. This constitutes one of the main contributions of this thesis.

In order to provide a clear description of the QVMC algorithm, most of the central equations required have been derived in details from scratch. Moreover, a complete documentation of the implementation can be found in the CD provided with this thesis. We hope that part of this material can be used as pedagogical tool for other students in computational physics.

The rest of this thesis is organised as follows. Chapter 2 gives an introduction to the many body problem in quantum mechanics. Further, a general description of the numerical method (variational Monte Carlo) is given in chapter 3. Because of the crucial role the trial wave functions play in variational Monte Carlo (and other Monte Carlo methods), chapter 4 is dedicated to this topic, focusing on the algorithms that improve the computational

[1] All the systems considered are modelled in terms of the non-relativistic Schrödinger equation. Moreover, all evaluations of the ground state energy are carried out within so-called closed shell systems.

performance. The cases of study, including analytical derivations to validate the code are discussed in chapter 5. Then, a detailed description concerning the software design and implementation is discussed in chapter 6. Finally, chapters 7 and 8 deal with computational experiments and conclusions, respectively.

Chapter 2

Physical background

2.1 The quantum mechanical state

In quantum mechanics, the state of a system is fully defined by its complex wave function

$$\Phi(\boldsymbol{x}, t) = \Phi(\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_N}, t),$$

with $\boldsymbol{x}_i$ being the set of coordinates (spatial or spin) that define the attributes of the $i^{th}-$particle. The interpretation of the wave function is that $|\Phi(\boldsymbol{x}, t)|^2$ represents the probability density of a measure of the particles' displacements yielding the value of $\boldsymbol{x}$ at time t. Then, the (quantum mechanical) probability of finding a particle $i = 1, 2, \ldots, N$ between $\boldsymbol{x_i}$ and $\boldsymbol{x_i} + d\boldsymbol{x_i}$ at time t is given by [?]

$$\begin{aligned} P(\boldsymbol{x}, t)d\boldsymbol{x} &= \Phi^*(\boldsymbol{x}, t)\Phi(\boldsymbol{x}, t)d\boldsymbol{x} = |\Phi(\boldsymbol{x}, t)|^2 d\boldsymbol{x} \\ &= |\Phi(\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_N}, t)|^2 d\boldsymbol{x_1} d\boldsymbol{x_2} \ldots d\boldsymbol{x_N}, \end{aligned} \tag{2.1}$$

which is a real and positive definite quantity. This is Max Born's postulate on how to interpret the wave function resulting from the solution of Schrödinger's equation. It is also the commonly accepted and operational interpretation.

Since a probability is a real number between 0 and 1, in order for the Born interpretation to make sense, the integral of all the probabilities must satisfy the normalization condition

$$\begin{aligned} \int_{-\infty}^{\infty} P(\boldsymbol{x}, t)d\boldsymbol{x} &= \int_{-\infty}^{\infty} |\Phi(\boldsymbol{x}, t)|^2 d\boldsymbol{x} \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \ldots \int_{-\infty}^{\infty} |\Phi(\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_N}, t)|^2 d\boldsymbol{x_1} d\boldsymbol{x_2} \ldots d\boldsymbol{x_N} = 1 \end{aligned} \tag{2.2}$$

at all the times. In consequence, $\int_{-\infty}^{\infty} \Phi(\boldsymbol{x}, t)^* \Phi(\boldsymbol{x}, t)d\boldsymbol{x} < \infty$, which implies that the wave function and its spatial derivative has to be finite, continuous and single valued. Moreover, since the probabilities are square integrable wave functions, this means that the Born interpretation constrains the wave function to belong to the class of functions in L^2, i.e., the wave function is a square-integrable complex-valued function.

2.2 Evolution of the quantum mechanical state

The temporal evolution of the wave function (and therefore of a quantum mechanical system) is given by the time-dependent Schröndinger equation

$$i\hbar\frac{\partial\Phi(\boldsymbol{x},t)}{\partial t} = \widehat{\mathbf{H}}\Phi(\boldsymbol{x},t), \tag{2.3}$$

where $\widehat{\mathbf{H}}$ is the Hamiltonian operator[1]. In fact, any measurable physical quantity (observable) $A(\boldsymbol{x},\boldsymbol{p})$, depending on position and momentum, has associated its corresponding quantum mechanical operator. By setting $\boldsymbol{x} \rightarrow \widehat{\mathbf{x}}$ and $\boldsymbol{p} \rightarrow -i\hbar\boldsymbol{\nabla}$ we get for operator $\widehat{\mathbf{A}} = A(\widehat{\mathbf{x}}, -i\hbar\boldsymbol{\nabla})$.

The Hamiltonian operator in Eq. (2.3) corresponds to the total energy of the system. It may be expressed in terms of a kinetic and potential operator as

$$\boxed{\widehat{\mathbf{H}} = \widehat{\mathbf{K}}(\boldsymbol{x}) + \widehat{\mathbf{V}}(\boldsymbol{x},t),} \tag{2.4}$$

where the kinetic energy becomes

$$\boxed{\widehat{\mathbf{K}} = \sum_{i=1}^{N}\widehat{\mathbf{t}}(\boldsymbol{x_i}) = \sum_{i=1}^{N}\frac{\widehat{\mathbf{p}}_i^2}{2m} = -\sum_{i=1}^{N}\frac{\nabla_i^2}{2m}.} \tag{2.5}$$

When the potential energy does not change with xtime,

$$\boxed{\widehat{\mathbf{V}}(\boldsymbol{x}) = \underbrace{\sum_{i=1}^{N}\widehat{\mathbf{u}}(\boldsymbol{x_i})}_{\substack{\text{One-particle}\\ \text{interaction}}} + \overbrace{\underbrace{\sum_{i>j}^{N}\widehat{\mathbf{v}}(\boldsymbol{x_i},\boldsymbol{x_j})}_{\substack{\text{Two-particle}\\ \text{interaction}}} + \cdots}^{\text{Correlations}}.} \tag{2.6}$$

and a solution to Eq. (2.3) may be found by the technique of separation of variables. Expressing the total N-body wave function as $\Phi(\boldsymbol{x},t) = \Psi(\boldsymbol{x})T(t)$ and after substitution in Eq. (2.3) one obtains two solutions, one of which is the time-independent Schröndinger equation

$$\boxed{\widehat{\mathbf{H}}\Psi(\boldsymbol{x}) = E\Psi(\boldsymbol{x}).} \tag{2.7}$$

The only possible outcome of an ideal measurement of the physical quantity A are the eigenvalues of the corresponding quantum mechanical operator $\widehat{\mathbf{A}}$, i.e., the set of energies $E_1, E_2, \cdots, E_n$ are the only outcomes possible of a measurement and the corresponding eigenstates $\Psi_1, \Psi_2, \cdots, \Psi_n$ contain all the relevant information about the system in relation with the operator they are eigenstates of [?].

In this thesis we are concerned with solving the non-relativistic time-independent Schrödinger

[1] An operator $\widehat{\mathbf{A}}$ is a mathematical entity which transforms one function into another [?], . In the following we will see, for example, that $\widehat{\mathbf{H}}$ is a differential operator, then the expressions containing it are differential equations.

equation (2.7) for electronic system described by central symmetric potentials including a maximum of two-body interations as expressed in Eq. (2.6). In the following we will omit the time dependence in the equations.

2.3 System of non-interacting particles

The substitution of Eq. (2.5) into Eq. (2.4) yields the following Hamiltonian for a time-independent and non-relativistic multi-particle system

$$\widehat{\mathbf{H}}(\boldsymbol{x_1},\ldots,\boldsymbol{x_N}) = \sum_{i=1}^{N}\left[-\frac{1}{2m}\nabla_i^2 + \widehat{\mathbf{V}}(\boldsymbol{x_1},\ldots,\boldsymbol{x_N})\right], \tag{2.8}$$

where N is the number of electrons in the system. The first term in the bracket represents the total kinetic energy. The potential term $V(\boldsymbol{x})$ includes the nature of the interactions between the various particles making up the system as well as those with external forces, as described in Eq. (2.6).

Assuming that the particles do not interact is equivalent to say that each particle moves in a common potential, i.e., $V(\boldsymbol{x}) = \sum_{i=1}^{N} V_i(x_i)$ and therefore we can write

$$\widehat{\mathbf{H}} = \sum_{i=1}^{N}\left[-\frac{1}{2m}\nabla_i^2 + \widehat{\mathbf{V}}(\boldsymbol{x_i})\right] = \sum_{i=1}^{N}\widehat{\mathbf{h}}_i, \tag{2.9}$$

where $\widehat{\mathbf{h}}_i$ represents the energy of the i^{th} particle. This quantity is independent of the other particles. The assumption of a system of non-interaction particles implies also that their positions are independent of each other. Hence, the total wave function can be expressed as a product of $N-$independent single particle wave functions[2] [?, ?, ?],

$$\Psi(\boldsymbol{x}) = \Psi(\boldsymbol{x_1},\ldots,\boldsymbol{x_N}) = \phi\alpha_1(\boldsymbol{x_1})\phi\alpha_2(\boldsymbol{x_2})\ldots\phi\alpha_N(\boldsymbol{x_N}) = \prod_{i=1}^{N}\phi_i(\boldsymbol{x_i}), \tag{2.10}$$

where α_i denotes the set of quantum numbers necessary to specify a single electron state.

Comparing with Eq. (2.7) we arrive to

$$\widehat{\mathbf{H}}\Psi(\boldsymbol{x}) = \sum_{i=1}^{N}\widehat{\mathbf{h}}_i\phi_i(\boldsymbol{x_i}) = E_i\phi_i(\boldsymbol{x_i}) = E\Psi(\boldsymbol{x}). \tag{2.11}$$

What this shows is that for a system with $N-$non-interating particles, the total energy of the system equals the sum of the total energy of each single particle, i.e.,

$$\boxed{E = \sum_{i=1}^{N} E_i.} \tag{2.12}$$

[2] This kind of total wave function is known as the product state or Hartree product.

2.4 Identical particles and the Pauli exclusion principle

While the functional form of Eq. (2.10) is fairly convenient, it violates a very fundamental principle of quantum mechanics: The principle of indistinguishability. It states that it is not possible to distinguish identical particles[3] one from another by intrisic properties such as mass, charge and spin.

Denoting the $N-$body wave function by $\Psi(\boldsymbol{x_1}, \ldots, \boldsymbol{x_i}, \ldots, \boldsymbol{x_j}, \ldots, \boldsymbol{x_N})$ and using the probabilistic interpretation given by Eq. (2.1), the squared wave function gives us the probability of finding a particle i between $\boldsymbol{x_i}$ and $\boldsymbol{x_i} + d\boldsymbol{x_i}$ and another particle j between $\boldsymbol{x_j}$ and $\boldsymbol{x_j} + d\boldsymbol{x_j}$, and so on. Because there is no interaction that can distinguish the particles, a physical observable must be symmetric with respect to the exchange of any pair of two particles. Then, for particles i and j exchanging labels we get

$$|\Psi(\boldsymbol{x_1}, \ldots, \boldsymbol{x_i}, \ldots, \boldsymbol{x_j}, \ldots, \boldsymbol{x_N})|^2 = |\Psi(\boldsymbol{x_1}, \ldots, \boldsymbol{x_j}, \ldots, \boldsymbol{x_i}, \ldots, \boldsymbol{x_N})|^2,$$

which can be written as

$$|\Psi_{ij}|^2 = |\Psi_{ji}|^2, \tag{2.13}$$

implying that $|\Psi_{ij}| = e^{i\varphi}|\Psi_{ji}|$, where φ is a real number representing the phase of the wave function.

Using the same argument as above, we can swap the index again yielding $|\Psi_{ji}| = e^{i\varphi}|\Psi_{ij}|$. Inserting this expression in Eq. (2.13) leads to

$$|\Psi_{ij}|^2 = e^{i2\varphi}|\Psi_{ji}|^2.$$

That is, $e^{i2\varphi} = 1$, which has two solutions given by $\varphi = 0$ or $\varphi = \pi$. In other words,

$$|\Psi_{ij}| = \pm|\Psi_{ji}|.$$

Then, for a system consisting of N identical particles, its total wave function must be either symmetric or antisymmetric under the exchange of two particle labels. Particles described by symmetric functions are said to obey Bose-Einstein statistics and are known as bosons. Example are photons and gluons. On the other hand, particles having antisymmetric total wave functions are known as fermions and they are modeled by Fermi-Dirac statistics. In this group we have electrons, protons, neutrons, quarks and other particles with half-integer spin values. Bosons have integer spin while fermions have half-integer spin.

In order to illustrate what the particle symmetry principle implies, let $\Psi(x_i, E)$ be a stationary single particle wave function corresponding to a state with energy E. Later, assume that it is properly normalized and consider first a system formed by two identical and non-interacting bosons. A total wave function describing the system and fulfilling the symmetry principle under exchange of labels is

$$\Psi_{\text{boson, E}}(\boldsymbol{x_1}, \boldsymbol{x_2}) = \frac{1}{\sqrt{2}}\left[\psi(\boldsymbol{x_1}, E_\alpha)\psi(\boldsymbol{x_2}, E_\beta) + \psi(\boldsymbol{x_2}, E_\alpha)\psi(\boldsymbol{x_1}, E_\beta)\right].$$

[3] Of course one can distinguish, for example, an electron from a proton by applying a magnetic field, but in general there are not experiments nor theory letting us to distinguish between two electrons.

Since the particles are identical, we cannot decide which particle has energy E_α or E_β, only that one has energy E_α and the other E_β. Note that the particles can be in the same quantum mechanical state, i.e., $E_\alpha = E_\beta$.

For the case of two non-interacting identical fermions, the symmetry principle requires the total wave function to be antisymmetric, that is

$$\Psi_{\text{fermion, E}}(\boldsymbol{x_1}, \boldsymbol{x_2}) = \frac{1}{\sqrt{2}} \left[\psi(\boldsymbol{x_1}, E_\alpha)\psi(\boldsymbol{x_2}, E_\beta) - \psi(\boldsymbol{x_2}, E_\alpha)\psi(\boldsymbol{x_1}, E_\beta)\right]. \tag{2.14}$$

Exchanging the labels of the particles fulfills the symmetry principle, but note that if $E_\alpha = E_\beta$ the total wave function becomes zero, which means that it is impossible for any two identical fermions in an $N-$body system to occupy the same single particle (quantum mechanical) state. This important result is known as the Pauli exclusion principle.

The number appearing in the denominator of the normalization factor equals the number of permutations of $\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_N}$ in $\phi_{\alpha_1}(\boldsymbol{x_1}), \phi_{\alpha_2}(\boldsymbol{x_2}), \ldots, \phi_{\alpha_N}(\boldsymbol{x_N})$, which is in general $N!$. Then, a total wave function satisfying the Pauli exclusion principle is

$$\boxed{\Psi_D(\boldsymbol{x}) = \frac{1}{\sqrt{N!}} \begin{vmatrix} \phi_{\alpha_1}(\boldsymbol{x_1}) & \phi_{\alpha_2}(\boldsymbol{x_1}) & \cdots & \phi_{\alpha_N}(\boldsymbol{x_1}) \\ \phi_{\alpha_1}(\boldsymbol{x_2}) & \phi_{\alpha_2}(\boldsymbol{x_2}) & \cdots & \phi_{\alpha_N}(\boldsymbol{x_2}) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{\alpha_1}(\boldsymbol{x_N}) & \phi_{\alpha_2}(\boldsymbol{x_N}) & \cdots & \phi_{\alpha_N}(\boldsymbol{x_N}) \end{vmatrix},} \tag{2.15}$$

which is known as a Slater determinant. Observe that if two particles i and j are in the same state ($\alpha_i = \alpha_j$), two of the rows will be equal and the determinant will become zero, telling us that two fermions cannot occupy the same quantum mechanical state, in concordance with Pauli's exclusion principle. Moreover, exchange of two columns gives as result a change in the sign of the determinant, meaning that the total wave function of a fermionic system is antisymmetric under the exchange of two particles.

2.5 The Born-Openheimer approximation

The Hamiltonian of a quantum mechanical system is given by Eq. (2.4). For atoms, the kinetic energy operator of Eq. (2.5) takes the form

$$\hat{T}(\mathbf{x}) = \underbrace{-\frac{\hbar^2}{2M}\nabla_0^2}_{\substack{\text{Kinetic energy} \\ \text{of the nucleus}}} - \underbrace{\sum_{i=1}^{N} \frac{\hbar^2}{2m}\nabla_i^2}_{\substack{\text{Kinetic energy} \\ \text{of the electrons}}},$$

where M is the mass of the nucleus and m is the electron mass.

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. Moreover, because the forces between the particles are of similar magnitude due to their similar charge, one can assume that the momenta of the particles are also similar. Hence, 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 with the nucleus at a fixed location[4]. This separation of the electronic and nuclear degrees of freedom is known as the Born-Oppenheimer approximation [?, ?].

In the center of mass (CM) reference system the kinetic energy operator reads

$$\hat{T}(\mathbf{x}) = \underbrace{-\frac{\hbar^2}{2(M+Nm)}\nabla_{CM}^2}_{\substack{\text{Kinetic energy of}\\ \text{the center of mass}}} \underbrace{-\frac{\hbar^2}{2\mu}\sum_{i=1}^{N}\nabla_i^2}_{\substack{\text{Kinetic energy}\\ \text{of N electrons}}} \underbrace{-\frac{\hbar^2}{M}\sum_{i>j}^{N}\nabla_i\cdot\nabla_j}_{\substack{\text{Mass polarization due}\\ \text{to nuclear motion}}} ,$$

where $\mu = mM/(m+M)$ is the reduced mass for each of the N electrons, i.e., its mass m replaced by μ, because of the motion of the nucleus.

In the limit $M \to \infty$ the first and third terms of the equation above become zero. Then, the kinetic energy operator reduces to

$$\boxed{\hat{T} = -\sum_{i=1}^{N}\frac{\hbar^2}{2m}\nabla_i^2.} \tag{2.16}$$

On the other hand, the potential energy operator Eq. (2.6) is given by

$$\boxed{\hat{V}(\mathbf{x}) = \underbrace{-\sum_{i=1}^{N}\frac{Ze^2}{(4\pi\epsilon_0)r_i}}_{\substack{\text{Nucleus-electron}\\ \text{potential}}} + \underbrace{\sum_{i=1,i<j}^{N}\frac{e^2}{(4\pi\epsilon_0)r_{ij}}}_{\substack{\text{Electron-electron}\\ \text{potential}}},} \tag{2.17}$$

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

2.5.1 Hydrogen like-atom

An hydrogen atom consist of a proton sorrounded by an electron, being it the simplest of all the atoms. In the Born-Openheimer approximation, the non-relativistic Hamiltonian is given by

$$\widehat{\mathbf{H}} = -\frac{\hbar^2}{2m}\frac{1}{2}\nabla^2 - \frac{Ze^2}{4\pi\epsilon_0}\frac{1}{r}. \tag{2.18}$$

Because of the form of the Coulomb potential, the problem is said to have spherical symmetry. For the above Hamiltonian, it is possible to get an analytical solution of the time-independent Schröndiger equation (2.7) by separation of variables. Rewritting the potential

[4]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. References are books on elementary physics or chemistry.

in terms of spherical coordinates and setting

$$\Psi(r,\theta,\phi) = R(r)P(\theta)F(\phi), \tag{2.19}$$

leads to a set of three ordinary second-order differential equations whose analytical solutions give rise to three quantum numbers associated with the energy levels of the hydrogen atom.

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

$$Y_{lm_l}(\theta,\phi) = P(\theta)F(\phi) = \sqrt{\frac{(2l+1)(l-m_l)!}{4\pi(l+m_l)!}}P_l^{m_l}(cos(\theta))\exp(im_l\phi), \tag{2.20}$$

with $P_l^{m_l}$ being the associated Legendre polynomials.

The quantum numbers l and m_l represent the orbital momentum and projection of the orbital momentum, respectively and take the values $l = 0, 1, 2, \ldots$ and $-l \leq m_l \leq l$.

In general the radial Schrödinger equation (in atomic units) reads,

$$\left[-\frac{1}{2}\left(\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d}{dr}\right) - \frac{l(l+1)}{r^2}\right) + \frac{Z}{r}\right]R(r) = ER(r). \tag{2.21}$$

2.6 When two charged particle approach each other: cusp conditions

Cusp conditions are boundary conditions that must be satisfied by the wave function where the potential diverges. In general, the divergencies appear when the distance between interacting particles approaches zero. Under these circumstances, the forces between these two particles become dominant. Because the local energy is constant everywhere in the space when the wave function is exact, we should take care of constructing wave functions that, at least, make the local energy finite in space. As a consequence, each of the singularities in the potential should be cancelled by a corresponding term in the kinetic energy. This condition results in a discontinuity (cusp) in the first derivative of the wave function Ψ anywhere two charged particles meet. We analyze these two situations in the following.

2.6.1 Electron-nucleus cusp of a hydrogenic atom

The exact wave function of hydrogenic atoms in spherical polar coordinates can be written as a product between a radial function $R(r)$ and an angular part $\Omega(\theta,\phi)$, yielding

$$\Psi(\boldsymbol{r},\theta,\phi) = R(r)\Omega(\theta,\phi).$$

Here we are interested in the behaviour of Ψ as $r \to 0$. Therefore we examine just the radial part, which is the solution to the radial hydrogenic Schrödinger equation,

$$\left(\frac{d^2}{dr^2} + \frac{2}{r}\frac{d}{dr} + \frac{2Z}{r} - \frac{l(l+1)}{r^2} + 2E\right)R(r) = 0, \tag{2.22}$$

where Z is the atomic charge, and l is the quantum number for the orbital angular momentum.

We consider first the case for $l = 0$. For a given electron approaching the nucleus labeled, in order for the equation above to be fulfilled $\frac{2}{r}\frac{d}{dr} + \frac{2Z}{r} = 0$, leading to the cusp condition

$$\boxed{\left(\frac{1}{R(r)}\frac{dR(r)}{dr}\right)\Big|_{r=0} = -Z, \qquad R(0) \neq 0.} \tag{2.23}$$

For $l > 0$, we factor out the radial wave function as $R(r) = r^l\rho(r)$, with $\rho(r)$ being a function not going to zero at the origin. Substitution into the radial Schrödinger equation leads to

$$\frac{2(l+1)}{r}\rho'(r) + \frac{2Z}{r}\rho(r) + \rho''(r) + 2E\rho(r) = 0$$

Equating the diverging $1/r-$terms we get the general electron-nucleus cusp condition,

$$\boxed{\frac{\rho'}{\rho}\Big|_{r=0} = -\frac{Z}{l+1}.} \tag{2.24}$$

The generalization for the many-electron case follows from considering the asymptotic behaviour of the exact wave function as two electrons approach each other.

2.6.2 Electron-electron cusp conditions

For this case we consider an electron i approaching an electron j, constituting a two-body problem. Now, both of them contribute to the kinetic energy. The expansion of the wave function in term of the relative distance only yields

$$\left(2\frac{d^2}{dr_{ij}^2} + \frac{4}{r_{ij}}\frac{d}{dr_{ij}} + \frac{2}{r_{ij}} - \frac{l(l+1)}{r_{ij}^2} + 2E\right)R_{ij}(r_{ij}) = 0, \tag{2.25}$$

where r_{ij} is the distance between the two electrons. As in the previous section, this equation leads to the electron-electron cusp condition

$$\boxed{\frac{\rho'_{ij}}{\rho_{ij}}\Big|_{r=0} = \frac{1}{2(l+1)}.} \tag{2.26}$$

For a pair of electrons with opposite spins, the wave function with the lowest energy is an s-state ($l = 0$). Moreover, because of the Pauli exclusion principle, for two electrons with parallel spin, the wave function with lowest energy is a p-state ($l = 1$). With these two conditions the right hand side equals 1/4 for parallel spin and 1/2 for antiparallel spin.

2.7 Further considerations

Since the Slater determinant Ψ_D does not depend on the inter-electronic distance r_{ij} where i and j are electrons of opposite spin values, $\partial\Psi_D/\partial r_{ij} = 0$ and Ψ_D cannot satisfy the electron-electron cusp condition. On the other hand, for parallel spin $\Psi_D \to 0$ as rij because of the Paulis principle. By setting $\Psi_D(\boldsymbol{x_i}, \boldsymbol{x_j}, \ldots) = \Psi_D(\boldsymbol{x_i}, \boldsymbol{x_i} + \boldsymbol{r_{ij}}, \ldots)$, where

$\boldsymbol{r_{ij}} = \boldsymbol{x_j} - \boldsymbol{x_i}$ and expanding around $\boldsymbol{x_j} = \boldsymbol{x_i}$ or $r_{ij} = 0$ we get,

$$\Psi_D(\boldsymbol{x_i}, \boldsymbol{x_j}, \ldots) = \Psi_D(\boldsymbol{x_i}, \boldsymbol{x_i} + \boldsymbol{r_{ij}} = 0, \ldots) + r_{ij} \frac{\partial \Psi_D}{\partial r_{ij}}\Big|_{r_{ij}=0} + \ldots$$

The first term is zero, since two electrons cannot occupy the same position having parallel spin. Furthemore, $\frac{\partial \Psi_D}{\partial r_{ij}} \neq 0$ in most cases [?]. As in section 2.6.1 we set $\Psi_D = r_{ij}\rho_{ij}$, but because ρ_{ij} is independent on r_{ij}, the Slater determinant cannot satisfy the electron-electron cusp conditions. In order to cancel this singularity, the wave function has to be modified [?, ?].

There are, however, ways of taking into account these types of correlations. Typically, a Slater determinant having as entries single particle wave functions is multiplied by a product of correlation functions depending of the inter-electronic coordinates $f(r_{ij}) \equiv f_{ij}$. The independence that they frequently exhibit as function of the single particle wave functions makes it possible to write the total wave function as $\Psi = \Psi_{SD}\Psi_C$, with Ψ_{SD} and Ψ_C representing the Slater determinant and the correlation part, respectively. The explicit dependence of f_{ij} on r_{ij} is that the total wave function can now satisfy the cusp conditions by constraining Ψ_C so that for three spatial dimensions,

$$\boxed{\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial r_{ij}} = \begin{cases} \frac{1}{4}, & \text{like-spin.} \\ \frac{1}{2}, & \text{unlike-spin.} \end{cases}} \tag{2.27}$$

For the two-dimensional case [?],

$$\boxed{\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial r_{ij}} = \begin{cases} \frac{1}{3}, & \text{like-spin.} \\ 1, & \text{unlike-spin.} \end{cases}} \tag{2.28}$$

2.8 A final note on units and dimensionless models

When solving equations numerically, it is often convenient to rewrite the equation in terms of dimensionless variables. Several of the constants may differ largely in value leading to potential losses of numerical precision. Moreover, the equation in dimensionless form is easier to code, sparing one for eventual typographic errors. It is, also, common to scale atomic units (table 2.1) by setting $m = e = \hbar = 4\pi\epsilon_0 = 1$.

With the ideas outlined above, the next chapter will be devoted to developing the theory behind the variational Monte carlo method with a strategy to solve Schröndinger equation numerically.

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 used in atomic and molecular physics calculations.

Chapter 3

Numerical methods: Quantum Monte Carlo

The progress in the undertanding of quantum mechanics relies in the posibility of solving the many-body Schrödinger equation (2.7). In most of the quantum mechanical problems of interest, however, the total number of particles interacting is usually sufficiently large that an exact solution cannot be found. This fact motivates the use of numerical approximations and methods.

3.1 Monte Carlo integration

Often, scientists and engineers face the problem of solving the integral $I = \int_{x_1}^{x_2} f(x)dx$ numerically, with various quadrature rules among the methods most widely used. The recipe consists of evaluating $f(x)$ at points x_i on a grid, after which the weighted average of the values of $f(x_i)$ is taken. Both the sampling points and the weights are predefined, but they vary from method to method. A limitation with this approach is the difficulty of dealing with problems contaning a large number of strongly coupled degrees of freedom, due to the rapid growth of the number of grid points with dimensionality.

In contrast, Monte Carlo integration (MCI) methods are stochastic (statistical), where the sampling points are choosen at "random" from a probability distribution (PDF), i.e., the integrand is evaluated using a set of (pseudo) randomly generated grid points. The advantage of MCI is that the statistical error becomes proportional to $1/\sqrt{N}$ [?], with N being the number of samples, which is independent of the dimensionality of the integral. Moreover, it scales reasonably well with the size of the system. For a system of N particles, it means often a number between N^2 and N^3 when combined with other techniques. The primary disadvantage is that the calculated quantity contains a statistical uncertainty, which requires to increase the number of samples N as the uncertainty decreases as $N^{1/2}$ [?].

3.1.1 Probability distribution functions

A probability density function $p(x)$ describes the probabilities of all the possible events and it may be discrete or continuous. Therefore the sum or integral of the probabilities must equal unity. If the $p(x)dx$ is the probability that an event ocurrs between x and $x + dx$,

then the probability distribution function $PDF(x) \equiv \int_{-\infty}^{x} p(x)dx$ represents the posibility that the value of a single given point is less than equal x.

Two continuous PDFs with especial importance in the context of this thesis are the uniform and Gaussian distributions, whose density functions are

$$p(x) = \begin{cases} \frac{1}{\beta-\alpha} & \alpha \leq x \leq \beta \\ 0 & \text{otherwise.} \end{cases} \qquad \text{and} \qquad p(x) = \frac{e^{-\frac{(x-\mu)^2}{2\sigma^2}}}{\sigma\sqrt{2\pi\sigma}}, \tag{3.1}$$

respectively.

The expectation value of a function $f(x)$ with respect to a probability function $p(x)$ is said to be the average value with respect to the density p and it is given by

$$\langle f \rangle \equiv \int_{-\infty}^{\infty} f(x)p(x)dx \tag{3.2}$$

3.1.2 Measures of dispersion of data

Monte Carlo simulations can be treated as computer experiments and the results can be analysed with the same statistical tools we would use in analysing laboraty experiments. The expectation values we are looking for can contain two kind of errors. A statistical error, telling how accurate the estimate is, and one associated with the kind of numerical method employed. Thus quantity is normally called the (systematic error)[1]. Measures of dispersion are important for describing the spread of the data, or its variations around a central value.

Uncorrelated data

When it is assumed that the succesive values of the random quantity A are statistically independent the variance of the mean is given by

$$var = \sigma^2 = \langle A^2 \rangle - \langle A \rangle^2, \tag{3.3}$$

Moreover, the standard deviation of the mean is $\sigma = \sqrt{var}$, and it gives, roughly speaking, the width of a normally distributed dataset

$$\sigma = \sqrt{\frac{1}{N}\sum_i x_i^2 - \frac{1}{N^2}(\sum_i x_i)^2},$$

where N is the total number of points in the dataset. Using the variance we just calculated, the standard error can be expressed as

$$S_E = \sqrt{\frac{\text{var}}{N_{\text{eff}}}}.$$

that returns the variance and "naive" standard error of your dataset, the standard error assuming that all points in your dataset were independent, i.e., the number of effective points in a dataset equals the total number of points in the experiement $N_{\text{eff}} = N$.

[1] In variational Monte Carlo methods a common source for the systematic error is the so-called step length.

Correlated data and datablocking

If the samples are correlated, it can be shown [?,?] that

$$\sigma = \sqrt{\frac{1 + 2\tau/\Delta t}{n}(\langle A^2\rangle - \langle A\rangle^2)},$$

where τ is the correlation time, i.e., the time between two uncorrelated samples and Δt is the time between each sample. For $\Delta t \gg \tau$, the estimate of σ assuming uncorrelated samples still holds. The common case however, is that $\Delta t < \tau$. In this case what we do is to divide the sequence of samples into blocks, hence the expression datablocking. Then, we take the mean $\langle A_i\rangle$ of block $i = 1 \ldots n_{blocks}$ to calculate the total mean and variance. The size of each block must be so large that sample j of block i is not correlated with sample j of block $i + 1$. The correlation time τ would be a good choice, but it is not known in advance or is too expensive to compute. What we know, however, is that by increasing the blocksize, the variance should increase exponentially until it reaches a plateau, indicating that the blocks are no longer correlated with each other. The corresponding block size will be an estimate for the correlation time for the true correlated data

3.2 The variational principle

The variational principle states that the ground state energy E_0 of a quantum mechanical system is always less or equal than the expectation value of the Hamiltonian $\widehat{\mathbf{H}}$ calculated with a trial wave function Ψ_T. The Variational Monte Carlo method is based on this principle, and it will be used here to compute the expectation value of the energy given a Hamiltonian $\widehat{\mathbf{H}}$ and a trial wave function Ψ_T. This expectation values is given the following expression

$$\langle\widehat{\mathbf{H}}\rangle = \frac{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R})\widehat{\mathbf{H}}(\boldsymbol{R})\Psi_T(\boldsymbol{R})}{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R})\Psi_T(\boldsymbol{R})} = \frac{\langle\Psi_T|\widehat{\mathbf{H}}|\Psi_T\rangle}{\langle\Psi_T|\Psi_T\rangle} \geq E_0. \tag{3.4}$$

The only possible outcome for the energy of an ideal experiment carried out on an assemble of measures are the eigenvalues of the Hamiltonian. Moreover, the corresponding eigenstates contain all the relevant information on the system in relation with this operator. Assuming that these eigenstates are normalized, the trial wave function can be expanded using them as basis as they form a complete set,

$$\Psi_T(\mathbf{R}) = \sum_n c_n\Psi_n(\mathbf{R}),$$

which inserted in Eq. (3.4) yields[2]

$$\begin{aligned}\langle H\rangle &= \frac{\int\left(\sum_n c_n^*\Psi_n^*\right)\widehat{\mathbf{H}}\left(\sum_m c_m\Psi_m\right)d\boldsymbol{R}}{\int\left(\sum_n c_n^*\Psi_n^*\right)\left(\sum_m c_m\Psi_m\right)d\boldsymbol{R}} = \frac{\langle\sum_n c_n\Psi_n|\widehat{\mathbf{H}}|\sum_m c_m\Psi_m\rangle}{\langle\sum_n c_n\Psi_n|\sum_m c_m\Psi_m\rangle} \\ &= \frac{\sum_n\sum_m c_n^*c_m\langle\Psi_n|E_m|\Psi_m\rangle}{\sum_n\sum_m c_n^*c_m\underbrace{\langle\Psi_n|\Psi_m\rangle}_{\delta_{nm}}} = \frac{\sum_n\sum_m c_n^*c_m E_m\overbrace{\langle\Psi_n|\Psi_m\rangle}^{\delta_{nm}}}{\sum_n|c_n|^2} = \frac{\sum_n E_n|c_n|^2}{\sum_n|c_n|^2} \\ &= \sum_n E_n|c_n|^2 = E_0|c_0|^2 + \sum_{n>0}E_n|c_n|^2. \end{aligned} \tag{3.5}$$

Here we have used $\langle\Psi_n|\Psi_m\rangle = \delta_{nm} = \begin{cases}1 & \text{if } n=m\\ 0 & \text{if } n\neq m\end{cases}$ and $\sum_n|c_n|^2 = 1$. We have assumed that the states are normalized. The squared coefficients $|c_n|^2$ define the probability of finding the system in state Ψ_n. Since they are probabilities they sum up to one. Then,

$$|c_0|^2 = 1 - \sum_{n>0}|c_n|^2$$

which substituted in Eq. (3.5) gives rise to $\langle H\rangle = E_0 + \sum_{n>0}(E_n - E_0)|c_n|^2$. Since $E_n \geq E_0\,\forall n \geq 0$ and $|c_n|^2$ is a positive quantity, it follows that

$$\boxed{\langle H\rangle = \langle\Psi_T|\widehat{\mathbf{H}}|\Psi_T\rangle = \sum_n E_n|c_n|^2 \geq E_0.} \tag{3.6}$$

What equation (3.6) implies is that by varying Ψ_T until the expectation value $\langle\widehat{\mathbf{H}}\rangle$ becomes minimized, we can obtain an approximation to the true exact wave function and to the ground-state energy. Equality is reached when the trial wave function equals the true wave function. Therefore, the degree of accuracy in variational calculations rely on making a physically plausible guess at the form of the ground state wavefunction.

3.2.1 Computing the energy

The expectation value of the energy evaluated with the trial wave function can be obtained by

$$E_{VMC} = \langle H\rangle = \frac{\int\Psi_T^*\widehat{\mathbf{H}}\Psi_T d\boldsymbol{R}}{\int\Psi_T^*\Psi_T d\boldsymbol{R}}. \tag{3.7}$$

An alternative way of writing it is

$$E_{VMC} = \frac{\int|\Psi_T|^2\left(\frac{\widehat{\mathbf{H}}\Psi_T}{\Psi_T}\right)d\boldsymbol{R}}{\int|\Psi_T|^2 d\boldsymbol{R}} = \int\frac{|\Psi_T|^2\left(\frac{\widehat{\mathbf{H}}\Psi_T}{\Psi_T}\right)d\boldsymbol{R}}{\int|\Psi_T|^2 d\boldsymbol{R}} = \int\frac{|\Psi_T|^2 d\boldsymbol{R}}{\int|\Psi_T|^2 d\boldsymbol{R}}\left(\frac{\widehat{\mathbf{H}}\Psi_T}{\Psi_T}\right). \tag{3.8}$$

[2]For simplicity in the notation, we have dropped the $\boldsymbol{R}$ dependence of the trial wave function.

From the last equation we can define the so-called local energy[3] operator

$$\boxed{\widehat{\mathbf{E}}_L(\mathbf{R}) = \frac{1}{\Psi_T(\boldsymbol{R})}\widehat{\mathbf{H}}\Psi_T(\boldsymbol{R})} \tag{3.9}$$

and the probrability distribution function (PDF)

$$\boxed{P(\boldsymbol{R}) = \frac{|\Psi_T(\boldsymbol{R})|^2}{\int |\Psi_T(\boldsymbol{R})|^2 \, d\boldsymbol{R}}.} \tag{3.10}$$

All the calculations of expectation values in the quantum variational monte Carlo (QVMC) method are carried out in relation with this PDF[4]. The variational energy can therefore be obtained by

$$\boxed{\langle E \rangle_P = \int P(\boldsymbol{R})\widehat{\mathbf{E}}_L(\boldsymbol{R}) d\boldsymbol{R} \approx \frac{1}{N}\sum_{i=1}^{N} E_L(\boldsymbol{X_i})),} \tag{3.11}$$

where $\boldsymbol{X}$ and N are the set of sample points and the total number of Monte Carlo steps, respectively.

An interpretation of Eq. (3.11) is that it represents the average of the local energy. The error in the local energy can be estimated using the definition of the variance in Eq. (3.3). Noting that the energy squared is given by

$$\boxed{\langle \widehat{\mathbf{E}}_L^2 \rangle = \int P(\mathbf{R})\widehat{\mathbf{E}}_L^2(\mathbf{R}) d\mathbf{R} \approx \frac{1}{N}\sum_{i=1}^{N} E_L^2(x_i),} \tag{3.12}$$

and using it in combination with Eq. (3.11) we get

$$\sigma_{E_L}^2 = \int P(\boldsymbol{R})\langle (E_L - \langle E_L \rangle)^2 \rangle d\boldsymbol{R} = \langle (E_L - \langle E_L \rangle)^2 \rangle = \langle E_L^2 \rangle_P - \langle E_L \rangle_P^2, \tag{3.13}$$

or

$$\boxed{\sigma_{E_L}^2 \approx \left(\frac{1}{N}\sum_{i=1}^{N} E_L^2\right) - \left(\frac{1}{N}\sum_{i=1}^{N} E_L\right)^2,} \tag{3.14}$$

which in turns means that as the trial wave function approaches an exact eigenstate wave function of the Hamiltonian, the local energy approaches the exact energy and the variance becomes equal zero. It is called the zero variance property of the variational Monte Carlo method. In order to get the variational energy of the system, the Metropolis algorithm samples sets of electron positions from the probability distribution function Ψ_T^2 and calculates the local energy for each space configuration.

[3] The local energy term is central both in variational and diffusion monte Carlo. Moreover, it is one of the most consuming execution time operations in quantum monte Carlo [?, ?]. For most hamiltonians, $\widehat{\mathbf{H}}$ is a sum of kinetic energy, involving a second derivative, and a momentum independent potential. The contribution from the potential term is hence just the numerical value of the potential.

[4] When compared with probability distribution function in the diffusion monte Carlo method, the positiveness of the weight function in (3.10) is advantageous when dealing with fermionic systems.

3.3 The quantum variational Monte Carlo (QVMC) method

The quantum variational Monte Carlo method derives from the use of Monte Carlo techniques in combination with the variational principle. It allows sus to calculate quantum expectation values given a trial wave function. In essence, the QVMC method chooses/creates and then samples a probability distribution function (PDF) which is proportional to the square of a trial wave function $\Psi_T^2(\boldsymbol{R})$.

3.3.1 Sampling the trial wave function: the Metropolis algorithm

The Metropolis algorithm constructs space configurations based on the so-called Markov chains[5] by box sampling[6]. The recipe for doing one complete cycle (random walk) of the Markov chain starts with moving a particle from the current position $\boldsymbol{x^{curr}}$ to a new one $\boldsymbol{x^{new}}$ according to

$$\boxed{\boldsymbol{x^{new}} = \boldsymbol{x^{curr}} + \Delta \boldsymbol{x}\xi,} \tag{3.15}$$

where Δx is the step size and ξ is a $3N-$dimensional vector of uniformly distributed random numbers in the range $\xi \in [-1, 1]$.

Later, we perform a so called metropolis test to accept or reject the new position. First, we define the ratio

$$q(\boldsymbol{x^{new}}, \boldsymbol{x^{cur}}) \equiv \frac{P(\boldsymbol{x^{new}})}{P(\boldsymbol{x^{cur}})} = \frac{|\Psi_T(\boldsymbol{x^{new}})|^2}{|\Psi_T(\boldsymbol{x^{cur}})|^2}, \tag{3.16}$$

and compare with a uniformly distributed random number between zero and one. The new position is accepted if and only if this number is less that the ratio above, i.e.,

$$\boxed{A = \min\{1, q(\boldsymbol{x^{new}}, \boldsymbol{x^{cur}})\}.} \tag{3.17}$$

In Eq. (3.15) the next step does not depend on the wave function. This means in turn that our sampling of points may not be very efficient. This leads to the concept of importance sampling.

3.3.2 Improving the sampling: The Fokker-Planck formalism

In the algorithm just described, the sampling does not take into account some properties of the probability distribution. As a consequence some movements are rejected unnecessarily. The Fokker-Planck algorithm, replaces the brute force Metropolis algorithm with a walk in coordinate space biased by the trial wave function. This approach is based on the Fokker-Planck equation and the Langevin equation for generating a trajectory in coordinate space.

The process of isotropic diffusion characterized by a time-dependent probability density $P(\boldsymbol{x}, t)$ obeys the Fokker-Planck equation

$$\frac{\partial P}{\partial t} = \sum_i D \frac{\partial}{\partial \boldsymbol{x_i}} \left(\frac{\partial}{\partial \boldsymbol{x_i}} - \boldsymbol{F_i} \right) P(\boldsymbol{x}, t),$$

[5] A Markov chain is defined as a sequence of independent random variables in which each new configuration is generated with a probability disribution depending on the previous one [?].

[6] It refers to the fact that each particle is moved in a cartesian system in one direction at the time.

where $\boldsymbol{F_i}$ is the i^{th} component of the drift term (drift velocity) caused by an external potential, and D is the diffusion coefficient. The convergence to a stationary probability density as the one in Eq.(3.10) can be obtained by setting the left hand side to zero. The resulting equation will be satisfied if and only if all the terms of the sum are equal zero,

$$\frac{\partial^2 P}{\partial \boldsymbol{x_i}^2} = P\frac{\partial}{\partial \boldsymbol{x_i}}\boldsymbol{F_i} + \boldsymbol{F_i}\frac{\partial}{\partial \boldsymbol{x_i}}P.$$

The drift vector should be of the form $\boldsymbol{F} = g(\boldsymbol{x})\frac{\delta P}{\delta \boldsymbol{x}}$. Then,

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

The condition of stationary density means that the left hand side equals zero. In other words, the terms containing first and second derivatives have to cancel each other. It is possible only if $g = \frac{1}{P}$, which yields

$$\boxed{\boldsymbol{F} = 2\frac{1}{\Psi_T}\nabla\Psi_T,} \tag{3.18}$$

which is known as the so-called quantum force. This term is responsible for pushing the walker towards regions of configuration space where the trial wave function is large, increasing the efficiency of the simulation in contrast to the Metropolis algorithm where the walker has the same probability of moving in every direction, i.e., the Langevin approach provides some kind of importance sampling.

In statistical mechanics the resulting trajectories of the Fokker-Planck equation are generated from the Langevin equation [?]

$$\frac{\partial \mathbf{x}(t)}{\partial t} = D\boldsymbol{F}(\mathbf{x}(t)) + \eta, \tag{3.19}$$

with η being a stochastic force[7] distributed in a Gaussian with mean zero and variance $2D$, where $D = 0.5$ is a diffusion constant. The new positions in coordinate space (a path configuration or assamble of random walkers) are given as

$$\boxed{\boldsymbol{x^{new}} = \boldsymbol{x^{cur}} + D\boldsymbol{F}(\boldsymbol{x^{new}})\Delta t + \eta,} \tag{3.20}$$

where η is a gaussian random variable and Δt is a chosen time step.

3.3.3 A generalized Metropolis algorithm

Considering that the first step in the Metropolis algorithm is taken with a transition probability $T(\boldsymbol{x^{new}} \rightarrow \boldsymbol{x^{cur}})$ and denoting the acceptance/rejection probability for the move being accepted by $A(\boldsymbol{x^{new}} \rightarrow \boldsymbol{x^{cur}})$, the total probability that a walker moves from $\boldsymbol{x^{new}} \rightarrow \boldsymbol{x^{cur}}$ is $T(\boldsymbol{x^{new}} \rightarrow \boldsymbol{x^{cur}})A(\boldsymbol{x^{new}} \rightarrow \boldsymbol{x^{cur}})$. Since we are sampling the probability distribution using a Markov process, at equilibrium (the most likely state) the fraction of walkers making a transition from $\boldsymbol{x^{new}} \rightarrow \boldsymbol{x^{cur}}$ equals the fraction moving from

[7] A stochastic force is a force fluctuating alleatorily.

$\boldsymbol{x^{cur}} \rightarrow \boldsymbol{x^{new}}$. This condition is known as detailed balance and is a sufficient condition to reach steady state. For a given probability distribution function $P(\boldsymbol{x})$, the acceptance probability must satisfy

$$\frac{A(\boldsymbol{x^{cur}} \rightarrow \boldsymbol{x^{new}})}{A(\boldsymbol{x^{new}} \rightarrow \boldsymbol{x^{cur}})} = \frac{P(\boldsymbol{x^{new}})}{P(\boldsymbol{x^{cur}})} \frac{T(\boldsymbol{x^{new}} \rightarrow \boldsymbol{x^{cur}})}{T(\boldsymbol{x^{cur}} \rightarrow \boldsymbol{x^{new}})}.$$

The Fokker-Planck equation yields a transition probability given by the Green's function

$$G(\boldsymbol{x^{new}}, \boldsymbol{x^{cur}}, \Delta t) = \frac{1}{(4\pi D\Delta t)^{3N/2}} \exp\left(-(\boldsymbol{x^{new}} - \boldsymbol{x^{cur}} - D\Delta t F(\boldsymbol{x^{cur}}))^2/4D\Delta t\right), \tag{3.21}$$

which in turn means that the term defined by (3.16) appearing in (3.17) is replaced by

$$\boxed{q(\boldsymbol{x^{new}}, \boldsymbol{x^{old}}) = \frac{G(\boldsymbol{x^{old}}, \boldsymbol{x^{new}}, \Delta t)|\Psi_T(\boldsymbol{x^{new}})|^2}{G(\boldsymbol{x^{new}}, \boldsymbol{x^{old}}, \Delta t)|\Psi_T(\boldsymbol{x^{old}})|^2}.} \tag{3.22}$$

3.3.4 An algorithm for the VMC method

A short pseudocode, as well as a chartflow for the variational Monte Carlo method are shown below.

Algorithm 1 . Quantum Variational Monte Carlo method with drift diffusion.

Require: Number of particles (nel), monte Carlo cycles (nmc), thermalization steps, step size when moving particles, initial space configuration $\boldsymbol{R}$ and a many body N-particle wave function $\Psi_{\boldsymbol{alpha}} = \Psi_\alpha(\boldsymbol{R})$ with variational parameters $\boldsymbol{\alpha}$.

.

Ensure: Estimate the value of the local energy $\langle E_\alpha \rangle$.

Equilibrate first

for $c = 1$ to nmc do
  for $p = 1$ to nel do
    $\boldsymbol{x}_p^{trial} = \boldsymbol{x}_p^{cur} + \chi + D\boldsymbol{F}(\boldsymbol{x}_p^{cur})\delta t$

    Accept trial move as new position with probability

$$min\left[1, \frac{\omega(\boldsymbol{x}^{cur}, \boldsymbol{x}^{new})}{\omega(\boldsymbol{x}^{new}, \boldsymbol{x}^{cur})} \frac{|\Psi(\boldsymbol{x}^{new})|^2}{|\Psi(\boldsymbol{x}^{cur})|^2}\right]$$

  end for
  Compute the local energy, and update all other observables.
end for
Compute the mean energy and standard deviation.

The time-consuming part in the variational Monte Carlo calculation is the evaluation of the kinetic energy term. The potential energy, as long as it has a simple r-dependence adds only a simple term to the local energy operator.

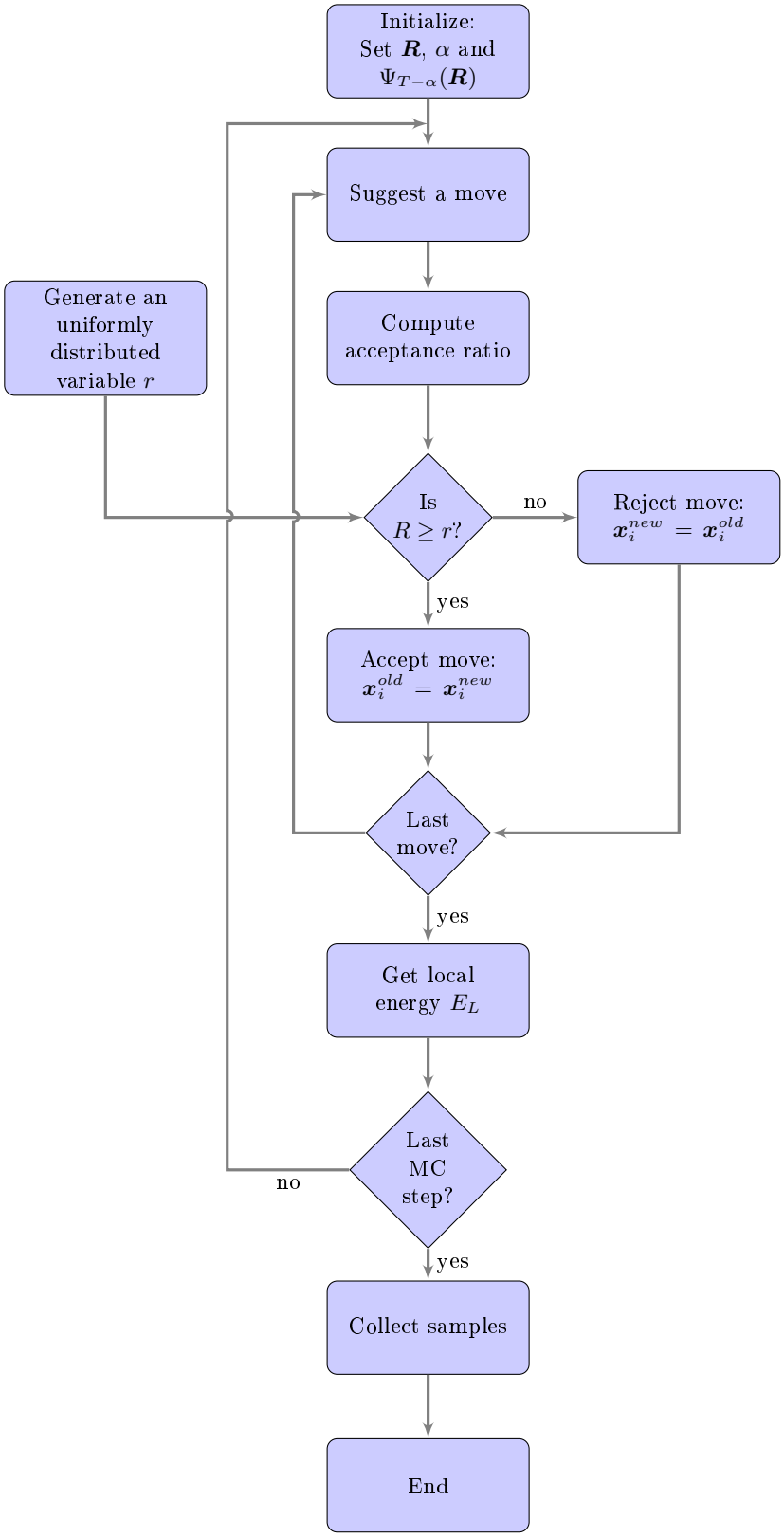

Figure 3.1: Chart flow for the Quantum Varitional Monte Carlo algorithm.

3.4 Trial wave functions

In the QVMC method, all observables are evaluated with respect to the probability distribution given by Eq. (3.10). Therefore, the choice of the trial wave function is critical in order to get sufficiently accurate results in the calculations. Despite its importance, it is, however, a highly non-trivial task. The quality of the results and the cost of obtaining a certain statistical accuracy depends on how well the trial wave function approximates an exact eigenstate, since it improves the importance sampling.

A trial wave function frequently used in the simulation of electronic systems has the form of a product between a Slater determinant and a correlation function. The former takes into account the Pauli exclusion principle, the latter includes the correlation between particles. In general,

$$\boxed{\Psi_T(\boldsymbol{x}) = \Psi_D \Psi_C,} \tag{3.23}$$

where Ψ_D is a Slater determinant[8] and Ψ_C is a correlation function, generally depending on the interparticle distances r_{ij},

$$\Psi_C = \prod_{i<j} g_{ij} = \prod_{i<j}^{N} g(r_{ij}) = \prod_{i=1}^{N} \prod_{j=i+1}^{N} g(r_{ij}), \tag{3.24}$$

where

$$r_{ij} = |\boldsymbol{r}_j - \boldsymbol{r}_i| = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2} \tag{3.25}$$

is the interelectronic scalar distance. In this thesis we use the linear Padé-Jastrow correlation function[9], which has the following form:

$$\boxed{\Psi_{PJ} = \exp\left(\sum_{j<i} \frac{a_{ij} r_{ij}}{1 + \beta_{ij} r_{ij}}\right),} \tag{3.26}$$

where a_{ij} is the cusp factor given by equations (2.27) and (2.27), and β_{ij} is a free parameter which can be varied for optimizing the correlation part of the total trial wave function.

3.5 Optimization of the trial wave function

The optimization of the parameters of the trial wave function is crucial for the success of the QVMC method. For a trial wave function, the local energy is not constant and its spatially averaged variance is a measure of how well the trial wave function approximates an eigenstate [?]. This characteristic is therefore exploited to optimize both the energy and the wave function. The second method, computes the dispersion of the local energy, the variance $\sigma = \int \widehat{\mathbf{H}}\Psi^2 - E_v$. If in each step of a Monte Carlo calculation one gets samples that are uncorrelated with the others, the dispersion is proportional to the variance of the calculation and the variance of the average energy will be σ^2/N, where N is the number

[8] The final form of the Slater determinant depends on the choice of the single-particle wave functions.

[9] The Jastrow term in the trial wave function reduces the possibility of two electrons to get close. Then, it decreases the average value of the repulsive interation providing an energy gain.

of Monte Carlo cycles. The variance is a positive definite quantity with known minimum value of zero. Therefore some authors prefer its minimization over the variational energy [?].

Several methods for finding the minimum of multivariable functions in variational Monte Carlo have been proposed. Some of them are the fixed sampling reweighting for minimization of the variance, with the current state of the art discussed in [?]; stochastic gradient approximation (SGA), designed by Robins and Munro (1951) to handle optimization with noise gradients; gradient biased random walk and Newton's method, among others [?, ?]. The last one was used by Lin et al(2000) [?] in energy minimization along with analytical derivatives of the energy with respect to the variational parameters.

In this thesis we use the quasi-Newton method described in chapter 10.7 of Numerical Recipes [?] to minimize the energy. The same reference provides the function dfpmin, which requires the derivative of the energy with respect to the variational parameters. In this thesis, we encapsulate this function in a Optimizer and provide pointers to the MonteCarlo, Psi and Energy class to implement and running the optimizing algorithm in a convenient way.

3.5.1 The derivative of the energy with respect to its variational parameters

Because the non-relativistic Hamiltonian we are dealing with has inversion symmetry ($\widehat{\mathbf{V}}(\boldsymbol{r}) = \widehat{\mathbf{V}}(-\boldsymbol{r})$), the true ground-state wave function can generally be constructed as a real one [?,?]. The first derivative of Eq. (3.7) with respect to the variational parameters can be written

$$\begin{aligned}\frac{\partial E}{\partial c_m} = & \frac{1}{\int \Psi^2 d\boldsymbol{r}} \left[\int \frac{\partial \Psi}{\partial c_m} \widehat{\mathbf{H}} \Psi d\boldsymbol{r} + \int \Psi \widehat{\mathbf{H}} \frac{\partial \Psi}{\partial c_m} d\boldsymbol{r} \right] \\ & - \frac{1}{(\int \Psi^2 d\boldsymbol{r})^2} \int \Psi \widehat{\mathbf{H}} \Psi d\boldsymbol{r} \int 2\Psi \frac{\partial \Psi}{\partial c_m} d\boldsymbol{r}.\end{aligned}$$

By Hermiticity, $\int \widehat{\mathbf{H}} \Psi \frac{\partial \Psi}{\partial c_m} d\boldsymbol{r} = \int \frac{\partial \Psi}{\partial c_m} \widehat{\mathbf{H}} \Psi d\boldsymbol{r}$, and

$$\begin{aligned}\frac{\partial E}{\partial c_m} = & \frac{2}{\int \Psi^2 d\boldsymbol{r}} \left[\int \Psi^2 \left(\frac{\widehat{\mathbf{H}} \Psi}{\Psi} \right) \left(\frac{\frac{\partial \Psi}{\partial c_m}}{\Psi} \right) d\boldsymbol{r} \right] \\ & - \frac{2}{(\int \Psi^2 d\boldsymbol{r})^2} \int \Psi^2 \left(\frac{\widehat{\mathbf{H}} \Psi}{\Psi} \right) d\boldsymbol{r} \int \Psi^2 \left(\frac{\frac{\partial \Psi}{\partial c_m}}{\Psi} \right) d\boldsymbol{r}\end{aligned}$$

or

$$\frac{\partial E}{\partial c_m} = 2 \left[\left\langle E_L \frac{\frac{\partial \Psi_{T_{c_m}}}{\partial c_m}}{\Psi_{T_{c_m}}} \right\rangle - E \left\langle \frac{\frac{\partial \Psi_{T_{c_m}}}{\partial c_m}}{\Psi_{T_{c_m}}} \right\rangle \right], \tag{3.27}$$

where the average $\langle ... \rangle$ is taken over the whole Metropolis simulation [?].

Most of the computational cost in the variational Monte Carlo method is related to the trial wave function. It is because all the observables and other calculations are based on it. The deveploment of algorithms making such calculations efficient is essential in the context of this thesis. The next chapter is devoted to this topic.

Chapter 4

Computational and parametric optimization of the trial wave function

The trial wave function plays a central role in quantum variational Monte Carlo simulations. Its importance lies in the fact that all the observables are computed with respect to the probability distribution function defined from the trial wave function. Moreover, it is needed in the Metropolis algorithm and in the evaluation of the quantum force term when importance sampling is applied. Computing 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 Nd Slater determinants for the gradient (quantum force) and $N \cdot d$ for the Laplacian (kinetic energy). Therefore, it is imperative to find alternative ways of computating quantities related to the trial wave function such that the computational perfomance can be improved.

4.1 Splitting the Slater determinant

Following Ref. [?], assume that we wish to compute the expectation value of a spin-independent quantum mechanical operator $\widehat{\mathbf{O}}(\boldsymbol{r})$ using the spin-dependent state $\Psi(\boldsymbol{x})$, where $\boldsymbol{x} = (\boldsymbol{r}, \boldsymbol{\sigma})$ represents the space-spin coordinate par. Then,

$$\langle \widehat{\mathbf{O}} \rangle = \frac{\langle \Psi(\boldsymbol{x}) | \widehat{\mathbf{O}}(\boldsymbol{r}) | \Psi(\boldsymbol{x}) \rangle}{\langle \Psi(\boldsymbol{x}) | \Psi(\boldsymbol{x}) \rangle}.$$

If for each spin configuration $\boldsymbol{\sigma} = (\boldsymbol{\sigma_1}, \ldots, \boldsymbol{\sigma_N})$ we replace the total antisymmetric wave function by a version with permuted arguments arranged such that the first $N_\uparrow$ arguments are spin up and the rest $N_\downarrow = N - N_\uparrow$ are spin down we get

$$\begin{aligned} \Psi(\boldsymbol{x_1}, \ldots, \boldsymbol{x_N}) &\rightarrow \Psi(\boldsymbol{x_{i1}}, \ldots, \boldsymbol{x_{iN}}) \\ &= \Psi(\{\boldsymbol{r_{i1}}, \uparrow\}, \ldots, \{\boldsymbol{r_{iN_\uparrow}}, \uparrow\}, \{\boldsymbol{r_{iN_\uparrow+1}}, \downarrow\}, \ldots, \{\boldsymbol{r_{iN}}, \downarrow\}) \\ &= \Psi(\{\boldsymbol{r_1}, \uparrow\}, \ldots, \{\boldsymbol{r_{N_\uparrow}}, \uparrow\}, \{\boldsymbol{r_{1N_\uparrow+1}}, \downarrow\}, \ldots, \{\boldsymbol{r_N}, \downarrow\}). \end{aligned}$$

Because the operator $\widehat{\mathbf{O}}$ is symmetric with respect to the exchange of labels in a pair of particles, each spin configuration gives an identical contribution to the expectation value.

Hence,

$$\langle \widehat{\mathbf{O}} \rangle = \frac{\langle \Psi(\boldsymbol{r}) | \widehat{\mathbf{O}}(\boldsymbol{r}) | \Psi(\boldsymbol{r}) \rangle}{\langle \Psi(\boldsymbol{r}) | \Psi(\boldsymbol{r}) \rangle}$$

The new state is antisymmetric with respect to exchange of spatial coordinates of pairs of spin-up or spin-down electrons. Therefore, for spin-independent Hamiltonians, the Slater determinant can be splitted in a product of Slater determinants obtained from single particle orbitals with different spins. For electronic systems we get then

$$\Psi_D = D_\uparrow D_\downarrow,$$

where

$$D_\uparrow = |\boldsymbol{D}(\boldsymbol{r_1}, \boldsymbol{r_2}, \ldots, \boldsymbol{r_{N/2}})|_\uparrow = \begin{vmatrix} \phi_1(\boldsymbol{r_1}) & \phi_2(\boldsymbol{r_1}) & \cdots & \phi_{N/2}(\boldsymbol{r_1}) \\ \phi_1(\boldsymbol{r_2}) & \phi_2(\boldsymbol{r_2}) & \cdots & \phi_{N/2}(\boldsymbol{r_2}) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\boldsymbol{r_{N/2}}) & \phi_2(\boldsymbol{r_{N/2}}) & \cdots & \phi_{N/2}(\boldsymbol{r_{N/2}}) \end{vmatrix}_\uparrow . \tag{4.1}$$

In a similar way, $D_\downarrow = |\boldsymbol{D}(\boldsymbol{r_{N/2+1}}, \boldsymbol{r_{N/2+2}}, \ldots, \boldsymbol{r_N})|_\downarrow$. The normalization factor has been removed, since it cancels in the ratios needed by the variational Monte Carlo algorithm, as shown later. The new state $\Psi_D(\boldsymbol{r})$ gives in this case the same expectation value as $\Psi(\boldsymbol{x})$, but is more convenient in terms of computational cost. Observe that the Slater determinant in Eq. (3.23) can now be factorized as

$$\boxed{\Psi_T(\boldsymbol{x}) = D_\uparrow D_\downarrow \Psi_C.} \tag{4.2}$$

4.2 Computational optimization of the Metropolis/hasting ratio

In the Metropolis/Hasting algorithm, the acceptance ratio determines the probability for a particle to be accepted at a new position. The ratio of the trial wave functions evaluated at the new and current positions is given by

$$\boxed{R \equiv \frac{\Psi_T^{new}}{\Psi_T^{cur}} = \underbrace{\frac{|\boldsymbol{D}|_\uparrow^{new}}{|\boldsymbol{D}|_\uparrow^{cur}} \frac{|\boldsymbol{D}|_\downarrow^{new}}{|\boldsymbol{D}|_\downarrow^{cur}}}_{R_{SD}} \underbrace{\frac{\Psi_C^{new}}{\Psi_C^{cur}}}_{R_C}.} \tag{4.3}$$

4.2.1 Evaluating the determinant-determinant ratio

Evaluating the determinant of a $N \times N$ matrix by Gaussian elimination takes of the order of $\mathcal{O}(N^3)$ operations, which is rather expensive for a many-particle quantum system. An alternative algorithm not requiring the separated evaluation of the determinants will be derived in the following. We start by defining a Slater matrix $\boldsymbol{D}$ with its corresponding $(i, j)-$entries given by

$$D_{ij} \equiv \phi_j(\boldsymbol{r_i}), \tag{4.4}$$

where $\phi_j(\boldsymbol{r_i})$ is the j^{th} single particle wave function evaluated for the particle at position $\boldsymbol{r_i}$.

The inverse of a (Slater) matrix is related to its adjoint (transpose matrix of cofactors) and its determinant by

$$\boldsymbol{D^{-1}} = \frac{adj\boldsymbol{D}}{|\boldsymbol{D}|} \Rightarrow |\boldsymbol{D}| = \frac{adj\boldsymbol{D}}{\boldsymbol{D^{-1}}}, \tag{4.5}$$

or

$$|\boldsymbol{D}| = \sum_{j=1}^{N} \frac{C_{ji}}{D_{ij}^{-1}} = \sum_{j=1}^{N} D_{ij}C_{ji}, \tag{4.6}$$

i.e., the determinant of a matrix equals the scalar product of any column(row) of the matrix with the same column(row) of the matrix of cofactors.

In the particular case when only one particle is moved at the time (say particle at position $\boldsymbol{r_i}$), this changes only one row (or column)[1] of the Slater matrix. An efficient way of evaluating that ratio is as follows [?, ?].

We define the ratio of the new to the old determinants in terms of Eq. (4.6) such that

$$R_{SD} \equiv \frac{|\boldsymbol{D}(\boldsymbol{x^{new}})|}{|\boldsymbol{D}(\boldsymbol{x^{cur}})|} = \frac{\sum_{j=1}^{N} D_{ij}(\boldsymbol{x^{new}})C_{ji}(\boldsymbol{x^{new}})}{\sum_{j=1}^{N} D_{ij}(\boldsymbol{x^{cur}})C_{ji}(\boldsymbol{x^{cur}})}.$$

When the particle at position $\boldsymbol{r_i}$ is moved, the $i^{th}-$row of the matrix of cofactors remains unchanged, i.e., the row number i of the cofactor matrix are independent of the entries in the rows of its corresponding matrix $\boldsymbol{D}$. Therefore,

$$C_{ij}(\boldsymbol{x^{new}}) = C_{ij}(\boldsymbol{x^{cur}}),$$

and

$$R_{SD} = \frac{\sum_{j=1}^{N} D_{ij}(\boldsymbol{x^{new}})C_{ji}(\boldsymbol{x^{cur}})}{\sum_{j=1}^{N} D_{ij}(\boldsymbol{x^{cur}})C_{ji}(\boldsymbol{x^{cur}})} = \frac{\sum_{j=1}^{N} D_{ij}(\boldsymbol{x^{new}})D_{ji}^{-1}(\boldsymbol{x^{cur}})|\boldsymbol{D}|(\boldsymbol{x^{cur}})}{\sum_{j=1}^{N} D_{ij}(\boldsymbol{x^{cur}})D_{ji}^{-1}(\boldsymbol{x^{cur}})|\boldsymbol{D}|(\boldsymbol{x^{cur}})}. \tag{4.7}$$

The invertibility of $\boldsymbol{D}$ implies that

$$\sum_{k}^{N} D_{ik}D_{kj}^{-1} = \delta_{ij}. \tag{4.8}$$

Hence, the denominator in Eq. (4.7) is equal to unity. Then,

$$R_{SD} = \sum_{j=1}^{N} D_{ij}(\boldsymbol{x^{new}})D_{ji}^{-1}(\boldsymbol{x^{cur}}).$$

Substituting Eq. (4.4) we arrive at

$$\boxed{R_{SD} = \sum_{j=1}^{N} \phi_j(\boldsymbol{x_i^{new}})D_{ji}^{-1}(\boldsymbol{x^{cur}})} \tag{4.9}$$

[1] Some authors prefer to express the Slater matrix by placing the orbitals in a row wise order and the position of the particles in a column wise one.

which means that determining R_{SD} when only particle i has been moved, requires only the evaluation of the dot product between a vector containing orbitals (evaluated at the new position) and all the entries in the i^{th} column of the inverse Slater matrix (evaluated at the current position). This requires approximately $\mathcal{O}(N)$ operations.

Further optimizations can be done by noting that when only one particle is moved at the time, one of the two determinants in the numerator and denominator of Eq. (4.3) is unaffected, cancelling each other. This allows us to carry out calculations with only half of the total number of particles every time a move occurs, requiring only $(N/2)^d$ operations, where d is the number of spatial components of the problem, in systems with equal number of electrons with spin up and down. The total number of operations for a problem in three dimensions becomes $(N/2)^3 = N^3/8$, i.e., the total calculations are reduced up to by a factor of eight.

4.3 Optimizing the $\nabla\Psi_T/\Psi_T$ ratio

Equation (3.18) defines the quantum force required by the Metropolis algorithm with importance sampling. Setting $\Psi_D = |\boldsymbol{D}|_\uparrow|\boldsymbol{D}|_\downarrow$ in Eq. (4.2) we get,

$$\begin{aligned}\frac{\boldsymbol{\nabla}\Psi}{\Psi} &= \frac{\boldsymbol{\nabla}(\Psi_D\,\Psi_C)}{\Psi_D\,\Psi_C} = \frac{\Psi_C\boldsymbol{\nabla}\Psi_D + \Psi_D\boldsymbol{\nabla}\Psi_C}{\Psi_D\Psi_C} = \frac{\boldsymbol{\nabla}\Psi_D}{\Psi_D} + \frac{\boldsymbol{\nabla}\Psi_C}{\Psi_C} \\ &= \frac{\boldsymbol{\nabla}(|\boldsymbol{D}|_\uparrow|\boldsymbol{D}|_\downarrow)}{|\boldsymbol{D}|_\uparrow|\boldsymbol{D}|_\downarrow} + \frac{\boldsymbol{\nabla}\Psi_C}{\Psi_C},\end{aligned}$$

or

$$\boxed{\frac{\boldsymbol{\nabla}\Psi}{\Psi} = \frac{\boldsymbol{\nabla}(|\boldsymbol{D}|_\uparrow)}{|\boldsymbol{D}|_\uparrow} + \frac{\boldsymbol{\nabla}(|\boldsymbol{D}|_\downarrow)}{|\boldsymbol{D}|_\downarrow} + \frac{\boldsymbol{\nabla}\Psi_C}{\Psi_C}.} \tag{4.10}$$

4.3.1 Evaluating the gradient-determinant-to-determinant ratio

The evaluation of Eq. (4.10) requires differentiating the N entries of the Slater matrix with respect to all the d spatial components. Since the evaluation of the Slater determinant scales as $\mathcal{O}(N^3)$ this would involve of the order of $N \cdot d \cdot \mathcal{O}(N^3) \approx \mathcal{O}(N^4)$ floating point operations. A cheaper algorithm can be derived by noting that when only one particle is moved at the time, only one row in the Slater matrix needs to be evaluated again. Thus, only the derivatives of that row with respect to the coordinates of the particle moved need to be updated. Obtaining the gradient-determinant ratio required in Eq. (4.10) becomes straigforward. It is analogous to the procedure used in deriving Eq. (4.3). From Eq. (4.9) and Eq. (4.3) we see that

$$\boxed{\frac{\boldsymbol{\nabla_i}|\boldsymbol{D}(\boldsymbol{x})|}{|\boldsymbol{D}(\boldsymbol{x})|} = \sum_{j=1}^{N}\boldsymbol{\nabla_i}D_{ij}(\boldsymbol{x})D^{-1}_{ji}(\boldsymbol{x}) = \sum_{j=1}^{N}\boldsymbol{\nabla_i}\phi_j(\boldsymbol{x_i})D^{-1}_{ji}(\boldsymbol{x}),} \tag{4.11}$$

which means that when one particle is moved at the time, the gradient-determinant ratio is given by the dot product between the gradient of the single wave functions evaluated for the particle at position $\boldsymbol{r_i}$ and the inverse Slater matrix. A small modification has to be

done when computing the gradient to determinant ratio after a move has been accepted. Denoting by $\boldsymbol{y}$ the vector containing the new spatial coordinates, by definition we get,

$$\frac{\boldsymbol{\nabla_i}|\boldsymbol{D}(\boldsymbol{y})|}{|\boldsymbol{D}(\boldsymbol{y})|} = \sum_{j=1}^{N} \boldsymbol{\nabla_i} D_{ij}(\boldsymbol{y}) D^{-1}_{ji}(\boldsymbol{y}) = \sum_{j=1}^{N} \boldsymbol{\nabla_i}\phi_j(\boldsymbol{y_i}) D^{-1}_{ji}(\boldsymbol{y}),$$

which can be expressed in terms of the transpose inverse of the Slater matrix evaluated at the old positions [?] to get

$$\boxed{\frac{\boldsymbol{\nabla_i}|\boldsymbol{D}(\boldsymbol{y})|}{|\boldsymbol{D}(\boldsymbol{y})|} = \frac{1}{R}\sum_{j=1}^{N} \boldsymbol{\nabla_i}\phi_j(\boldsymbol{y_i}) D^{-1}_{ji}(\boldsymbol{x}).} \tag{4.12}$$

Computing a single derivative is an $\mathcal{O}(N)$ operation. Since there are dN derivatives, the total time scaling becomes $\mathcal{O}(dN^2)$.

4.4 Optimizing the $\nabla^2\Psi_T/\Psi_T$ ratio

From the single-particle kinetic energy operator Eq. (2.5), the expectation value of the kinetic energy expressed in atomic units for electron i is

$$\langle \widehat{\mathbf{K}}_i \rangle = -\frac{1}{2}\frac{\langle \Psi|\nabla_i^2|\Psi\rangle}{\langle \Psi|\Psi\rangle}, \tag{4.13}$$

which is obtained by using Monte Carlo integration. The energy of each space configuration is cummulated after each Monte Carlo cycle. For each electron we evaluate

$$K_i = -\frac{1}{2}\frac{\nabla_i^2\Psi}{\Psi}. \tag{4.14}$$

Following a procedure similar to that of section 4.3, the term for the kinetic energy is obtained by

$$\begin{aligned}\frac{\nabla^2\Psi}{\Psi} &= \frac{\nabla^2(\Psi_D\,\Psi_C)}{\Psi_D\,\Psi_C} = \frac{\boldsymbol{\nabla}\cdot[\boldsymbol{\nabla}(\Psi_D\,\Psi_C)]}{\Psi_D\,\Psi_C} = \frac{\boldsymbol{\nabla}\cdot[\Psi_C\boldsymbol{\nabla}\Psi_D + \Psi_D\boldsymbol{\nabla}\Psi_C]}{\Psi_D\,\Psi_C} \\ &= \frac{\boldsymbol{\nabla}\Psi_C\cdot\boldsymbol{\nabla}\Psi_D + \Psi_C\nabla^2\Psi_D + \boldsymbol{\nabla}\Psi_D\cdot\boldsymbol{\nabla}\Psi_C + \Psi_D\nabla^2\Psi_C}{\Psi_D\,\Psi_C}\end{aligned} \tag{4.15}$$

$$\begin{aligned}\frac{\nabla^2\Psi}{\Psi} &= \frac{\nabla^2\Psi_D}{\Psi_D} + \frac{\nabla^2\Psi_C}{\Psi_C} + 2\frac{\boldsymbol{\nabla}\Psi_D}{\Psi_D}\cdot\frac{\boldsymbol{\nabla}\Psi_C}{\Psi_C} \\ &= \frac{\nabla^2(|\boldsymbol{D}|_\uparrow|\boldsymbol{D}|_\downarrow)}{(|\boldsymbol{D}|_\uparrow|\boldsymbol{D}|_\downarrow)} + \frac{\nabla^2\Psi_C}{\Psi_C} + 2\frac{\boldsymbol{\nabla}(|\boldsymbol{D}|_\uparrow|\boldsymbol{D}|_\downarrow)}{(|\boldsymbol{D}|_\uparrow|\boldsymbol{D}|_\downarrow)}\cdot\frac{\boldsymbol{\nabla}\Psi_C}{\Psi_C},\end{aligned}$$

or

$$\boxed{\frac{\nabla^2\Psi}{\Psi} = \frac{\nabla^2|\boldsymbol{D}|_\uparrow}{|\boldsymbol{D}|_\uparrow} + \frac{\nabla^2|\boldsymbol{D}|_\downarrow}{|\boldsymbol{D}|_\downarrow} + \frac{\nabla^2\Psi_C}{\Psi_C} + 2\left[\frac{\boldsymbol{\nabla}|\boldsymbol{D}|_\uparrow}{|\boldsymbol{D}|_\uparrow} + \frac{\boldsymbol{\nabla}|\boldsymbol{D}|_\downarrow}{|\boldsymbol{D}|_\downarrow}\right]\cdot\frac{\boldsymbol{\nabla}\Psi_C}{\Psi_C},} \tag{4.16}$$

Operation	No optimization	With optimization
Evaluation of R	$\mathcal{O}(N^2)$	$\mathcal{O}\left(\frac{N^2}{2}\right)$
Updating inverse	$\mathcal{O}(N^3)$	$\mathcal{O}\left(\frac{N^3}{4}\right)$
Transition of one particle	$\mathcal{O}(N^2)+\mathcal{O}(N^3)$	$\mathcal{O}\left(\frac{N^2}{2}\right)+\mathcal{O}\left(\frac{N^3}{4}\right)$

Table 4.1: Comparison of the computational cost involved in the computation of the Slater determinant with and without optimization.

where the laplace-determinant-to-determinant ratio is given by

$$\boxed{\frac{\nabla_i^2|\boldsymbol{D}(\boldsymbol{x})|}{|\boldsymbol{D}(\boldsymbol{x})|}=\sum_{j=1}^{N}\nabla_i^2 D_{ij}(\boldsymbol{x})D_{ji}^{-1}(\boldsymbol{x})=\sum_{j=1}^{N}\nabla_i^2\phi_j(\boldsymbol{x_i})D_{ji}^{-1}(\boldsymbol{x})} \tag{4.17}$$

for particle at $\boldsymbol{x_i}$ as deduced from Eq. (4.9) and Eq. (4.3). The comments given in section 4.3 on performance yields applies also to this case. Moreover, Eq. (4.17) is computed with the trial move only if it is accepted.

4.5 Updating the inverse of the Slater matrix

Computing the ratios in Eqs. (4.9), (4.11), (4.12) and (4.17) requires that we maintain the inverse of the Slater matrix evaluated at the current position. Each time a trial position is accepted, the row number i of the Slater matrix changes and updating its inverse has to be carried out. Getting the inverse of an $N\times N$ matrix by Gaussian elimination has a complexity of order of $\mathcal{O}(N^3)$ operations, a luxury that we cannot afford for each time a particle move is accepted. An alternative way of updating the inverse of a matrix when only a row/column is changed was suggested by Sherman and Morris[2] (1951) [?]. It has a time scaling of the order of $\mathcal{O}(N^2)$ [?, ?, ?, ?] and is given by

$$\boxed{D_{kj}^{-1}(\boldsymbol{x^{new}})=\begin{cases} D_{kj}^{-1}(\boldsymbol{x^{cur}})-\frac{D_{ki}^{-1}(\boldsymbol{x^{cur}})}{R}\sum_{l=1}^{N}D_{il}(\boldsymbol{x^{new}})D_{lj}^{-1}(\boldsymbol{x^{cur}}) & \text{if } j\neq i \\ \frac{D_{ki}^{-1}(\boldsymbol{x^{cur}})}{R}\sum_{l=1}^{N}D_{il}(\boldsymbol{x^{cur}})D_{lj}^{-1}(\boldsymbol{x^{cur}}) & \text{if } j=i \end{cases}} \tag{4.18}$$

The evaluation of the determinant of an $N\times N$ matrix by standard Gaussian elimination requires $\mathcal{O}(N^3)$ calculations. As there are Nd independent coordinates we need to evaluate Nd Slater determinants for the gradient (quantum force) and Nd for the Laplacian (kinetic energy). With the updating algorithm we need only to invert the Slater determinant matrix once. This can be done by standard LU decomposition methods.

Table 4.1 summarizes the computational cost associated with the Slater determinant part of the trial wave function.

[2] A derivation can be found in appendix A

4.6 Reducing the computational cost fo the correlation form

From Eq. (3.24), the total number of different relative distances r_{ij} is $N(N-1)/2$. In a matrix storage format, the set forms a strictly upper triangular matrix[3]

$$\boldsymbol{r} \equiv \begin{pmatrix} 0 & r_{1,2} & r_{1,3} & \cdots & r_{1,N} \\ \vdots & 0 & r_{2,3} & \cdots & r_{2,N} \\ \vdots & \vdots & 0 & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & r_{N-1,N} \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix}. \tag{4.19}$$

This applies to $\boldsymbol{g} = \boldsymbol{g}(r_{ij})$ as well.

4.7 Computing the correlation-to-correlation ratio

For the case where all particles are moved simlutaneously, all the g_{ij} have to be reevaluated. The number of operations for getting R_C scales as $\mathcal{O}(N^2)$ [?]. When moving only one particle at a time, say the kth, only $N-1$ of the distances r_{ij} having k as one of their indices are changed. It means that the rest of the factors in the numerator of the Jastrow ratio has a similar counterpart in the denominator and cancel each other. Therefore, only $N-1$ factors of Ψ_C^{new} and Ψ_C^{cur} avoid cancellation and

$$\boxed{R_C = \frac{\Psi_C^{\text{new}}}{\Psi_C^{\text{cur}}} = \prod_{i=1}^{k-1} \frac{g_{ik}^{\text{new}}}{g_{ik}^{\text{cur}}} \prod_{i=k+1}^{N} \frac{g_{ki}^{\text{new}}}{g_{ki}^{\text{cur}}}}. \tag{4.20}$$

For the Padé-Jastrow form

$$\boxed{R_C = \frac{\Psi_C^{\text{new}}}{\Psi_C^{\text{cur}}} = \frac{e^{U_{new}}}{e^{U_{cur}}} = e^{\Delta U},} \tag{4.21}$$

where

$$\Delta U = \sum_{i=1}^{k-1} \left(f_{ik}^{\text{new}} - f_{ik}^{\text{cur}}\right) + \sum_{i=k+1}^{N} \left(f_{ki}^{\text{new}} - f_{ki}^{\text{cur}}\right) \tag{4.22}$$

One needs to develop a special algorithm that iterates only through the elements of the upper triangular matrix $\boldsymbol{g}$ that have k as an index.

4.8 Evaluating the $\boldsymbol{\nabla}\Psi_C/\Psi_C$ ratio

The expression to be derived in the following is of interest when computing the quantum force and the kinetic energy. It has the form

$$\frac{\boldsymbol{\nabla_i}\Psi_C}{\Psi_C} = \frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_i},$$

[3] In the implementation, however, we do not store the entries lying on the diagonal.

for all dimensions and with i running over all particles. From the discussion in section 4.7, for the first derivative only $N-1$ terms survive the ratio because the g-terms that are not differentiated cancel with their corresponding ones in the denominator. Then,

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{1}{g_{ik}}\frac{\partial g_{ik}}{\partial x_k} + \sum_{i=k+1}^{N}\frac{1}{g_{ki}}\frac{\partial g_{ki}}{\partial x_k}. \tag{4.23}$$

An equivalent equation is obtained for the exponential form after replacing g_{ij} by $\exp(g_{ij})$, yielding:

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} + \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_k}, \tag{4.24}$$

with both expressions scaling as $\mathcal{O}(N)$.

Later, using the identity

$$\frac{\partial}{\partial x_i}g_{ij} = -\frac{\partial}{\partial x_j}g_{ij} \tag{4.25}$$

on the right hand side terms of Eq. (4.23) and Eq. (4.24), we get expressions where all the derivatives act on the particle are represented by the second index of g:

$$\boxed{\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{1}{g_{ik}}\frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N}\frac{1}{g_{ki}}\frac{\partial g_{ki}}{\partial x_i},} \tag{4.26}$$

and for the exponential case:

$$\boxed{\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_i}.} \tag{4.27}$$

4.8.1 Special case: correlation functions depending on the scalar relative distances

For correlation forms depending only on the scalar distances r_{ij} defined by Eq. (3.25), a trick introduced in [?] consists in using the chain rule. Noting that

$$\frac{\partial g_{ij}}{\partial x_j} = \frac{\partial g_{ij}}{\partial r_{ij}}\frac{\partial r_{ij}}{\partial x_j} = \frac{x_j - x_i}{r_{ij}}\frac{\partial g_{ij}}{\partial r_{ij}}, \tag{4.28}$$

after substitution in Eq. (4.26) and Eq. (4.27) we arrive at

$$\boxed{\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{1}{g_{ik}}\frac{\boldsymbol{r_{ik}}}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}} - \sum_{i=k+1}^{N}\frac{1}{g_{ki}}\frac{\boldsymbol{r_{ki}}}{r_{ki}}\frac{\partial g_{ki}}{\partial r_{ki}}.} \tag{4.29}$$

Note that for the Padé-Jastrow form we can set $g_{ij} \equiv g(r_{ij}) = e^{f(r_{ij})} = e^{f_{ij}}$ and

$$\frac{\partial g_{ij}}{\partial r_{ij}} = g_{ij}\frac{\partial f_{ij}}{\partial r_{ij}}. \tag{4.30}$$

Therefore,

$$\boxed{\frac{1}{\Psi_{PJ}}\frac{\partial\Psi_{PJ}}{\partial x_k} = \sum_{i=1}^{k-1}\frac{\boldsymbol{r_{ik}}}{r_{ik}}\frac{\partial f_{ik}}{\partial r_{ik}} - \sum_{i=k+1}^{N}\frac{\boldsymbol{r_{ki}}}{r_{ki}}\frac{\partial f_{ki}}{\partial r_{ki}},} \tag{4.31}$$

where

$$\boldsymbol{r}_{ij} = |\boldsymbol{r}_j - \boldsymbol{r}_i| = (x_j - x_i)\hat{\boldsymbol{e}}_1 + (y_j - y_i)\hat{\boldsymbol{e}}_2 + (z_j - z_i)\hat{\boldsymbol{e}}_3 \tag{4.32}$$

is the vectorial distance. When the correlation function is the linear Padé-Jastrow given by Eq. (3.26), we set

$$f_{ij} = \frac{a_{ij}r_{ij}}{(1+\beta_{ij}r_{ij})}, \tag{4.33}$$

which yields the analytical expression

$$\boxed{\frac{\partial f_{ij}}{\partial r_{ij}} = \frac{a_{ij}}{(1+\beta_{ij}r_{ij})^2}}. \tag{4.34}$$

4.9 Computing the $\nabla^2\Psi_C/\Psi_C$ ratio

For deriving this expression we note first that Eq. (4.29) can be written as

$$\boldsymbol{\nabla}_k\Psi_C = \sum_{i=1}^{k-1}\frac{1}{g_{ik}}\boldsymbol{\nabla}_k g_{ik} + \sum_{i=k+1}^{N}\frac{1}{g_{ki}}\boldsymbol{\nabla}_k g_{ki}.$$

After multiplying by Ψ_C and taking the gradient on both sides we get,

$$\begin{aligned}\nabla_k^2\Psi_C &= \boldsymbol{\nabla}_k\Psi_C\cdot\left(\sum_{i=1}^{k-1}\frac{1}{g_{ik}}\boldsymbol{\nabla}_k g_{ik} + \sum_{i=k+1}^{N}\frac{1}{g_{ki}}\boldsymbol{\nabla}_k g_{ki}\right)\\ &+ \Psi_C\nabla_k\cdot\left(\sum_{i=k+1}^{N}\frac{1}{g_{ki}}\boldsymbol{\nabla}_k g_{ki} + \sum_{i=k+1}^{N}\frac{1}{g_{ki}}\boldsymbol{\nabla}_k g_{ki}\right)\\ &= \Psi_C\left(\frac{\boldsymbol{\nabla}_k\Psi_C}{\Psi_C}\right)^2 + \Psi_C\nabla_k\cdot\left(\sum_{i=k+1}^{N}\frac{1}{g_{ki}}\boldsymbol{\nabla}_k g_{ki} + \sum_{i=k+1}^{N}\frac{1}{g_{ki}}\boldsymbol{\nabla}_k g_{ki}\right).\end{aligned} \tag{4.35}$$

Now,

$$\begin{aligned}
\boldsymbol{\nabla}_k \cdot \left(\frac{1}{g_{ik}}\boldsymbol{\nabla}_k g_{ik}\right) &= \boldsymbol{\nabla}_k \left(\frac{1}{g_{ik}}\right) \cdot \boldsymbol{\nabla}_k g_{ik} + \frac{1}{g_{ik}}\boldsymbol{\nabla}_k \cdot \boldsymbol{\nabla}_k g_{ik} \\
&= -\frac{1}{g_{ik}^2}\boldsymbol{\nabla}_k g_{ik} \cdot \boldsymbol{\nabla}_k g_{ik} + \frac{1}{g_{ik}}\boldsymbol{\nabla}_k \cdot \left(\frac{\boldsymbol{r}_{ik}}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right) \\
&= -\frac{1}{g_{ik}^2}(\boldsymbol{\nabla}_k g_{ik})^2 \\
&+ \frac{1}{g_{ik}}\left[\boldsymbol{\nabla}_k \left(\frac{1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right) \cdot \boldsymbol{r}_{ik} + \left(\frac{1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right)\boldsymbol{\nabla}_k \cdot \boldsymbol{r}_{ik}\right] \\
&= -\frac{1}{g_{ik}^2}\left(\frac{\boldsymbol{r}_{ik}}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right)^2 \\
&+ \frac{1}{g_{ik}}\left[\boldsymbol{\nabla}_k \left(\frac{1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right) \cdot \boldsymbol{r}_{ik} + \left(\frac{1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right) d\right] \\
&= -\frac{1}{g_{ik}^2}\left(\frac{\partial g_{ik}}{\partial r_{ik}}\right)^2 \\
&+ \frac{1}{g_{ik}}\left[\boldsymbol{\nabla}_k \left(\frac{1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right) \cdot \boldsymbol{r}_{ik} + \left(\frac{1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right) d\right], \qquad (4.36)
\end{aligned}$$

with d being the number of spatial dimensions.

Moreover,

$$\begin{aligned}
\boldsymbol{\nabla}_k \left(\frac{1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right) &= \frac{\boldsymbol{r}_{ik}}{r_{ik}}\frac{\partial}{\partial r_{ik}}\left(\frac{1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right) \\
&= \frac{\boldsymbol{r}_{ik}}{r_{ik}}\left(-\frac{1}{r_{ik}^2}\frac{\partial g_{ik}}{\partial r_{ik}} + \frac{1}{r_{ik}}\frac{\partial^2 g_{ik}}{\partial r_{ik}^2}\right).
\end{aligned}$$

The substitution of the last result in Eq. (4.36) gives

$$\boldsymbol{\nabla}_k \cdot \left(\frac{1}{g_{ik}}\boldsymbol{\nabla}_k g_{ik}\right) = -\frac{1}{g_{ik}^2}\left(\frac{\partial g_{ik}}{\partial r_{ik}}\right)^2 + \frac{1}{g_{ik}}\left[\left(\frac{d-1}{r_{ik}}\right)\frac{\partial g_{ik}}{\partial r_{ik}} + \frac{\partial^2 g_{ik}}{\partial r_{ik}^2}\right].$$

Inserting the last expression in Eq. (4.35) and after division by Ψ_C we get,

$$\begin{aligned}
\frac{\nabla_k^2 \Psi_C}{\Psi_C} &= \left(\frac{\boldsymbol{\nabla}_k \Psi_C}{\Psi_C}\right)^2 \\
&+ \sum_{i=1}^{k-1} -\frac{1}{g_{ik}^2}\left(\frac{\partial g_{ik}}{\partial r_{ik}}\right)^2 + \frac{1}{g_{ik}}\left[\left(\frac{d-1}{r_{ik}}\right)\frac{\partial g_{ik}}{\partial r_{ik}} + \frac{\partial^2 g_{ik}}{\partial r_{ik}^2}\right] \\
&+ \sum_{i=k+1}^{N} -\frac{1}{g_{ki}^2}\left(\frac{\partial g_{ki}}{\partial r_{ki}}\right)^2 + \frac{1}{g_{ki}}\left[\left(\frac{d-1}{r_{ki}}\right)\frac{\partial g_{ki}}{\partial r_{ki}} + \frac{\partial^2 g_{ki}}{\partial r_{ki}^2}\right]. \qquad (4.37)
\end{aligned}$$

For the exponential case we have

$$\begin{aligned}\frac{\nabla_k^2\Psi_{PJ}}{\Psi_{PJ}} &= \left(\frac{\boldsymbol{\nabla}_k\Psi_{PJ}}{\Psi_{PJ}}\right)^2 \\ &+ \sum_{i=1}^{k-1} -\frac{1}{g_{ik}^2}\left(g_{ik}\frac{\partial f_{ik}}{\partial r_{ik}}\right)^2 + \frac{1}{g_{ik}}\left[\left(\frac{d-1}{r_{ik}}\right)g_{ik}\frac{\partial f_{ik}}{\partial r_{ik}} + \frac{\partial}{\partial r_{ik}}\left(g_{ik}\frac{\partial f_{ik}}{\partial r_{ik}}\right)\right] \\ &+ \sum_{i=k+1}^{N} -\frac{1}{g_{ki}^2}\left(g_{ik}\frac{\partial f_{ki}}{\partial r_{ki}}\right)^2 + \frac{1}{g_{ki}}\left[\left(\frac{d-1}{r_{ki}}\right)g_{ki}\frac{\partial f_{ki}}{\partial r_{ki}} + \frac{\partial}{\partial r_{ki}}\left(g_{ki}\frac{\partial f_{ki}}{\partial r_{ki}}\right)\right].\end{aligned}$$

Using

$$\begin{aligned}\frac{\partial}{\partial r_{ik}}\left(g_{ik}\frac{\partial f_{ik}}{\partial r_{ik}}\right) &= \frac{\partial g_{ik}}{\partial r_{ik}}\frac{\partial f_{ik}}{\partial r_{ik}} + g_{ik}\frac{\partial^2 f_{ik}}{\partial r_{ik}^2} \\ &= g_{ik}\frac{\partial f_{ik}}{\partial r_{ik}}\frac{\partial f_{ik}}{\partial r_{ik}} + g_{ik}\frac{\partial^2 f_{ik}}{\partial r_{ik}^2} \\ &= g_{ik}\left(\frac{\partial f_{ik}}{\partial r_{ik}}\right)^2 + g_{ik}\frac{\partial^2 f_{ik}}{\partial r_{ik}^2}\end{aligned}$$

and substituting this result into the equation above gives rise to the final expression,

$$\begin{aligned}\frac{\nabla_k^2\Psi_{PJ}}{\Psi_{PJ}} &= \left(\frac{\boldsymbol{\nabla}_k\Psi_{PJ}}{\Psi_{PJ}}\right)^2 \\ &+ \sum_{i=1}^{k-1}\left[\left(\frac{d-1}{r_{ik}}\right)\frac{\partial f_{ik}}{\partial r_{ik}} + \frac{\partial^2 f_{ik}}{\partial r_{ik}^2}\right] + \sum_{i=k+1}^{N}\left[\left(\frac{d-1}{r_{ki}}\right)\frac{\partial f_{ki}}{\partial r_{ki}} + \frac{\partial^2 f_{ki}}{\partial r_{ki}^2}\right].\end{aligned} \tag{4.38}$$

Again, for the linear Padé-Jastrow of Eq. (3.26), we get in this case the analytical result

$$\boxed{\frac{\partial^2 f_{ij}}{\partial r_{ij}^2} = -\frac{2a_{ij}\beta_{ij}}{(1+\beta_{ij}r_{ij})^3}}. \tag{4.39}$$

4.10 Efficient parametric optimization of the trial wave function

Energy minimization as expressed by Eq. (3.27) requires the evaluation of the derivative of the trial wave function with respect to the variational parameters. The computational cost of this operation depends, of course, on the algorithm selected. In practice, evaluating the derivatives of the trial wave function with respect to the variational parameters analitically is possible only for small systems (two to four electrons). On the other hand, the numerical solution needs the repetead evaluation of the trial wave function (the product of a Slater determinant by a Jastrow function) with respect to each variational parameter. As an example, consider using a central difference scheme to evaluate the derivative of the Slater determinant part with respect to a parameter α,

$$\frac{d\Psi_{SD}}{d\alpha} = \frac{\Psi_{SD}(\alpha+\Delta\alpha) - \Psi_{SD}(\alpha-\Delta\alpha)}{2\Delta\alpha} + \mathcal{O}(\Delta\alpha^2).$$

The reader should note that for the Slater determinant part we need to compute the expression above two times per Monte Carlo cycle per variational parameter. Computing a determinant is a highly costly operation. Moreover, the numerical accuracy in the solution will depend on the choice of the step size $\Delta\alpha$.

In the following we suggest a method to efficiently compute the derivative of the energy with respect to the variational parameters. It derives from the fact that Eq. (3.27) is equivalent to

$$\frac{\partial E}{\partial c_m} = 2\left[\left\langle E_L \frac{\partial \ln \Psi_{T_{c_m}}}{\partial c_m}\right\rangle - E\left\langle \frac{\partial \ln \Psi_{T_{c_m}}}{\partial c_m}\right\rangle\right],$$

or more precisicely,

$$\boxed{\frac{\partial E}{\partial c_m} = 2\left\{\frac{1}{N}\sum_{i=1}^{N}\left[(E_L[c_m])_i\left(\frac{\partial \ln \Psi_{T_c}}{\partial c_m}\right)_i\right] - \frac{1}{N^2}\sum_{i=1}^{N}(E_L[c_m])_i\sum_{j=1}^{N}\left(\frac{\partial \ln \Psi_{T_c}}{\partial c_m}\right)_j\right\}}, \quad (4.40)$$

and because $\Psi_{T_{c_m}} = \Psi_{SD_{c_m}}\Psi_{J_{c_m}}$, we get that

$$\begin{aligned}\ln \Psi_{T_{c_m}} &= \ln(\Psi_{SD_{c_m}}\Psi_{J_{c_m}}) = \ln(\Psi_{SD_{c_m}}) + \ln(\Psi_{J_{c_m}}) \\ &= \ln(\Psi_{SD_{c_m}\uparrow}\Psi_{SD_{c_m}\downarrow}) + \ln(\Psi_{J_{c_m}}) \\ &= \ln(\Psi_{SD_{c_m}\uparrow}) + \ln(\Psi_{SD_{c_m}\downarrow}) + \ln(\Psi_{J_{c_m}}).\end{aligned}$$

Then,

$$\boxed{\frac{\partial \ln \Psi_{T_{c_m}}}{\partial c_m} = \frac{\partial \ln(\Psi_{SD_{c_m}\uparrow})}{\partial c_m} + \frac{\partial \ln(\Psi_{SD_{c_m}\downarrow})}{\partial c_m} + \frac{\partial \ln(\Psi_{J_{c_m}})}{\partial c_m}}, \quad (4.41)$$

which is a convenient expression in terms of implementation in an object oriented fashion because we can compute the contribution to the expression above in two separated classes independently, namely the Slater determinant and Jastrow classes.

Note also that for each of the derivatives of concerning the determinants above we have, in general, that

$$\frac{\partial \ln(\Psi_{SD_{c_m}})}{\partial c_m} = \frac{\frac{\partial \Psi_{SD_{c_m}}}{\partial c_m}}{\Psi_{SD_{c_m}\uparrow}}$$

For the derivative of the Slater determinant yields that if $\mathbf{A}$ is an invertible matrix which depends on a real parameter t, and if $\frac{\mathrm{d}\mathbf{A}}{\mathrm{d}t}$ exists, then [?, ?]

$$\frac{\mathrm{d}}{\mathrm{d}t}(\det \mathbf{A}) = (\det \mathbf{A})\,\mathrm{tr}\left(\mathbf{A}^{-1}\frac{\mathrm{d}\mathbf{A}}{\mathrm{d}t}\right).$$

$$\boxed{\frac{d}{dt}\ln\det \mathbf{A}(t) = \mathrm{tr}\left(\mathbf{A}^{-1}\frac{d\mathbf{A}}{dt}\right) = \sum_{i=1}^{N}\sum_{j=1}^{N} A^{-1}_{ij}\dot{A}_{ji}}, \quad (4.42)$$

where N is the number of entries in a row. What we have here is the expression for computing the derivative of each of the determinants appearing in Eq. (4.41). Furthemore, note that the specialization of this expression to the current problem implies that the term $\mathbf{A}^{-1}$ appearing on the right hand side is the inverse of the Slater matrix, already available

after finishing each Monte Carlo cycle as deduced from the algorithms discussed in the previous sections. It means that the only thing we have to do is to take the derivative of each single wave function in the Slater matrix with respect to its variational parameter and taking the trace of $\Psi_{SD}(\alpha)^{-1}\dot{\Psi_{SD}}(\alpha)$. The implementation of this expression and its computation using analytical derivatives for the single state wave functions is straighforward. The flow chart for the Quantum Variational Monte Carlo method with optimization of the trial wave function is shown in figure 4.1.

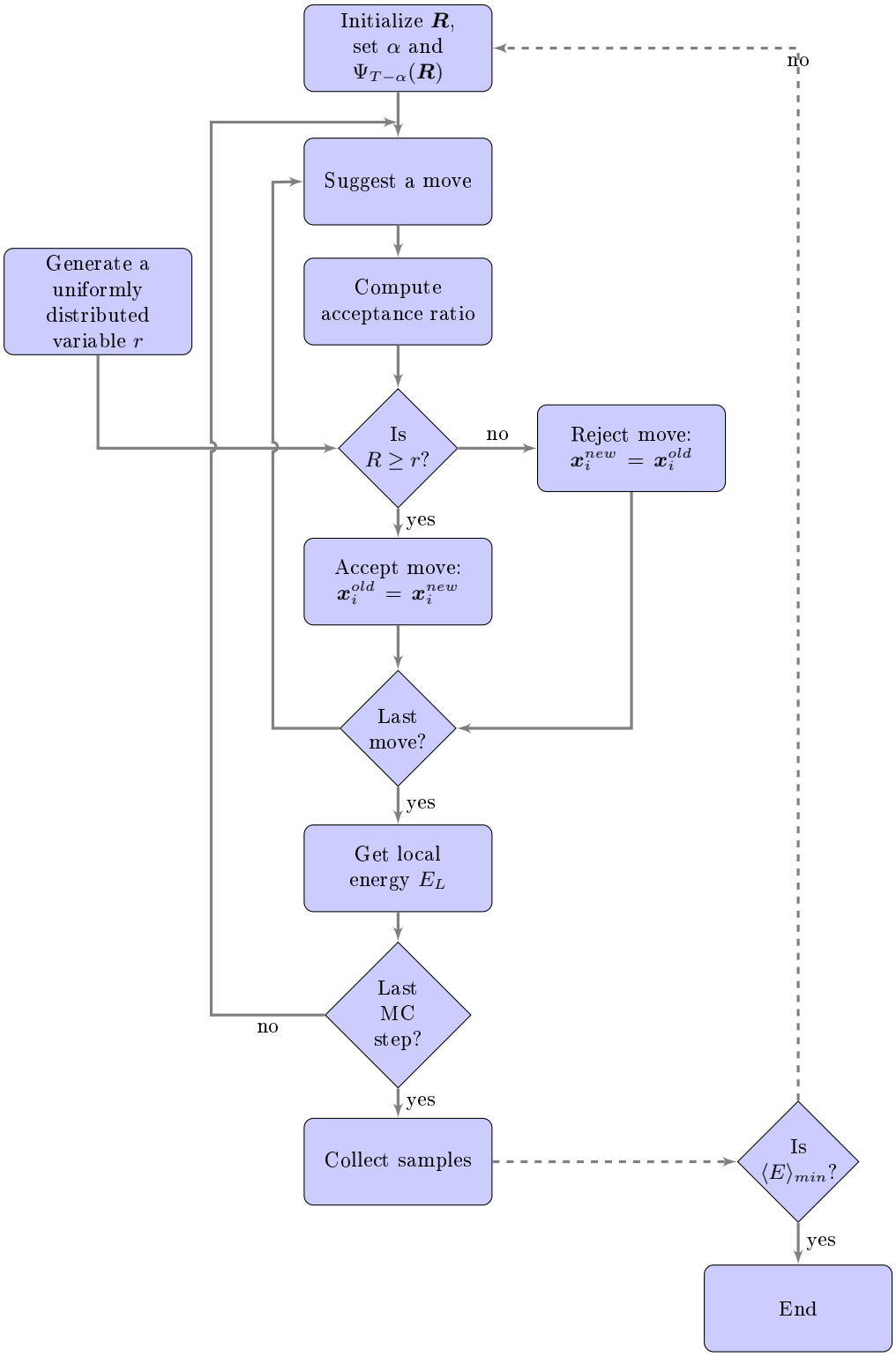

Figure 4.1: Optimization of the trial wave function $\Psi_{trial}(\boldsymbol{\alpha})$ and minimization of the energy with respect to the variational parameters.

Chapter 5

Cases of study: atoms, non-interacting electrons trapped in an harmonic oscillator potential and quantum dots

The aim of this chapter is to describe selected quantum mechanical systems in terms of the theory presented in chapter 2. First, we consider the physics of the helium and berilium atoms. They will be used as a starting point for designing an implementation in C++/Python. Later, the physics of non-interacting electrons trapped in a harmonic oscillator potential will be summarized and it will be useful in testing the flexibility of our implementation and in being able to handle other kinds of quantum mechanical problems. Finally, we consider the case of quantum dots modelled as interacting electrons trapped in a two-dimensional harmonic oscillator potential.

5.1 Case 1: He and Be atoms

The substitution of Eqs. (2.16) and (2.17) in Eq. (2.4) yields a general Hamiltonian for atomic systems[1] given in atomic units (see table 2.1) as

$$\widehat{\mathbf{H}} = \underbrace{-\sum_{i=1}^{N} \frac{\hbar^2}{2m}\nabla_i^2}_{\text{Kinetic energy}} \overbrace{\underbrace{-Z\sum_{i=1}^{N}\frac{e^2}{r_i}}_{\substack{\text{Nucleus-electron}\\ \text{potential (attraction)}}} + \underbrace{\sum_{i=1,i<j}^{N}\frac{e^2}{r_{ij}}}_{\substack{\text{Electron-electron}\\ \text{potential (repulsion)}}}}^{\text{Potential energy}}, \tag{5.1}$$

with $r_i = \sqrt{x_i^2 + y_i^2 + z_i^2}$ and $r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$ being the nucleus-electron and electron-electron distances, and where $i, j = 1, \ldots, N$.

The particular form of Eq. (5.1) for the respective Hamiltonians of He and Be atoms can be easily obtained after substitution of columns two and three of table 5.1 in Eq. (5.1).

[1] The time dependence is not considered in this thesis.

Atom	# electrons (N)	Nuclear charge (Z)
He	2	2.0
Be	4	4.0

Table 5.1: Parameters associated to the Hamiltonian of He and Be atoms.

5.1.1 Setting a trial wave function for atoms

Because of the electron-nucleus (one-body) and electron-electron (two-body) Coulomb interactions, a trial wave function for He and Be atoms should fulfill the cusp conditions discussed in section 2.6. Moreover, due to the Pauli principle, the total wave function should be antisymmetrized with respect to the interchange of particles. This is elegantly represented by the so-called Slater determinant. Therefore we start looking at Eqs. (2.23) and (2.24) for a Slater determinant. Because each row of the Slater matrix involves only the coordinates of a single electron, the derivative of the determinant equals the determinant in which row i is replaced by $\partial\phi_k/\partial r_{iA}$. Also, these two cusp conditions require that the derivative of the determinant divided by the determinant be a constant. Therefore, we must require, as discussed in [?], that each orbital satisfies:

$$\frac{1}{\phi_k}\frac{\partial\phi_k}{\partial r_{iA}} = \begin{cases} -Z_A & \text{if } l = 0 \\ -\frac{Z_A}{l+1} & \text{if } l > 0, \end{cases} \tag{5.2}$$

where the subscript k and iA represent a set of quantum numbers (n, l, m_l) and the label of the particle considered, respectively.

The radial equation can also be solved analytically resulting in the quantum numbers n in addition to (l, m_l). The solution R_{nl} to the radial equation is expressed as a function of the Laguerre polynomials. The analytic solutions are given by

$$\psi_{nlm_l}(r,\theta,\phi) = \psi_{nlm_l} = R_{nl}(r)Y_{lm_l}(\theta,\phi) = R_{nl}Y_{lm_l}. \tag{5.3}$$

The ground state is defined by $nlm_l = 100$ and reads

$$\psi_{100} = \frac{1}{a_0^{3/2}\sqrt{\pi}}e^{-r/a_0}, \tag{5.4}$$

where we have defined the Bohr radius $a_0 = 0.05$ nm

$$a_0 = \frac{\hbar^2}{mke^2}. \tag{5.5}$$

The first excited state with $l = 0$ is

$$\psi_{200} = \frac{1}{4a_0^{3/2}\sqrt{2\pi}}\left(2 - \frac{r}{a_0}\right)e^{-r/2a_0}. \tag{5.6}$$

For states from $l = 1$ and $n = 2$, the spherical harmonics can have imaginary components. Fortunately, the introduction of solid harmonics allows the use of real orbital wave-functions for a wide range of applications. This, however, is out of the scope of this thesis. We remark

however, that due to the flexibility of our code, any basis can actually be accomodated rather easily.

Trial wave function for the Helium atom

The electron-nucleus cusp condition given above suggests a trial wave function which takes the following form

$$\phi_T(\boldsymbol{r}_i) \propto e^{-\frac{Z}{l+1}r_i}.$$

For atomic systems described by a central symmetric potential a choice would be to use hydrogen-like wave functions. In an He atom we have two electrons filling the single state $1s$, i.e., $n = 1$, $l = 0$, one with spin up and the other one with spin down. In this case we set

$$\phi_{1s}(\boldsymbol{r}_i) = e^{-\alpha r_i}, \quad i = 1, 2, \tag{5.7}$$

with $\boldsymbol{r}_i = x_i\boldsymbol{e}_1 + y_i\boldsymbol{e}_2 + z_i\boldsymbol{e}_3$ and $r_i = \sqrt{x_i^2 + y_i^2 + z_i^2}$.

A trial wave function[2] for the He atom is therefore

$$\Psi_T = e^{\alpha(r_1+r_2)}e^{\frac{a_{12}r_{12}}{(1+\beta_{12}r_{12})}}, \tag{5.8}$$

where the first exponential function is a Slater determinant[3] and the second is the linear Padé-Jastrow correlation function. The (set of) parameter(s) α has the interpretation of charge and together with β can be used to optimize the total trial wave function, while $a_{12} = 0.5$ as deduced from Eq. (2.27).

For the helium atom we 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 function we discussed above was not antisymmetric with respect to an 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 antisymmetric. The product is therefore antisymmetric as well. The Slater-determinant consists of single-particle spin-orbitals.

Trial wave function for the Beryllium atom

For the Be atom, we have four electrons, two of which occupy the single quantum state $1s$. As in the He case, the orbital corresponding orbital is given by Eq. (5.7). The other two electrons, in turn, complete the $2s$ state (shell $n = 2$, orbital $l = 0$), one having spin up and the other one spin down. The trial single particle wave function is in this case

$$\phi_{2s}(\boldsymbol{r_i}) = \left(1 - \frac{\alpha}{2}\right)e^{-\alpha/2}r_i, \quad \text{for} \quad i = 1, 2, \tag{5.9}$$

[2]The wave function is not properly normalized as in this thesis we are concerned only with ratios and therefore the normalization factors cancel each other. Moreover, the spin part of the wave-function is here antisymmetric. This has no effect when calculating physical observables because the sign of the wave function is squared in all expectation values.

[3]In general, the orbitals forming a Slater determinant are orthonormal.

and the total trial wave function takes the form,

$$\Psi_T(\boldsymbol{r}) = \frac{1}{\sqrt{4!}} \begin{vmatrix} \phi_{1s}(\boldsymbol{r_1}) & \phi_{2s}(\boldsymbol{r_1}) \\ \phi_{1s}(\boldsymbol{r_2}) & \phi_{2s}(\boldsymbol{r_2}) \end{vmatrix}_{\uparrow} \begin{vmatrix} \phi_{1s}(\boldsymbol{r_3}) & \phi_{2s}(\boldsymbol{r_3}) \\ \phi_{1s}(\boldsymbol{r_4}) & \phi_{2s}(\boldsymbol{r_4}) \end{vmatrix}_{\downarrow} \exp\left(\sum_{i<j} \frac{a_{ij} r_{ij}}{1+\beta_{ij} r_{ij}}\right), \tag{5.10}$$

where we have omitted the set of parameters α for the orbitals.

Energy levels in atoms and degeneracy

The energy, with no external magnetic field, is determined by the radial Schröndinger equation, which can also be solved analytically resulting in the quantum numbers n in addition to (l, m_l). The solution R_{nl} is expressed in terms of Laguerre polynomials. Its derivation can be carried out using a series method. This, also, gives rise to the following expression for the quantized energy levels [?]:

$$\boxed{E_n = -\frac{\hbar^2}{2ma_0^2}\frac{Z^2}{n^2},} \tag{5.11}$$

where the principal quantum number $n = 1, 2, \ldots$ decides the energy levels, and it is related to the quantum number of the orbital angular momentum by $l = 0, 1, 2, \ldots, n-1$.

For each value of n we can have $(n-1)$ values of l. Furthemore, for each value of l, there are $(2l+1)$-values for the magnetic quantum number m_l. Therefore, for a given energy level E_n we have a spatial degeneracy of

$$\boxed{g_n = \sum_{l=0}^{n-1}(2l+1) = n^2.} \tag{5.12}$$

Including the two spin degrees of freedom we get

$$\boxed{g_n = 2n^2.} \tag{5.13}$$

5.2 Case 2: Non-interacting particles trapped in an harmonic oscillator potential

The harmonic oscillator provides a useful model for vibrational phenomena encountered in several branches of physics. It can also be seen as a first approximation to quantum dots for which we neglect the interactions between the charge carriers. The Hamiltonian (in one dimension) for a particle of mass m oscillating with a frequency ω is

$$H = -\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + \frac{1}{2}kx^2 = -\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + \frac{1}{2}m\omega x^2, \tag{5.14}$$

where k is the force constant, e.g., the spring tension for a classical oscillator. The time-independent Schrödinger equation (2.7) takes the form

$$\boxed{-\frac{\hbar^2}{2m}\frac{d^2\Psi}{dx^2} + \frac{1}{2}m\omega^2 x^2\Psi = E\Psi.} \tag{5.15}$$

The introduction of the natural scales

$$\bar{x} = \sqrt{\frac{m\omega}{\hbar}}x, \quad \text{and} \quad \epsilon = \frac{2E}{\hbar\omega} \tag{5.16}$$

leads to the dimensionless Schrödinger equation

$$\frac{d^2\Psi}{dx^2} + (\epsilon - x)\Psi = 0, \tag{5.17}$$

with $\Psi = \Psi(x)$ and where we have dropped the bar in the dimensionless variable for simplicity in the notation.

Solutions to Eq. (5.17) can be obtained using the series method [?, ?]. They are the product of an exponential and a Hermite polynomial. These polynomials have the following Rodrigues formula:

$$H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2}, \tag{5.18}$$

from which we find that

$$\Psi_n(x) = H_n(x) e^{-\frac{x^2}{2}},$$

or in terms of the position, the solution (in the original coordinates) takes the form:

$$\boxed{\Psi_n(x) = A_n H_n\left(\sqrt{\frac{m\omega}{\hbar}}x\right) e^{-\frac{m\omega x^2}{2\hbar}},} \tag{5.19}$$

where $A_n = \frac{1}{\sqrt{2^n n!}} \left(\frac{m\omega}{\pi\hbar}\right)^{\frac{1}{4}}$ is a normalization constant.

Higher-order Hermite polynomials can be derived by using the following recursive relations:

$$H_{n+1}(x) = 2xH_n(x) - 2xH_{n-1}(x), \quad \text{and} \quad \frac{dH_n}{dx} = 2xH_{n-1}(x). \tag{5.20}$$

The energy is found from the series solution applied to the Schrödinger equation (5.17). The termination condition restrictes the dimensionless variable of energy to the quantized values $\epsilon_n = 2n + 1, \quad n = 0, 1, 2, \ldots$. This expression substituted in the right hand side member of Eq. (5.16) leads to the form of the energy of state n given by

$$E_n = \left(n + \frac{1}{2}\right)\hbar\omega, \quad n = 0, 1, 2, \ldots,$$

with the ground state $E_0 = \frac{1}{2}\hbar\omega$. The energy is both discrete and non-degenerate.

For the multi-dimensional case[4], say 3D, assuming a central potential given by

$$V(r) = \frac{1}{2}m\omega^2 r^2 = \frac{1}{2}m(\omega_x^2 x^2 + \omega_y^2 y^2 + \omega_z^2 z^2) \tag{5.21}$$

[4] An example of a tridimensional harmonic oscillator is an atom in a crystal, which has an equilibrium position in the lattice, about which it executes single harmonic motion when it experiments a small perturbation [?].

we have the following Hamiltonian:

$$H = \frac{p^2}{2m} + \frac{1}{2}m\omega^2 r^2 \tag{5.22}$$

$$= \left(\frac{p_x^2}{2m} + \frac{1}{2}m\omega_x^2 x^2\right) + \left(\frac{p_y^2}{2m} + \frac{1}{2}m\omega_y^2 y^2\right) + \left(\frac{p_z^2}{2m} + \frac{1}{2}m\omega_z z^2\right)$$

$$= \left(-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + \frac{1}{2}m\omega_x x^2\right) + \left(-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial y^2} + \frac{1}{2}m\omega_y y^2\right) + \left(-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial z^2} + \frac{1}{2}m\omega_z z^2\right)$$

$$= H_x + H_y + Hz \tag{5.23}$$

This problem can be solved analytically by separation of variables. Introducing

$$\Psi_n(x, y, z) = \psi_{nx}(x)\psi_{ny}(y)\psi_{nz}(z) \tag{5.24}$$

with the splitted Hamiltonian given above into the time-independent Schrödinger equation, we get three independent harmonic oscillators. The total energy of the system equals therefore the sum of energies of each of them along respective cartesian coordinates, i.e.,

$$E_n = \left(n_x + \frac{1}{2}\right)\hbar\omega_x + \left(n_y + +\frac{1}{2}\right)\hbar\omega_y + \left(nz + \frac{1}{2}\right)\hbar\omega_z. \tag{5.25}$$

A generalization of equations (5.23) to (5.25) for the $d-$dimensional is easy to obtain [?]. Starting with $H = \sum_{i=1}^{d} H_i$, for d running over the number of spatial dimensions, one notes that $\Psi_n(x_1, \ldots, x_d) = \prod_{i=1}^{d} \psi_{n_i}(x_i)$, or more precisely,

$$\boxed{\Psi_n(x) = \prod_{i=1}^{d} \frac{1}{\sqrt{2^{n_i} n_i!}} \left(\frac{m\omega_i}{\pi\hbar}\right)^{\frac{1}{4}} H_n\left(\sqrt{\frac{m\omega_i}{\hbar}} x_i\right) e^{-\frac{m\omega x_i^2}{2\hbar}},} \tag{5.26}$$

where the trial wave function needed by the VMC algorithm is obtained by setting $\omega \to \alpha\omega$.

The above equation implies that $E_n = \sum_{i=1}^{d} E_i$. Hence,

$$\boxed{E_n = \sum_{i=1}^{d}\left(n_i + \frac{1}{2}\right)\hbar\omega_i = \left(n_1 + \frac{1}{2}\right)\hbar\omega_1 + \ldots + \left(n_d + \frac{1}{2}\right)\hbar\omega_d.} \tag{5.27}$$

Under isotropy, the angular frequency is the same in all the directions and $\omega_i = \omega$. With exception of the ground state $n_i = 0$, i=1,2,..,d, it is clear from Eq. (5.25) that different combinations of quantum numbers yield the same energy. Therefore we say that the excited states are degenerate.

Degeneracy

The degeneracy can be calculated by noting that all the states with the same n will have the same energy. The eigenenergies in an anisotropic case are not degenerate because the corresponding potential is not symmetric. In general, the degree of spatial degeneracy of a energy level equals the number of ways n can be written as the sum of d non-negative

numbers, where d equals the number of spatial dimensions [?]. That is,

$$\boxed{g_n = \binom{d+n-1}{n} = \frac{(d+n-1)!}{n!(d-1)!}.} \tag{5.28}$$

Then, for the 3D harmonic oscillator we have $g_n = \frac{1}{2}(n+1)(n+2)$, while the 2D-case yields $g_n = \frac{1}{2}(n+1)(n+2)$. Including the spin degrees of freedom, the total degeneracy becomes:

$$\boxed{G_n = 2(n+1)} \tag{5.29}$$

and

$$\boxed{G_n = (n+1)(n+2),} \tag{5.30}$$

for the 3D- and 2D- cases, respectively.

5.3 Case 3: Quantum dots

Semiconductor quantum dots are structures where charge carriers are confined in all three spatial dimensions, the dot size being of the order of the Fermi wavelength in the host material, typically between 10 nm and 1 μm. The confinement is usually achieved by electrical gating of a two-dimensional electron gas (2DEG), possibly combined with etching techniques. Precise control of the number of electrons in the conduction band of a quantum dot (starting from zero) has been achieved in GaAs heterostructures. The electronic spectrum of typical quantum dots can vary strongly when an external magnetic field is applied, since the magnetic length corresponding to typical laboratory fields is comparable to typical dot sizes. In coupled quantum dots Coulomb blockade effects, tunneling between neighboring dots, and magnetization have been observed as well as the formation of a delocalized single-particle state.

In the following we look at a single two-dimensional quantum dot confined modelled by interacting electrons trapped by an harmonic oscillator potential. The classical Hamiltonian for an electron moving in a central potential[5] $V(r)$ besides being subject to a uniform magnetic field $\boldsymbol{B}$ is given by [?, ?]

$$\widehat{\mathbf{H}} = \frac{1}{2m^*}\left(\widehat{\mathbf{p}} - e\boldsymbol{A}\right)^2 + e\phi + V(r),$$

with $\boldsymbol{p}$ being the classical momentum, e the charge and m^* the effective mass[6] of the electron in the host material, respectively. Furthemore, $\boldsymbol{A}$ and $e\phi$ denote the external magnetic (vector) and electric (scalar) potentials per charge unit. Setting $e\phi = 0$ and remembering

[5] Note that this potential is not of electromagnetic origin.
[6] The effective mass can be determined as long as we have the full energy of the system.

that in quantum mechanics $\boldsymbol{p} \to \widehat{\mathbf{p}} = -i\hbar\nabla$ we get

$$\begin{aligned}\widehat{\mathbf{H}} &= \frac{1}{2m^*}\left(-i\hbar\boldsymbol{\nabla} - e\boldsymbol{A}\right)\cdot\left(-i\hbar\boldsymbol{\nabla} - e\boldsymbol{A}\right) + \frac{1}{2}m^*\omega_0^2(x^2+y^2) \\ &= \underbrace{-\frac{\hbar^2}{2m^*}\nabla^2 + \frac{1}{2}m^*\omega_0^2(x^2+y^2)}_{\widehat{\mathbf{H}}_0} - \frac{e}{2m^*}(\widehat{\mathbf{p}}\boldsymbol{A} + \boldsymbol{A}\widehat{\mathbf{p}}) + \frac{e^2}{2m^*}\boldsymbol{A}^2, \end{aligned} \tag{5.31}$$

where ω_0 represents the frequency of oscillation of the electron. In the cross terms we get, in general,

$$\widehat{\mathbf{p}}(\boldsymbol{A}\Psi) - \boldsymbol{A}(\widehat{\mathbf{p}}\Psi) = -i\hbar[(\boldsymbol{\nabla}\cdot\boldsymbol{A})\Psi + \boldsymbol{A}(\boldsymbol{\nabla}\Psi) - \boldsymbol{A}\boldsymbol{\nabla}\Psi] \neq 0,$$

i.e, the operators $\widehat{\mathbf{p}}$ and $\widehat{\mathbf{A}}$ do not commute. An exception is made when we set the constraint $\boldsymbol{\nabla}\cdot\boldsymbol{A} = 0$ (Coulomb gauge). From classical electrodynamics, the magnetic field can be written as $\boldsymbol{B} = \boldsymbol{\nabla}\times\boldsymbol{A}$. For a magnetic field oriented in the $z-$direction, normal to the two-dimensional xy-plane of the motion (where the electrons is supposed to be confined), we can write $\boldsymbol{B} = (0,0,B_z)$ chosing $\boldsymbol{A} = \frac{1}{2}(-B_y, B_x, 0) = \frac{1}{2}(\boldsymbol{B}\times\boldsymbol{r})$, for the Coulomb gauge to be fulfilled. Hence,

$$\boldsymbol{A}\cdot\boldsymbol{p} = \frac{1}{2}(\boldsymbol{B}\times\boldsymbol{r})\cdot\boldsymbol{p} = \frac{1}{2}\boldsymbol{B}\cdot(\boldsymbol{r}\times\boldsymbol{p}) = \frac{1}{2}\boldsymbol{B}\cdot\boldsymbol{L} = \frac{1}{2}BL_z,$$

where $\widehat{\mathbf{L}}_z = \widehat{\mathbf{x}}\widehat{\mathbf{p}}_y - \widehat{\mathbf{y}}\widehat{\mathbf{p}}_x$ is the $z-$component fo the angular momentum. Thus Eq. (5.31) becomes,

$$\widehat{\mathbf{H}} = \widehat{\mathbf{H}}_0 - \frac{e}{2m^*}BL_z + \frac{e^2B^2}{8m^*}(x^2+y^2),$$

If a particle with charge q has an intrisic spin $\boldsymbol{S}$, its spinning motion gives rise to a magnetic dipole moment $\boldsymbol{\mu_S} = \frac{q\boldsymbol{S}}{2m^*c}$ that when interacting with an external magnetic field generates an energy $-\boldsymbol{\mu_S}\cdot\boldsymbol{B}$ that must be added to the Hamiltonian [?]. We get therefore,

$$\begin{aligned}\widehat{\mathbf{H}} &= \widehat{\mathbf{H}}_0 - \frac{e}{2m^*}BL_z + \frac{e^2B^2}{8m^*}(x^2+y^2) - \boldsymbol{\mu_S}\cdot\boldsymbol{B} \\ &= \widehat{\mathbf{H}}_0 - \frac{e}{2m^*}BL_z + \frac{e^2B^2}{8m^*}(x^2+y^2) + g_s\frac{e}{2m^*}B\widehat{\mathbf{S}}_z \\ &= \widehat{\mathbf{H}}_0 - \frac{\omega_c}{2}L_z + \frac{1}{2}m^*\left(\frac{\omega_c}{2}\right)^2(x^2+y^2) + \frac{1}{2}g_s\omega_c\widehat{\mathbf{S}}_z,\end{aligned}$$

where $\widehat{\mathbf{S}}$ and g_s are the spin operator and the effective spin gyromagnetic ratio of the electron, respectively. Moreover $\omega_c = eB/m^*$ is known as the cyclotron frequency.

For several particles trapped in the same quantum dot, the electron-electron Coulomb in-

teraction should be included, yielding the general Hamiltonian:

$$\widehat{\mathbf{H}} = \sum_{i=1}^{N}\Bigg[-\frac{\hbar^2}{2m^*}\nabla_i^2 + \overbrace{\frac{1}{2}m^*\omega_0^2(x_i^2+y_i^2)}^{\text{Harmonic oscillator potential}}\Bigg] + \overbrace{\frac{e^2}{4\pi\epsilon_0\epsilon_r}\sum_{i=1}^{N}\sum_{j=i+1}^{N}\frac{1}{|\boldsymbol{r}_i-\boldsymbol{r}_j|}}^{\text{Electron-electron interaction}} \tag{5.32}$$
$$+\underbrace{\sum_{i=1}^{N}\Bigg[\underbrace{\frac{1}{2}m^*\left(\frac{\omega_c}{2}\right)^2(x_i^2+y_i^2)}_{\text{"Squeeze" term}} - \underbrace{\frac{\omega_c}{2}\widehat{\mathbf{L}}_z^{(i)}}_{\text{Rotation term}} + \frac{1}{2}g_s\omega_c\widehat{\mathbf{S}}_z^{(i)}\Bigg]}_{\text{electron-magnetic field interaction}}.$$

Setting $\omega_B = \left(\frac{\omega_c}{2}\right)^2$ and combining the magnetic confinement and the external potential we get the new oscillator frequency $\omega^2 = \omega_0^2 + \omega_B^2$, leading to

$$\widehat{\mathbf{H}} = \sum_{i=1}^{N}\left[-\frac{\hbar^2}{2m^*}\nabla_i^2 + \frac{1}{2}m^*\omega^2(x_i^2+y_i^2)\right] + \frac{e^2}{4\pi\epsilon_0\epsilon_r}\sum_{i=1}^{N}\sum_{j=i+1}^{N}\frac{1}{|\boldsymbol{r}_i-\boldsymbol{r}_j|} + \sum_{i=1}^{N}\left[-\frac{\omega_c}{2}\widehat{\mathbf{L}}_z^{(i)} + \frac{1}{2}g_s\omega_c\widehat{\mathbf{S}}_z^{(i)}\right]. \tag{5.33}$$

Using scaled atomic units[7], $\hbar = m^* = \epsilon = e = 1$

$$\widehat{\mathbf{H}} = \sum_{i=1}^{N}\left[-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2(x_i^2+y_i^2)\right] + \sum_{i=1}^{N}\sum_{j=i+1}^{N}\frac{1}{r_{ij}} + \sum_{i=1}^{N}\left[-\frac{\omega_c}{2}\widehat{\mathbf{L}}_z^{(i)} + \frac{1}{2}g_s\omega_c\widehat{\mathbf{S}}_z^{(i)}\right]. \tag{5.34}$$

Because the operators $\widehat{\mathbf{L}}_z$ and $\widehat{\mathbf{S}}_z$ commutes with the Hamiltonian, i.e., $[\widehat{\mathbf{H}}, \widehat{\mathbf{L}}_z] = [\widehat{\mathbf{H}}, \widehat{\mathbf{S}}_z] = 0$, the solution of the Schrödinger equation will be an eigenfunction of $\widehat{\mathbf{L}}_z$ and $\widehat{\mathbf{S}}_z$. Therefore, the contributions to the total energy due to these operators can be evaluated separately. Finally the Hamiltonian we implement has the simple form

$$\boxed{\widehat{\mathbf{H}} = \sum_{i=1}^{N}\left[-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2(x_i^2+y_i^2)\right] + \sum_{i=1}^{N}\sum_{j=i+1}^{N}\frac{1}{r_{ij}},} \tag{5.35}$$

where we can just add the contributions of the other terms to the resulting energy.

5.4 Verifying and validating the implementation

Verification and validation of codes are two important aspects in the design of a software. The former refers to an assessment of the accuracy of the solution to a computational model, the latter is the assessment of the accuracy of a computational model by comparison with experimental/analytical data. Both of them indicate when an algorithm is working as expected, and they are important for debugging while developing the implementation. With this aim in mind, some analytical results concerning He and Be atoms, as well as the

[7] The "scaled" term derives from the fact that we are equaling to one the dielectric constant and reduced mass (nor the mass).

case of non-interacting particles trapped in an harmonic oscillator potential in one to three dimensions are exposed in the following.

5.4.1 Analytical results for the He and Be atoms

The Slater determinant of the He and Be atoms

Analytical expressions can be obtained for the Slater determinant of the atoms studied. For He it is just

$$\boxed{\Psi_{SD} = e^{\alpha(r_1+r_2)}.} \tag{5.36}$$

For Be we compute it in the standard way leading to

$$\boxed{\Psi_{SD} = [\phi_{1s}(\boldsymbol{r_1})\phi_{2s}(\boldsymbol{r_2}) - \phi_{1s}(\boldsymbol{r_2})\phi_{2s}(\boldsymbol{r_1})][\phi_{1s}(\boldsymbol{r_3})\phi_{2s}(\boldsymbol{r_4}) - \phi_{1s}(\boldsymbol{r_4})\phi_{2s}(\boldsymbol{r_3})].} \tag{5.37}$$

The derivatives needed in the quantum force and the kinetic energy can be computed numerically using central difference schemes.

Ground state energies of He and Be without including the correlation part

A first approach to derive the value of the ground state energy for He and Be analytically consists in omitting the correlation part of the potential and in the trial wave functions. The Hamiltonian in Eq. (2.9) expressed in atomic units reads

$$\boxed{\widehat{\mathbf{H}} = -\frac{1}{2}\sum_{i=1}^{N}\nabla_i^2 - Z\sum_{i=1}^{N}\frac{1}{r_i},} \tag{5.38}$$

and similarly, for the energy Eq. (5.11) of the eigenstate with quantum numbers nlm_l we get

$$\boxed{E_n = -\frac{1}{2}\frac{Z^2}{n^2}.} \tag{5.39}$$

From Eqs. (2.11) and (2.12), the total energy of the system equals the sum of eigenenergies of each single state. Then, the ground states energies[8] of He and Be atoms when no correlation part is included are

$$\boxed{E = E_1 + E_1 = -\frac{1}{2}(2)^2\left(\frac{1}{1}+\frac{1}{1}\right) = -4 \quad \text{au},} \tag{5.40}$$

and

$$\boxed{E = (E_1+E_2) + (E_1+E_2) = -\frac{1}{2}(4)^2\left(\frac{1}{1}+\frac{1}{2^2}+\frac{1}{1}+\frac{1}{2^2}\right) = -20 \quad \text{au}.} \tag{5.41}$$

In order to reproduce these ground states energies, the parameter α in the trial wave functions (or better in the single state wave functions) of He and Be should be set equal to 2.0 and 4.0, respectively. This comes from the fact that, in atoms, the α parameter has the interpretation of a nuclear charge.

[8] In order to convert from atomic to SI units multiply by $2E_0$, where $E_0 = 13.6\,eV$ is the binding energy of the H atom.

Analytical gradient and Laplacian of 1s and 2s orbitals

The gradient and Laplacian of the single state wave functions are needed in the evaluation of the quantum force and the kinetic energy terms. The gradients for this two orbitals are given by

$$\boxed{\boldsymbol{\nabla}\phi_{1s} = -\alpha\left(\frac{x}{r}\hat{\boldsymbol{e}}_1 + \frac{y}{r}\hat{\boldsymbol{e}}_2 + \frac{z}{r}\hat{\boldsymbol{e}}_3\right)e^{-\alpha r} = -\alpha\frac{\boldsymbol{r}}{r}e^{-\alpha r},} \tag{5.42}$$

and

$$\boxed{\boldsymbol{\nabla}\phi_{2s} = -\frac{\alpha}{2}\left(2-\frac{\alpha}{2}r\right)\left(\frac{x}{r}\hat{\boldsymbol{e}}_1 + \frac{y}{r}\hat{\boldsymbol{e}}_2 + \frac{z}{r}\hat{\boldsymbol{e}}_3\right)e^{-\frac{\alpha}{2}r} = -\frac{\alpha}{2}\left(2-\frac{\alpha}{2}r\right)\frac{\boldsymbol{r}}{r}e^{-\frac{\alpha}{2}r}.} \tag{5.43}$$

The laplacians are

$$\boxed{\nabla^2\phi_{1s} = \left(\alpha^2 - \frac{2\alpha}{r}\right)e^{-\alpha r},} \tag{5.44}$$

and

$$\boxed{\nabla^2\phi_{2s} = -\frac{2}{\alpha r}\left[4 - 5\left(\frac{\alpha r}{2}\right) + \left(\frac{\alpha r}{2}\right)^2\right]e^{-\frac{\alpha}{2}r}.} \tag{5.45}$$

Analytical expressions for the local energy of He

The local energy operator reads

$$\widehat{\mathbf{E}}_L = \frac{1}{\Psi_T}\widehat{\mathbf{H}}\Psi_T.$$

At a first glance, we consider the case when the correlation part of the trial wave function is omitted. Then, the Hamiltonian is

$$\widehat{\mathbf{H}} = -\frac{1}{2}\boldsymbol{\nabla}_1^2 - \frac{1}{2}\boldsymbol{\nabla}_2^2 - \frac{Z}{r_1} - \frac{Z}{r_2} + \frac{1}{r_{12}}. \tag{5.46}$$

Noting that the quantity contributed by the potential to the local energy operator is the potential itself and after substitution of Eq. (5.44) we get

$$\boxed{\widehat{\mathbf{E}}_{L_1} = (\alpha - Z)\left(\frac{1}{r_1} + \frac{1}{r_2}\right) + \frac{1}{r_{12}} - \alpha^2,} \tag{5.47}$$

which has the expectation value

$$\boxed{\langle E_{L_1}\rangle = \alpha^2 - 2\alpha\left(Z - \frac{5}{16}\right).} \tag{5.48}$$

The last result is useful when testing the derivatives of the local energy during parameter optimization of the trial wave function. Observe, also, that setting $Z = 2$ the above expression gives a minimum for $\alpha = 1.6875$.

The local energy of the He atom with the Padé-Jastrow trial wave function given in Eq. (5.8)

n	(n_x, n_y)	g_n	$G_n = 2g_n$	Filled shell	$G_n E_n/(\hbar\omega)$
0	(00)	1	2	2	2
1	(10), (01)	2	4	6	8
2	(20), (02)	3	6	12	18
	(11)				

Table 5.2: Energy levels and their degeneracy for a two-dimensional isotropic harmonic oscillator. Filled shell refers to have all the energy levels occupied.

n	(n_x, n_y, n_z)	g_n	$G_n = 2g_n$	Filled shell	$G_n E_n/(\hbar\omega)$
0	(000)	1	2	2	3
1	(100), (010), (001)	3	6	8	15
2	(200), (020), (002)	6	12	20	42
	(110), (101), (011)				

Table 5.3: Energy levels and their degeneracy for a tri-dimensional isotropic harmonic oscillator. Filled shell refers to have all the energy levels occupied.

can be derived analytically and reads

$$\boxed{\widehat{\mathbf{E}}_L = -4 + \frac{\beta}{1+\beta r_{12}} + \frac{\beta}{(11+\beta r_{12})^2} + \frac{\beta}{(1+\beta r_{12})^3} - \frac{1}{4(1+\beta r_{12})^4} + \frac{\boldsymbol{r}_{12}\cdot(\boldsymbol{r}_1 - \boldsymbol{r}_2)}{(1+\beta r_{12})^2}.} \tag{5.49}$$

An alternative expression is

$$\boxed{\widehat{\mathbf{E}}_L = E_{L_1} + \frac{1}{2(1+\beta r_{12})^2}\left[\frac{\alpha(r_1 - r_2)}{r_{12}}\left(1 - \frac{\boldsymbol{r}_1\cdot\boldsymbol{r}_2}{r_1 r_2}\right) - \frac{1}{2(1+\beta r_{12})^2} - \frac{2}{r_{12}} + \frac{2\beta}{1+\beta r_{12}}\right].} \tag{5.50}$$

The computation of the energy for the Be atom can be done similarly. It will however involve more work, because the Hamiltonian is defined by eleven terms, more than twice the number of terms in the case of the He atom.

5.4.2 Analytical expressions for the two- and three-dimensional harmonic oscillator

Energy levels and degeneracy of the 2D and 3D cases

Equations (5.27) and (5.28) to (5.29) express the energy levels and degeneracies for the harmonic oscillator. Results concerning the ground-state and the two first excited states are summarized in tables 5.2 and 5.3 for the two- and three-dimensional cases, respectively. Column five (filled shell) shows the number of electrons in the system neccessary to occupy (fill) completely up to a given level of energy. Moreover, column six refers to the energy of the respective level. Then, the total energy of the system equals the sum of the energies of all the filled levels. For the first excited state we have values of 10 and 18 in two and three dimensions, respectively. These energies as well as the ground states should be reproduced by the simulator after setting $\omega = 1$ and $\alpha = 1$ in the trial wave functions.

Trial wave functions, gradients and Laplacians

The trial wave functions corresponding to the set of quantum numbers (eigenstates) listed in columns two of tables 5.2 and 5.3 can be constructed from Eq. (5.26). The Hermitian polynomials needed for the ground-state and the first excited states can be derived from Eq. (5.18) and Eq. (5.20) yielding

$$H_0(\sqrt{\alpha\omega}\xi) = 1 \qquad H_1(\sqrt{\alpha\omega}\xi) = 2\sqrt{\alpha\omega}\xi, \tag{5.51}$$

where ξ represents one of the three spatial components x, y or z.

For two dimensions we have

$$\phi_{00}(\boldsymbol{r}) = e^{-\frac{1}{2}\alpha\omega(x^2+y^2)} \tag{5.52}$$
$$\phi_{10}(\boldsymbol{r}) = 2\sqrt{\alpha\omega}xe^{-\frac{1}{2}\alpha\omega(x^2+y^2)} \tag{5.53}$$
$$\phi_{01}(\boldsymbol{r}) = 2\sqrt{\alpha\omega}ye^{-\frac{1}{2}\alpha\omega(x^2+y^2)}, \tag{5.54}$$

with the corresponding gradients

$$\boldsymbol{\nabla}\phi_{00}(\boldsymbol{r}) = -\alpha\omega(x\hat{\boldsymbol{e}}_1 + y\hat{\boldsymbol{e}}_2)\phi_{00}(\boldsymbol{r}) \tag{5.55}$$
$$\boldsymbol{\nabla}\phi_{10}(\boldsymbol{r}) = \frac{1-\alpha\omega x^2}{x}\phi_{10}(\boldsymbol{r})\hat{\boldsymbol{e}}_1 - \alpha\omega y\phi_{10}(\boldsymbol{r})\hat{\boldsymbol{e}}_2 \tag{5.56}$$
$$\boldsymbol{\nabla}\phi_{01}(\boldsymbol{r}) = -\alpha\omega x\phi_{01}(\boldsymbol{r})\hat{\boldsymbol{e}}_1 + \frac{1-\alpha\omega y^2}{y}\phi_{01}(\boldsymbol{r})\hat{\boldsymbol{e}}_2, \tag{5.57}$$

and Laplacians

$$\nabla^2\phi_{00}(\boldsymbol{r}) = -\alpha\omega[2-\alpha\omega(x^2+y^2)]\phi_{00}(\boldsymbol{r}) \tag{5.58}$$
$$\nabla^2\phi_{10}(\boldsymbol{r}) = [-4\alpha\omega + (\alpha\omega)^2(x^2+y^2)]\phi_{10}(\boldsymbol{r}) \tag{5.59}$$
$$\nabla^2\phi_{01}(\boldsymbol{r}) = [-4\alpha\omega + (\alpha\omega)^2(x^2+y^2)]\phi_{01}(\boldsymbol{r}). \tag{5.60}$$

For three dimensions, it reads

$$\phi_{000}(\boldsymbol{r}) = e^{-\frac{1}{2}\alpha\omega(x^2+y^2+z^2)} \tag{5.61}$$
$$\phi_{100}(\boldsymbol{r}) = 2\sqrt{\alpha\omega}xe^{-\frac{1}{2}\alpha\omega(x^2+y^2+z^2)} \tag{5.62}$$
$$\phi_{010}(\boldsymbol{r}) = 2\sqrt{\alpha\omega}ye^{-\frac{1}{2}\alpha\omega(x^2+y^2+z^2)} \tag{5.63}$$
$$\phi_{001}(\boldsymbol{r}) = 2\sqrt{\alpha\omega}ze^{-\frac{1}{2}\alpha\omega(x^2+y^2+z^2)}, \tag{5.64}$$

with gradients

$$\boldsymbol{\nabla}\phi_{000}(\boldsymbol{r}) = -\alpha\omega(x\hat{\boldsymbol{e}}_1 + y\hat{\boldsymbol{e}}_2 + zy\hat{\boldsymbol{e}}_3)\phi_{000}(\boldsymbol{r}) \tag{5.65}$$
$$\boldsymbol{\nabla}\phi_{100}(\boldsymbol{r}) = \frac{1-\alpha\omega x^2}{x}\phi_{100}(\boldsymbol{r})\hat{\boldsymbol{e}}_1 - \alpha\omega y\phi_{100}(\boldsymbol{r})\hat{\boldsymbol{e}}_2 - \alpha\omega z\phi_{100}(\boldsymbol{r})\hat{\boldsymbol{e}}_3 \tag{5.66}$$
$$\boldsymbol{\nabla}\phi_{010}(\boldsymbol{r}) = -\alpha\omega x\phi_{010}(\boldsymbol{r})\hat{\boldsymbol{e}}_1 + \frac{1-\alpha\omega y^2}{y}\phi_{010}(\boldsymbol{r})\hat{\boldsymbol{e}}_2 - \alpha\omega z\phi_{010}(\boldsymbol{r})\hat{\boldsymbol{e}}_3 \tag{5.67}$$
$$\boldsymbol{\nabla}\phi_{001}(\boldsymbol{r}) = -\alpha\omega x\phi_{001}(\boldsymbol{r})\hat{\boldsymbol{e}}_1 - \alpha\omega y\phi_{001}(\boldsymbol{r})\hat{\boldsymbol{e}}_2 + \frac{1-\alpha\omega z^2}{z}\phi_{001}(\boldsymbol{r})\hat{\boldsymbol{e}}_3, \tag{5.68}$$

and Laplacians

$$\nabla^2\phi_{000}(\boldsymbol{r}) = [-3\alpha\omega + (\alpha\omega)^2(x^2+y^2+z^2)]\phi_{000}(\boldsymbol{r}) \tag{5.69}$$
$$\nabla^2\phi_{100}(\boldsymbol{r}) = [-5\alpha\omega + (\alpha\omega)^2(x^2+y^2+z^2)]\phi_{100}(\boldsymbol{r}) \tag{5.70}$$
$$\nabla^2\phi_{010}(\boldsymbol{r}) = [-5\alpha\omega + (\alpha\omega)^2(x^2+y^2+z^2)]\phi_{010}(\boldsymbol{r}) \tag{5.71}$$
$$\nabla^2\phi_{001}(\boldsymbol{r}) = [-5\alpha\omega + (\alpha\omega)^2(x^2+y^2+z^2)]\phi_{001}(\boldsymbol{r}). \tag{5.72}$$

5.5 The zero variance property of the VMC method

Because a Slater determinant is an exact eigenfunction of the Hamiltonian when no correlation is included, the variance of the local energy computed ignoring the Jastrow and the two-body potential should be zero, according to the zero variance property discussed under Eq. (3.13).

Chapter 6

Software design and implementation

The use of compiled low level languages such as Fortran and C is a deeply-rooted tradition among computational scientists and physicists for doing numerical simulations requiring high performance. However, the increased demand for more flexibility has motivated the development of object oriented languages such as C++ and Fortran 95. The use of encapsulation, polymorphism and inheritance allows for setting up source codes into a public interface, and a private implementation of that interface. The style includes overloading of standard operators so that they have an appropriate behaviour according to the context and creating subclasses that are specializations of their parents. Big projects are split into several classes communicating with each other. This has several advantages, in particular for debugging, reuse, maintenance and extension of the code. These ideas will be used in the rest of the chapter for designing and implementing a Quantum Variational Monte Carlo simulator.

6.1 Structuring a software for QVMC in close shell systems

When design software one should take into account the efficient usage of memory, the possibility for future extensions (flexibility) and software structure (easy to understand for other programmers and the final users) [?, ?].

In general, several approaches exist in the design of software, one of them is the use of prototypes. The high level programming language Python[1] has been used in this thesis with this in mind. In essence what a prototype does is to help the designer exploring alternatives, making performance tests, and modifying design strategies.

6.1.1 Implementing a prototype in Python

We start implementing the Quantum Variational Monte Carlo algorithm from chapter 3. To do so, we implement a Vmc class which serves to administrate the simulation, setting

[1] Python is an interpreted object-oriented language that shares with Matlab many of its characteristics, but which is much more powerful and flexible when equipped with tools for numerical simulations and visualization. Because Python was designed to be extended with legacy code for efficiency, it is easier to interface it with software written in C++, C and fortran than in other environments. A balance between computational efficienty, to get fast codes, and programming efficiency, is preferred.

the parameters of the simulation, creating objects related to the algorithm, and running it, as shown below.

```
#Import some packages
...
class VMC():
  def __init__(self, _dim, _np, _charge, ...,_parameters):
     ...
     #Set parameters and create objects for the simulation
     ...
     particle  = Particle(_dim, _np, _step)
     psi       = Psi(_np, _dim, _parameters)
     energy    = Energy(_dim, _np, particle, psi, _nVar, _ncycles, _charge)
     self.mc = MonteCarlo(psi, _nVar, _ncycles, _dim, _np, particle, energy)


  def doVariationalLoop(self):
     if (self.mcMethod=='BF'):
       for var in xrange(nVar):
          self.mc.doMonteCarloBruteForce()
          self.mc.psi.updateVariationalParameters()
     else:
       for var in xrange(nVar):
          self.mc.doMonteCarloImportanceSampling()
          self.mc.psi.updateVariationalParameters()

  ...
```

The information regarding the configuration space can be encapsulated in a class called Particle with the straightforward implementation

```
class Particle():
  def __init__(self, _dim, _np, _step):
    # Initialize matrices for configuration space
     ...

  def acceptMove(self, i):
    for j in xrange(dim):
      self.r_old[i,j] = self.r_new[i,j]

  def resetPosition(self, i):
    for k in xrange(self.np):
      if(k != i):
        for j in xrange(self.dim):
          self.r_new[k,j] = self.r_old[k,j]

  def setTrialPositionsBF(self):
    self.r_old = self.step*random.uniform(-0.5,0.5,size=np*dim).reshape(np,dim)

  def setTrialPositionsIS(self):
    for i in xrange(self.np):
      for j in xrange(self.dim):
        self.r_old[i,j] = random.normal()*sqrt(self.step) # step=dt
```

```
  def updatePosition(self, i):
    #Note use of vectorized loop
    self.r_new[i,0:dim]=self.r_old[i,0:dim] + self.step*(random.random()-0.5)
    self.resetPosition(i)
```

In the class MonteCarlo we implement a function equilibrate() meant to deal with the equilibration of the Markov chain, that is to reach the most likely state of the system before we start with the production part of the simulation. The kind of sampling with which to run the simulation can be choosen from the Vmc class.

```
class MonteCarlo():
  def __init__(self, psi, ..., _ncycles, ..., particle, energy):
    #Set data and member functions
    ...

  def doMonteCarloImportanceSampling(self):
    ...
    #Matrices storing the quantum force terms
    Fold = zeros(np*dim).reshape(np, dim)
    Fnew = zeros(np*dim).reshape(np, dim)

    #Reset energies
    self.energy.resetEnergies()
    deltaEnergy = 0.0
    ...
    #Set initial trial positions (space configuration)
    self.particle.setTrialPositionsIS()

    #Reach the most likely state
    self.equilibrate()

    #Evaluate the trial wave function for the current positions
    wfold = getPsiTrial(r_old)

    #Evaluate the quantum force at the current positions
    Fold = self.psi.getQuantumForce(r_old, wfold)

    for c in xrange(1, self.ncycles+1, 1):  # Monte Carlo loop
      for i in xrange(np):                  # Loop over particles
        for j in xrange(dim):               # Loop over coordinates
          #Suggest a trial move
          r_new[i,j] = r_old[i,j] + random.normal()*sqrt(dt) + Fold[i,j]*dt*D

        #Reset the position of the particles, but the ones containing i as its first index
        resetPostion(i)

        #Evaluate the trial wave function at the suggested position
        wfnew = self.psi.getPsiTrial(r_new)

        #Evaluate the quantum force for particles at the suggested position
        Fnew = self.psi.getQuantumForce(r_new, wfnew)

        #Compute the Green's function
        greensFunction = 0.0

        for j in xrange(dim):
          greensFunction += 0.5*(Fold[i,j] + Fnew[i,j]) \
                            *(D*dt*0.5*(Fold[i,j] - Fnew[i,j]) - r_new[i,j] + r_old[i,j])

        greensFunction = exp(greensFunction)
```

```
        acceptanceRatio = greensFunction*wfnew*wfnew/wfold/wfold

        #Do the metropolis/hasting test
        if(random.random() <= acceptanceRatio):
```

```
            for j in xrange(dim):
              r_old[i,j] = r_new[i,j]
              Fold[i,j] = Fnew[i,j]

            #Update the wave function
            wfold = wfnew

        #Compute local energy and its cumulants
        self.energy.getLocalEnergy(wfold)
      self.energy.computeEnergyExpectationValue()
      self.energy.resetEnergies()
```

To encapsulate information concerning the trial wave function and energy we create the Psi and Energy classes, respectively.

```
class Psi:
  def __init__(self, _np, _dim, _parameters):
    ...
    self.cusp = zeros((_np*(_np-1)/2))   # Cusp factors
    self.setCusp()

  def setCusp(self):
    ...
    if(self.np==2):
      self.cusp[0] = 0.5
    else:
      for i in xrange(size):
        self.cusp[i] = 0.5

      self.cusp[0] = 0.5
      self.cusp[5] = 0.5

  #Define orbitals (single particle wave functions)
  def phi1s(self, rij):
    return exp(-self.parameters[0]*rij)

  def phi2s(self, rij):
    return (1.0 - self.parameters[0]*rij/2.0)*exp(-self.parameters[0]*rij/2.0)

  def getPsiTrial(self, r):
    return self.getModelWaveFunctionHe(r)*self.getCorrelationFactor(r)

  def getCorrelationFactor(self, r):
    correlation = 1.0
    for i in xrange(self.np-1):
      for j in xrange(i+1, self.np):
        idx = i*(2*self.np - i-1)/2 - i + j - 1

    r_ij = 0.0
    for k in xrange(self.dim):
      r_ij += (r[i,k]-r[j,k])*(r[i,k]-r[j,k])
    r_ij=sqrt(r_ij)

    correlation *= exp(self.cusp[idx]*r_ij/(1.0 + self.parameters[1]*r_ij))
    return correlation
```

```
  #Set the Slater determinant part of Be
  def getModelWaveFunctionBe(self, r):
    argument = zeros((self.np))
    wf = 0.0
```

```
    ...
    psi1s = self.phi1s #Shortcut, convenient in for-loops
    ...
    for i in xrange(self.np):
      argument[i] = 0.0
      r_singleParticle = 0.0
        for j in xrange(self.dim):
          r_singleParticle += r[i,j]*r[i,j]

      argument[i] = sqrt(r_singleParticle)

    wf = (psi1s(argument[0])*psi2s(argument[1]) \
        -psi1s(argument[1])*psi2s(argument[0])) \
        *(psi1s(argument[2])*psi2s(argument[3]) \
        -psi1s(argument[3])*psi2s(argument[2]));

    return wf

  #Set the Slater determinant part of He
  def getModelWaveFunctionHe(self, r):
    argument = 0.0
    for i in xrange(self.np):
      r_singleParticle = 0.0
      for j in xrange(self.dim):
        r_singleParticle += r[i,j]*r[i,j]
      argument += sqrt(r_singleParticle)

    return exp(-argument*self.parameters[0])

  #Compute the quantum force numerically
  def getQuantumForce(self, r, wf):
    ...
    for i in xrange(np):
      for j in xrange(dim):
            r_plus[i,j] = r[i,j] + h
        r_minus[i,j] = r[i,j] - h
        wfminus = self.getPsiTrial(r_minus)
        wfplus = self.getPsiTrial(r_plus)

        quantumForce[i,j] = (wfplus - wfminus)/wf/h

        r_plus[i,j] = r[i,j]
        r_minus[i,j] = r[i,j]

    return quantumForce
...
```

Class Energy is also equipped with some functions to do the statistical analysis of uncorrelated data.

```
class Energy:
  def __init__(self, dim, np, particle, psi, maxVar, ncycles, charge):
    self.cumEnergy = zeros(maxVar)      #Cumulant for energy
    self.cumEnergy2= zeros(maxVar)   #Cumulant for energy squared
    ...
```

```
  def getLocalEnergy(self, wfold):
    EL     = self.getKineticEnergy(wfold) + self.getPotentialEnergy()
    self.E  += EL
    self.E2 += EL*EL
```

```
  ...

  def getPotentialEnergy(self):
      ...
      PE = 0.0

      # Contribution from electron-proton potential
      for i in xrange(np):
          r_single_particle = 0.0
          for j in xrange(dim):
              r_single_particle += r_old[i,j]*r_old[i,j]
      PE -= charge/sqrt(r_single_particle)

      # Contribution from electron-electron potential
      for i in xrange(0, np-1):
          for j in xrange(i+1, np):
              r_12 = 0.0
              for k in xrange(0, dim):
                  r_12 += (r_old[i,k] - r_old[j,k])*(r_old[i,k] - r_old[j,k])
              PE += 1.0/sqrt(r_12)
      return PE

  def getKineticEnergy(self, wfold):
      KE = 0.0            # Initialize kinetic energy
      h = 0.001           #Step for the numerical derivative
      h2= 1000000.        #hbar squared
      ...
      for i in xrange(np):
          for j in xrange(dim):
              r_plus[i,j] = self.particle.r_old[i,j] + h
              r_minus[i,j]= self.particle.r_old[i,j] - h
              wfplus  = self.psi.getPsiTrial(r_plus)
              wfminus = self.psi.getPsiTrial(r_minus)

              # Get the laplacian_wave_function to wave_function ratio
              KE -= (wfminus + wfplus - 2.0*wfold)

              r_plus[i,j]=self.particle.r_old[i,j].copy()
              r_minus[i,j]  = self.particle.r_old[i,j].copy()

  return 0.5*h2*KE/wfold # include electron mass and hbar squared
```

The calling code is just

```
from VMC import *

#Set parameters of simulation
dim   = 3                 #Number of spatial dimensions
nVar  = 10                #Number of variations for a quick optimization method
ncycles = 10000           #Number of monte Carlo cycles
np  =2              # Number of electrons
charge  =2.0        # Nuclear charge
...
vmc = VMC(dim, np, charge, mcMethod, ncycles, step, nVar, parameters)
vmc.doVariationalLoop()
vmc.mc.energy.printResults()
```

The source code[2] above is relatively straightforward to implement, besides being short (600 lines). The syntax used is clear, compact and close to the way we formulate the expressions

[2]For details, we refer to the CD included with this thesis under QVMC/Python.

mathematically. The language is easy to learn and one can develop a program and modify it easily. The dual nature of Python of being both an object oriented and scripting language is convenient in structuring the problem such that we can split it into smaller parts, while testing quickly the results (output) against analytical or experimental data. A limitation, especially in problems managing a high number of degrees of freedom, as happens in quantum mechanics, is its speed as shown in figures 7.3 to 7.5.

Computing the ground energy of a He atom (2 electrons) using 10000 Monte Carlo cycles and five parameter variations with this Python simulator takes about 1.5 minutes on a Pentium-4 laptop running at 2.4 GHz. On the other hand, attempting to do the same for a Be atom (4 electrons) with only 5×10^4 Monte Carlo cycles (not enough to get statistically acceptable results for Be) takes more than one hour. Several alternatives exists for being able to simulate bigger systems than He. Three of them are exposed here and explored in the following:

1. Keeping the software structure as above, but implement all the classes in C++ (or another low-level programming language).

2. Integrate Python and C++ in a way that the portions of the software that requires high-performance computing considerations can be implemented in C++. The administrative parts are then written in Python.

3. Exploting the object-oriented characteristics of C++ for developing an efficient, flexible and easy to use software (optimized algorithm).

6.1.2 Moving the prototype to a low level programming language

The implementation of the simulator in C++ follows more or less the recipe above. Here, however, we separate the potential energy from the Hamiltonian by introducing a Potential class, which becomes an object member of the class Energy. The use of dynamic memory to manipulate data structures is usually a source of bugs, as C++ does not have automatic garbage collection as Python does.

In the present implementation we use a ready made class for manipulating array data structures. MyArray class stores all the array entries in contiguous blocks of memory [?]. This design aims at interfacing Python and C++, as the Python arrays are, also, stored in the same way. The class MyArray supports arrays in one, two and three dimensions. [?]. Because it is a template class, it can be reused easily with any type. Moreover, changing its behaviour is easily done in only one place and to free of memory is made in its destructor [?].

6.1.3 Mixing Python and C++

There are several ways one can integrate Python with C/C++. Two of them are called extending and embedding, respectively. The first one, extending, involves creating a wrapper for C++ that Python imports, builds, and then can execute. We will focus on the wrapper extension. In the second alternative, embedding, the C++ part is given direct access to the Python interpreter. Python is extended for many different reasons. A developer may want to use an existing C++ library, or port work from an old project into a new Python

development effort.

```
...
template< typename T > class MyArray
{
 public:
  T* A;                    // the data
  int ndim;                // no of dimensions (axis)
  int size[MAXDIM];        // size/length of each dimension
  int length;              // total no of array entries
  T* allocate(int n1);
  T* allocate(int n1, int n2);
  T* allocate(int n1, int n2, int n3);
  void deallocate();
  bool indexOk(int i) const;
  bool indexOk(int i, int j) const;
  bool indexOk(int i, int j, int k) const;

 public:
  MyArray() { A = NULL; length = 0; ndim = 0; }
  MyArray(int n1) { A = allocate(n1); }
  MyArray(int n1, int n2) { A = allocate(n1, n2); }
  MyArray(int n1, int n2, int n3) { A = allocate(n1, n2, n3); }
  MyArray(T* a, int ndim_, int size_[]);
  MyArray(const MyArray<T>& array);
  ~MyArray() { deallocate(); }

  bool redim(int n1);
  bool redim(int n1, int n2);
  bool redim(int n1, int n2, int n3);

  // return the size of the arrays dimensions:
  int shape(int dim) const { return size[dim-1]; }

  // indexing:
  const T& operator()(int i) const;
  T& operator()(int i);
  const T& operator()(int i, int j) const;
  T& operator()(int i, int j);
  const T& operator()(int i, int j, int k) const;
  T& operator()(int i, int j, int k);

  MyArray<T>& operator= (const MyArray<T>& v);

  // return pointers to the data:
  const T* getPtr() const { return A;}
  T* getPtr() { return A; }

   ...
};
```

Before attempting to extend Python, we should identify the parts to be moved to C++. The results of a profile of the prototype discussed in section 6.1.1 are summarized in tables 7.2 and 7.3. It is not surprising that Energy.py, MonteCarlo.py and Psi.py are the most time consuming parts of the algorithm. It is there where the major number of operations and

calls to functions in other classes are carried out. Fortunately, most of the work needed to rewrite these classes to their equivalent C++ part has been already carried out in the last subsection.

The class Vmc.py is responsible of initializing the C++ objects of type Psi, Particle, Potential, Energy and MonteCarlo. Letting Python create the objects has the advantage of introducing automatic garbage collection, reducing the risk implied in managing memory. Memory handling is very often a source of errors in C++. It is also often very difficult to spot such errors.

To access C++ via an extension we write a wrapper. The wrapper acts as a glue between the two languages, converting function arguments from Python into C++, and then returning results from C++ back to Python in a way that Python canunderstand and handle. Fortunately, SWIG (Simplified Wrapper and Interface Generator) [?] does much of the work automatically.

Before trying to implement a wrapper for the present application, one should note that the Psi class takes a generic object of type MyArray<double> as argument in its constructor. It means that we should find the way of converting the variational parameters (a numpy object) into a reference to an object of type MyArray<double>. This kind of conversion is not supported by SWIG and has to be done manually. Doing this for a lot of functions is a hard and error prone job.

```
...
#include "MyArray.h"
...
class Psi{
  private:
    ...
    MyArray<double> cusp;   // Cusp factors in the Jastrow form
    MyArray<double> varPar;  // Variational parameters
    MyArray<double> quantumForce;

  public:
    Psi(MyArray<double>& varParam, int _np, int dim);
    void setParameters(MyArray<double>& varPar);
    double getCorrelationFactor(const MyArray<double>& r);
    double getModelWaveFunctionHe(const MyArray<double>& r);
    MyArray<double>& getQuantumForce(const MyArray<double>& r, double wf);
    ...
};
```

A conversion class Convert specially adapted for MyArray class has already been proposed in [?]. The advantage with it is that we do not need to interface the whole MyArray class. Instead, the conversion class is equiped with static functions for converting a MyArray object to a Numpy array and vice versa. The conversion functions in this class can be called both manually or using SWIG to automatically generate wrapper code. The last option is preferible because the conversion functions has only pointers or reference as input and output data.

```
...
```

```
#include <Python.h>
#include <pyport.h>
#include <numpy/arrayobject.h>
#include "MyArray.h"
...
class Convert
{
 public:
    Convert();
  ~Convert();

    // borrow data:
    PyObject*       my2py (MyArray<double>& a);
    MyArray<double>* py2my (PyObject* a);

    // copy data:
    PyObject*       my2py_copy (MyArray<double>& a);
    MyArray<double>* py2my_copy (PyObject* a);

    // npy_intp to/from int array for array size:
    npy_intp        npy_size[MAXDIM];
    int             int_size[MAXDIM];
    void            set_npy_size(int*      dims, int nd);
    void            set_int_size(npy_intp* dims, int nd);

    // print array:
    void            dump(MyArray<double>& a);

     ...
    #Code necessary to make callbacks
     ...

};
```

With the information above we create the SWIG interface file ext_QVMC.i having the same name as the module [?]

```
%module ext_QVMC
%{
#include "Convert.h"
#include "Energy.h"
#include "MonteCarlo.h"
#include "Psi.h"
#include "Particle.h"
#include "Potential.h"
%}
%include "Convert.h"
%include "Energy.h"
%include "MonteCarlo.h"
%include "Particle.h"
%include "Potential.h"
%include "Psi.h"
```

To build a Python module with extension to C++ we run SWIG with the options -Python and -c++. Running

```
swig -c++ -Python ext_QVMC.i
```

generates a wrapper code in ext_QVMC_wrp.cxx and the Python module _ext_QMVC.py. Assuming that all the sources files are in the same directory, this file has to be compiled

```
c++ -c -O3 *.cpp *.cxx -I/usr/include/Python2.6/
```

and linked to a shared library file with name _ext_QVMC.so

```
c++ -shared -o _ext_QVMC.so *.o
```

The Python VMC.py class calling the C++ code has to be slightly adjusted such that it can use the new extension module ext_QVMC.

```
from math import sqrt  # use sqrt for scalar case
from numpy import*
import string

# Set the path to the extensions
import sys
sys.path.insert(0, './extensions')
import ext_QVMC

class Vmc():
  def __init__(self, _Parameters):
    # Create an object of the 'conversion class' for
    # convert Python-to-C++-to-Python data structure.
    self.convert = ext_QVMC.Convert()

    # Get the paramters of the currrent simulation
    simParameters = _Parameters.getParameters()

    self.nsd       = simParameters[0]
    self.nel       = simParameters[1]
    self.nVar      = simParameters[2]
    self.nmcc      = simParameters[3]
    self.charge    = simParameters[4]
    self.step      = simParameters[5]
    alpha     = simParameters[6]
    beta      = simParameters[7]

    # Set the Variational parameters
    self.varpar = array([alpha, beta])

    # Convert a Python array to a MyArray object
    self.v_p = self.convert.py2my_copy(self.varpar)
```

```
    # Create an wave function object
    self.psi      = ext_QVMC.Psi(self.v_p, self.nel, self.nsd)

    self.particle     = ext_QVMC.Particle(self.nsd, self.nel, self.step)
    self.potential    = ext_QVMC.Potential(self.nsd, self.nel, self.charge)
    self.energy     = ext_QVMC.Energy(self.nsd, self.nel, self.potential, self.psi, self.nVar,
          self.nmcc)
    self.mc    = ext_QVMC.MonteCarlo(self.nmcc, self.nsd, self.nel, 0.001, self.particle, self.psi,
          self.energy)

  def doVariationalLoop(self):
    nVar = self.nVar
    for var in xrange(nVar):
      self.mc.doMonteCarloLoop()

...
```

The caller[3] code in Python resembles the one in the simulator prototype from section 6.1.1, but with an additional class SimParameters.py that is able to read from a conveniently

[3] The function that calls another (callee).

formatted file containing the parameters of the simulation. An example for the Be atom is:

```
Number of spatial dimensions  = 3
Number of electrons  = 4
Number of variations = 10
Number of Monte Carlo cycles   = 100000
Nuclear charge   = 4.0
Step = 0.01
Alpha = 2.765
Beta = 0.6
```

With this small change, the new caller looks more compact and user friendly.

```
import sys

from SimParameters import * #Class in Python encapsulating the parameters
from Vmc import *

# Create an object containing the parameters of the current simulation
simpar = SimParameters('Be.data')

# Create a Variational Monte Carlo simulation
vmc = Vmc(simpar)

vmc.doVariationalLoop()
vmc.energy.doStatistics("resultsBe.data", 1.0)
```

6.2 Developing a robust QVMC simulator in pure C++

The rest of this chapter is dedicated to describe how one can take advantage of the object oriented characteristic of C++ to create a flexible and robust QVMC simulator. In order to reduce the computational cost of evaluating quantities associated with the trial wave function, we implement instead the algorithms discussed in chapter (4). This requires that we add some extra functionality. We discuss these additional functionalities in the rest of this chapter[4].

Parameters of the system

The SimParameters class encapsulates[5] the parameters of the simulation. It creates an object of a class especially designed to read files in ascii format containing comments as well. In this way the user can easyly know what kind of information the software requires. The function readSimParameters() reads the parameters and loads them as data members. Moreover, functions to return each of them are provided.

[4]The corresponding source codes can be found in the CD following this thesis in QVMC/parallel/. Also, an extensive documentation has been prepared and can be compiled by running doxygen pyVMC in a terminal under the directory of the sources codes. It will generate the documentation in html (QVMC/-parallel/doc).

[5]This kind of encapsulation reduces the number of parametersthat are sent to the various functions. It reduces also the possibility of making mistakes when swapping them. Furthemore, the localization of bugs in the code can become easier, because the whole program is divided into small parts.

```
sys = 2DQDot2e          # Two dimensional quantum dot with two electrons.
nsd = 2                 # Number of spatial dimensions.
nel = 2                 # Number of electrons.
potpar = 2.0            # Potential parameter (omega).
lambda = 1              # Confinament parameter in Coulomb potential.
nmc = 1.000000e+07      # Number of Monte Carlo cycles.
nth = 1.000000e+06      # Number of equilibration steps.
dt = 1.0000e-06         # Step size.
nvp = 2                 # Number of Variational parameters.
alpha = 0.75            # Variational parameter.
beta = 0.16             # Variational parameter.
enableBlocking = no     # Blocking option off.
enableCorrelation = yes # Correlation part included.
```

Initializing and setting in operation a simulation

We declare a class VmcSimulation.h having a pointer to the SimParameters class. The parameters of the simulation are then used in setting a Variational Monte Carlo simulator in operation. A function setInitialConditions() initializes the space configuration and the trial wave function (includes the gradients and Laplacians of the single wave functions and the Jastrow correlation function). By declaring the virtual functions setWaveFunction() and setHamiltonian to be called by setPhysicalSystem(), the software is able to handle atomic systems and quantum dots, that is either non-interacting or interacting electrons in harmomic-oscillator ike traps. The particular implementation of these functions takes place in the subclass VmcApp depending of the physical system treated.

```
void VmcApp::setHamiltonian(){
  //Set the potential form and add it to the kinetic energy term
  potential = new CoulombAtom(simParameters);

  energy = new Energy(simParameters, potential, psi, parallelizer);
}
```

By calling setHamiltonian(), the application creates a Potential object which is sent as a pointer to the class Energy. In the same function, we call setWaveFunction() to set the Jastrow and the SlaterDeterminant objects, which are used as argument to initialize the WaveFnc class.

At the time the VmcSimulation is created, we construct a RandomNumberGenerator() object that deals with (pseudo) random numbers and uses them together with other parameters to create a MonteCarlo object implementing the equilibration and a sampling algorithm as the one shown in figure 3.1. The Monte Carlo for-loop is initiated by VmcSimulation::runSimulation(). The current code does not provide an interface for the Monte Carlo class, but it should be straightforward to create one with the following functions declared as virtual: doMonteCarloLoop(), suggestMove() and doMetropolisTest().

```
...
class VmcSimulation{
  protected:
    ...
    Energy *energy;        // Hamiltonian
    MonteCarlo* mc;        // Monte Carlo method
    Optimizer *optimizer; // Optimization algorithm
     Parallelizer  *para;   // Encapsulate MPI
    ...
```

```
    Potential *potential;
    RandomGenerator *random;
    RelativeDistance *relativeDistance;    // Vectorial and scalar relative distances
    SimParameters *simpar;                 // Parameters of simulation
    SingleStateWaveFuncs **singleStates;   // Single state wave functions
    Slater *slater;
    WaveFnc *psi;                          // Trial wave function

    int numprocs;
    int myRank;
    int rankRoot;
    ...
  public:
    VmcSimulation(SimParameters *_simpa, int argc, char** argv);
    virtual ~VmcSimulation();
    void dumpResults(string resultsFile);
    void runSimulation();
    void setPhysicalSystem(){}
    void setInitialConditions(long& idum);
    void setPhysicalSystem(long& idum);

    virtual void optimizeWaveFunction(MyArray<double>& p, ...);
    virtual void setRandomGenerator();
    virtual void setHamiltonian(){}
    virtual void setWaveFunction(){}
};
```

To create a trial wave function object, we first instanciate singleStateWaveFuncs and JastrowForm and use them to create the SlaterDeterminant and the Jastrow objects. Function objects are used to encapsulate the forms of the Jastrow function[6]. For the single particle wave functions we create an array of pointers to objects of type SingleSateWaveFnc.

```
void VmcApp::setWaveFunction(){

  // Set single state wave functions (spin orbitals in atoms)
  singleStates     = new SingleStateWaveFuncs*[2];
  singleStates[0] = new phi1s(nsd, varPar);
  singleStates[1] = new phi2s(nsd, varPar);

  // set Slater determinant
  slater   = new Slater(simpar, varPar, singleStates);

  if(enableCorrelation == true){
   // set correlation function
   setJastrow();
   // set Slater Jastrow trial wave function
   psi = new SlaterJastrow(slater, jastrow, simpar, varPar, relativeDistance);

  }else{
   // set Slater determinant as trial wave function
   psi = new SlaterNOJastrow(slater, simpar, varPar, relativeDistance);
  }
}
```

[6] In this case in particular we use a function object, an instance of an class that behaves like a function by virtue of its function call operator () are those that can be called as if they were a function.

Class structure of the trial wave function

The definition of the trial wave function, single particle wave functions, the Jastrow correlation function, etc, has to be flexible enough in order to easily be provided by the user for different physical systems and computational methods (numerical or analytical). This kind of behaviour is known as polymorphism and it is achieved with the use of interfaces, that is classes that define a general behaviour without specifying it. Examples are WaveFnc, Potential.h and the interface for the single particle wave functions SingleStateWaveFuncs.h.

```
...
class SingleStateWaveFuncs{
  ...
 public:
    ...
    virtual double evaluate(int p, const MyArray<double>& r)=0;
    virtual MyArray<double>& getStateGradient(int p, const MyArray<double>& r)=0;
    virtual double getSecondDerivative(int p, const MyArray<double>& r)=0;
    virtual double getParametricDerivative(int p, const MyArray<double>& r)=0;
    ...
 };
  ...
```

A convenient class derived from the interface above is SlaterNOJastrow. Because it does not include a Jastrow correlation factor, it can be used both for computing energies of harmonic oscillators and in the validitation of results derived from simulations with trial wave function including only the Slater determinant part. What one should expect in the last case is that experiments carried out in this way reproduce some of the analytical results derived in chapter 5. Moreover, because a Slater determinant is an exact wave function when no correlation is included, it is expected that the variance resulting from the experiments is zero.

```
class WaveFnc{
 public:
    virtual ~WaveFnc(){}
    virtual double getLapPsiRatio(int i)=0;
    virtual double getPsiPsiRatio(int p, const MyArray<double>& rnew)=0;
    virtual MyArray<double>& getVariationalParameters()=0;
    virtual MyArray<double>& getParametricDerivative(const MyArray<double>& r)=0;
    virtual void getQuantumForce(int p, MyArray<double>& F)=0;
    virtual void resetJastrow(int i)=0;
    virtual void getTrialRow(int p, const MyArray<double>& rnew)=0;
    virtual void getTrialQuantumForce(int p, MyArray<double>& Fnew, ...) = 0;
    virtual void  initializePsi (const MyArray<double>& rold)=0;
    virtual void updatePsi(int k, const MyArray<double>& rnew)=0;
    virtual void setVariationalParameters(MyArray<double>& varPar)=0;
};
```

On the other hand, the class SlaterJastrow has SlaterDeterminant and Jastrow as data members. Thus, the functionalities of SlaterJastrow can be carried out independently on each of the member classes and sometimes, as in the case of computing the total quantum force, summed up in the calling class, as suggested by table 6.2. Furthermore, since the the Slater determinant equals the product of two determinants, we create a SlaterDeterminant class containing two pointers, pointing to a SlaterMatrix whose entries are single particle wave

Class	Function	Implemented equation
SlaterMatrix	getDetDetRatio()	4.9
	getGradDetRatio()	4.11-4.12
	getLapDetRatio()	4.17
	updateInverse()	4.18
Jastrow	getJasJasRatio()	4.7
	getGradJasRatio()	4.31
	getLapJasRatio()	4.38
SlaterJastrow	getPsiPsiRatio()	4.3
	getGradPsiRatio()	4.10
	getLapPsiRatio()	4.16

Table 6.1: Some functions implemented in relation with the trial wave function.

functions for electrons with spin up and down, respectively[7]. Besides the functionality shown in table 6.2, this class has some arrays for storing $\phi_j(\boldsymbol{x}_i^{cur})$, $\phi_j(\boldsymbol{x}_i^{trial})$, $\boldsymbol{\nabla}_i\phi_j(\boldsymbol{x}_i^{cur})$, $\boldsymbol{\nabla}_i\phi_j(\boldsymbol{x}_i^{new})$ and $\nabla_i^2(\boldsymbol{x}_i)$.

```
class SlaterJastrow: public WaveFnc
{
  private:
    SlaterDeterminant *slater;
    Jastrow           *jastrow;
    MyVector<double> varPar;   // variational parametes
     ...
  public:
     ...
    SlaterJastrow(Slater *_slt, Jastrow *_jtr ,...) ;
    void getPsiPsiRatio (...) ;
    double getLapPsiRatio(...) ;
    MyArray<double>& getGradPsiRatio(...);
    void setVariationalParameters(MyArray<double>& _varPar);
     ...
}
```

Because the operations with Slater matrices are done by moving one particle (one row) at the time, the SlaterDeterminant class needs to know which matrix it has to operate on. Therefore, during the initialization process we define an array of pointers to SlaterMatrix objects called curSM (for current Slater matrix) whose size equals the number of electrons in the system. Half of its entries point to a Slater matrix with spin up and the rest to the one with spin down. This can be used to decide which matrix to compute. Doing so is more efficient than using an if-test each time an electron is moved.

For the Jastrow we create upper-triangular matrices for the scalar quantities $g(r_{ij})$ and $\nabla^2 g(r_{ij})$ (two per term) , and two for the vectorial term $\boldsymbol{\nabla} g(r_{ij})$, all of them evaluated at the current and new positions. A similar storage scheme is used for the matrix a_{ij} containing the electron-electron cusp condition factors given in Eqs. (2.28) and (2.27). The access to the scalar and vectorial (interelectronic) distances is carried out by means of a

[7] The way the spin is set up is by placing half of electrons in each of the matrices.

pointer to a RelativeDistance object[8]. Each time a move is suggested, accepted or rejected, the matrices in Jastrow have to be updated following the algorithms described in sections 4.6 to 4.9. Similar comments apply to the class RelativeDistance.

Each scalar and vectorial matrix has a size $N(N-1)/2$ and $N(N-1)/2 \times nsd$, respectively, with N the number of electrons in the system and nsd being the number of spatial dimensions. In the implementation, the matrices are stored as one-dimensional arrays in contiguous blocks of memory, using the generic class MyArray[9]. MyArray supports arrays in one, two and three dimensions. The mapping between the entry (i, j) in the upper triangular matrix and the index idx in the 1D-array is given by the equation $idx = i(2N-i-1)/2-i+j-1$, with $0 \leq i < N-1$ and $i+1 \leq j < N$.

Computing the energy, parallelizing with MPI and optimizing with a quasi-Newton method

The collection of samples and evaluation of the energy happens in the Energy class, which has a pointer to a Potential class with some functions declared virtual.

```
class Potential{
  public:
    virtual ~Potential(){};
    virtual double getPotentialEnergy(const MyArray<double>& r){
      ...
    };
};
```

In this way, what we need to do for implementing new potential forms is to inherit from it and overriding its functions in subclases. The creation of a new object and its use has the form

```
  // Create a potential form for atoms
  Potential *coulombAt = new CoulombAtoms(...);

  // Get the potential energy
  coulombsAt->getPotentialEnergy(...);
```

On the other hand, the Energy class does not care about the kind of potential gotten in its constructor. Therefore we do not need to modify it when the potential form changes.

```
....
class  Parallelizer {
  private:
    int numprocs;        // Number of procesors
    int myRank;          // Number (label) of this procesor
    int rankRoot;        // Number (label) of the root procesor

    #if PARALLEL
      MPI_Status status;
    #endif
```

```
  public:
```

[8] Information on scalar and vectorial interelectronic relative distances are also stored in upper-triangular matrices by the RelativeDistance class.

[9] Arrays will be read and written a large number of times during the simulation, so the access should be fast as possible. It is reached by storing all the array entries in contiguous blocks [?].

```
    Parallelizer (int argc, char** argv);
    ~Parallelizer (){mpi_final();}
    void mpi_final(); // Finalize MPI
    int read_MPI_numprocs() const;
    int read_MPI_myRank() const;
    int read_MPI_rankRoot() const;
};// end of Parallelizer class
```

```
...
#include "Parallelizer.h"
...
class Energy{
  private:
    ... //
    double eMean; // Mean energy
    double eSum; // Cumulative energy
    double eSqSum; // Cumulative energy squared

    #if OPTIMIZER_ON
      MyArray<double> daMean; // Mean value of dPhi/dAlpha
      MyArray<double> daSum; // Cumulative dPhi/dAlpha
      MyArray<double> ELdaMean; // Mean of local energy times dPhi/dAlpha
      MyArray<double> ELdaSum; // Cumulative local energy times dPhi/dAlpha
    #endif

    ofstream ofile, blockofile;
    int totalNMC;

    #if PARALLEL
      double totalESum;
      double totalESqSum;
      MyArray<double> allEnergies; // Save all the local energies
      MPI_Status status;
    #endif

    Potential *potential;
    WaveFnc *psi;
    int numprocs, rankRoot, myRank;
    Parallelizer *para;

  public:
    Energy(SimParameters *simpar, Potential *potential, WaveFnc *psi,Parallelizer * para);
    ~Energy();
    void computeExpectationValues(int mcCycles);
    void computeLocalEnergy(int cycle, const MyArray<double>& rnew);
    void dumpResults(const char* resultsFile);
    double getEnergy();
    MyArray<double>& getLocalParametricDerivative(const MyArray<double>& r);
    void getParametricDerivatives(MyArray<double>& dEda);
    void resetCumulants();
    void sum_dPda(double E_Local, const MyArray<double>& r);
    void updateCumulants(double E_Local, const MyArray<double>& r);
};
...
```

Furthemore, the class defined by Parallelizer.h, appearing on the top of Energy.h and VmcSimulation.h files encapsulates information to handle MPI in a straighforward way.

When working in parallel (by setting PARALLEL=1 during the compilation time) each processor executes the QVMC algorithm separately. The energy data for each processor is accumulated in its own eSum member. To get the expectation value from all the computations we call Energy::dumpResults() with PARALLEL=1 directive. Because Energy.h has access to MPI by means of Parallelizer.h, the only execution needed is

```
// Collect data in total averages
MPI_Reduce(&eSum, &totalESum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(&eSqSum, &totalESqSum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

inside this function, which can be translated as: "Collect the cumulative energies from all the processors, sum them and divide by the total number of processors to get the mean energy". When the statistical data are assumed to be uncorrelated, the mean energy, variance and error can be computed in

```
if(myRank==rankRoot){
  ...
  energy    = totalESum/totalNMC;
  variance  = totalESqSum/totalNMC - energy*energy;
  error     = sqrt(variance/(totalNMC - 1));
  ...
}
```

The way the energy is computed above gives only an estimate of the real variance, and it serves only as a reference to be compared with the real variance we obtain when using the blocking technique. The data necessary to do it are obtained from the same function by

```
...
// Set file name for this rank
 ostringstream ost;
 ost << "blocks_rank" << myRank << ".dat";

 // Open file for writting in binary format
 blockofile.open(ost.str().c_str(), ios::out | ios::binary);

 // Write to file
 blockofile.write((char*)(allEnergies.A), ncycles*sizeof(double));

 blockofile.close(); // close file
```

The Energy class is also equipped with functions for computing and returning the expectation value of the energy and its derivative with respect to the variational parameters. This information is used by the class Optimizer to optimize the parameters of the wave function that hopefully will lead to the computation of the minimum variational energy. The process of optimizing before attempting to run the production stage of a simulation is illustred below:

```
#include "SimParameters.h"
#include "VmcSpecificApplication.h"
#include "VmcSimulation.h"
```

```
...
int main(int argc, char *argv[]){
  // Set some trial variational parameters
   ...
  // Declarations for the optimization
  int iter; double minEnergy=0.0;

  // Set the name of the files for the parameters
  // of this simulation and the output
   ...
  // Encapsulate the parameters of this simulation in an object
  SimParameters *simpa = new SimParameters();
   ...
  // Create a "simulation"
  VmcSimulation *vmcatoms = new VmcAtoms(simpa,varParam, argc, argv);

  // Set the Hamiltonian and the trial wave function corresponding
  // to atoms. Initialize space configuration
  vmcatoms->setPhysicalSystem(idum);

  // Run the optimizer
  vmcatoms->OptimizeWaveFunction(varParam, iter, minEnergy);

  // Output the results of the optimization
   ...
  return 0;
} // end main
```

And the class implementing the optimizer is:

```
#include "Energy.h"
#include "MonteCarloIS.h"
#include "WaveFnc.h"
...
class Optimizer{
  private:
    int n;                    // Number of variational parameters
    double gtol;              // Tolerance for convergence
    Energy *pt2energy;
    MonteCarloIS *pt2mc;
    WaveFnc *pt2psi;
     ...
  public:
    Optimizer(int _n, double _gtol, Energy *_pt2energy,
        MonteCarloIS *_pt2mc, WaveFnc *_pt2psi):
        n(_n), gtol(_gtol), pt2energy(_pt2energy),
        pt2mc(_pt2mc), pt2psi(_pt2psi){
       ...
    }

    // Function to be called from VmcApp::OptimizeWaveFunction(...).
    void run(MyArray<double>& p, int& _iter, double& _fret){
      dfpmin(p, _iter, _fret); _iter = _iter+1;
    }
```

```
    // Gets the expectation value of the energy for a set of variational
    // parameters in the current Monte Carlo loop.
```

```
    double func(MyArray<double>& p){
      pt2psi->setVariationalParameters(x);
      pt2mc->doMonteCarloLoop();
      return pt2energy->getEnergy();
    }

    // Sets the vector gradient g=df[1...n] of the function returned by
    // func() evaluated using the input parameters p.
    void dfunc(MyArray<double>& p, MyArray<double>& g){
      pt2energy->getParametricDerivatives(x, g);
    }

    // Given a starting set of parameters p[1...n] performs a Fletcher-
    // Reeves-Polak-Ribiere minimization on func().
    void dfpmin(MyArray<double>& p, int& _iter, double& _fret);
   ...
};
```

Running a simulation

The body of the main method for the productive phase has almost the same form. One needs only to run a simulation with the optimal parameters found in the step above.

```
...
int main(int argc, char** argv){
  ...
  // Set some trial variational parameters
  ...
  // Encapsulate the parameters of this simulation in an object
  SimParameters *simpa = new SimParameters();
  ...
  // Create a "simulation"
  VmcSimulation *vmcatoms = new VmcAtoms(simpa,varParam, argc, argv);

  // Initialize the simulation
  vmcatoms->setPhysicalSystem(idum);

  // Start the simulation
  vmcatoms->runSimulation();

  // Print results to file
  vmcatoms->dumpResults(resultsFile);
  ...
  return 0;
}
```

An overview of the classes and some of the functions to be implemented by the end user of the simulator are summarized in table 6.2. Because of limitations of space and time, it is impossible to make a more detailed description of the simulator. The interested reader is, therefore, invited to consult the documentation of the source code following the appended CD of this thesis.

Class	Function
-	main()
SingleStateWaveFncApp	evaluate(...) getStateGradient(...) getLaplacian(...) ...
PotentialApp	getPotentialEnergy(...) getOneBodyPotential(...) ...
VmcApp	setHamiltonian(...) setWaveFunction(...) ...

Table 6.2: Classes and functions to be implemented by the user of the software.

Chapter 7

Computational experiments and discussion

This chapter is divided into two parts. The first part deals with the evaluation of the computational performance of the high level programming language Python. Here, we discuss the task of evaluating the ground state energy of a quantum mechanical system by the Quantum Variational Monte Carlo (QVMC) method. In the second part, we use an optimized version of the QVMC simulator to evaluate the ground state energies of He, Be and two dimensional quantum dots with two and six electrons.

7.1 Comparing high to low level programming languages in a QVMC application

A prototype in Python capable of computing the ground state energy of the helium (He) atom by the QVMC method with importance sampling was developed and discussed in section 6.1.1. Results of simulations for a similar implementation in C++ are shown figure 7.1. The experiment was carried out with 10 millions Monte Carlo cycles using importance sampling.

To determine the ground state energy of the system, the variational parameter α was set equal to the effective nuclear charge[1]. Then, 40 variations of β in the range $0.1-1.2$ were carried out. The ground state energy was estimated to be $E_0 = -2.8731096 \pm 1.9705 \times 10^{-4}\, au$ with $\beta_{optimum} \approx 0.45$ by selecting the point with the lowest variance. The given value should be compared with table 7.1. We should pay attention to the fact that the lowest energy does not necessarily means that we obtain lowest variance as well, as indicated by the error bars appearing in the box inside figure 7.1.

The time taken by the C++ code to run this experiment was about 20 minutes. Doing the same experiment in Python would take more than a week. However, a test with only one million Monte Carlo cycles and 20 variations of the variational parameter β was carried out. Figure 7.2 shows that Python can compute energies for a single Helium atom in the range of the output from the C++ implementation, but with a higher variance due to the lower number of Monte Carlo cycles used in the simulation, and possibly because of the kind of random number generator used.

[1] The effective charge is the net positive charge experienced by an electron in a multi-electron atom. Because of the presence of other electrons, the ones in higher orbitals do not experience the full nuclear charge due to the shielding effect. The value $Z_{eff} = 1.6785$ is taken from Clementi(1963) [?]

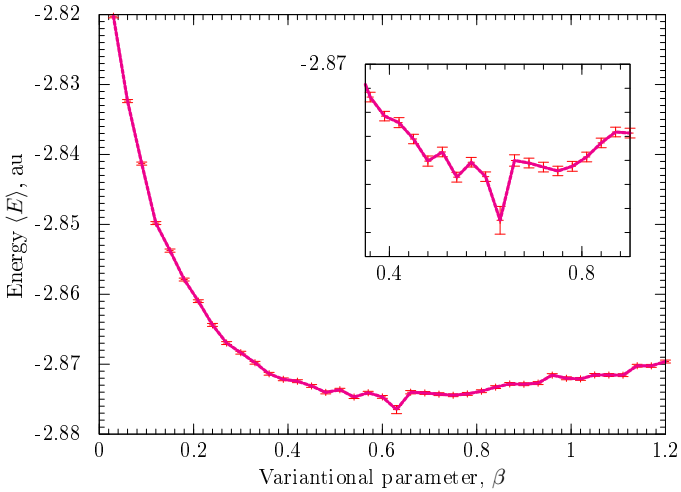

Figure 7.1: Energies for the He atom as a function of the variational parameter β executing a plain C++ simulator. The experiment was carried out with 1×10^7 cycles, $dt = 0.1$ and $\alpha = 1.6785$.

Method	$\langle E \rangle$, au
Hartree-Fock	-2.8617
DFT	-2.83
Exact	-2.9037

Table 7.1: Ground state energies for He atom obtained from reference [?].

7.1.1 Comparative performance of Python vs C++

A question of interest in computational physics is the performance[2] of the simulators, both because of the quantity of degrees of freedom involved in quantum mechanics problems and for the need of identifying bottlenecks of codes. The time used by Python and C++ to execute the QVMC simulator for He atom was compared, measured for a set of runs with the number of Monte Carlo cycles varying from 10000 to 100000 with the rest of the paramaters kept fixed. The results are summarized in figures 7.3 to 7.5.

Figures 7.3 and 7.4 show a clear delay in execution time for Python in relation to the corresponding implementation in C++. Such a behaviour follows from the fact that C++ is a compiled language. The delay in time of Python with respect to C++ is, according to figure 7.5, of about 18×10^{-4} seconds per Monte Carlo cycle for the current application.

7.1.2 Profiling Python to detect bottlenecks

Profilers are useful for creating a ranked list of the computing time spent in the various functions of a program. In order to identify bottlenecks in the pure Python code implemented

[2] All the serial experiments reported in this thesis were carried out on a laptop with a processor Pentium IV running at 2.4 GHz (performance mode). Moreover, the plain C++ code in this particular experiment was compiled using optimization -O3.

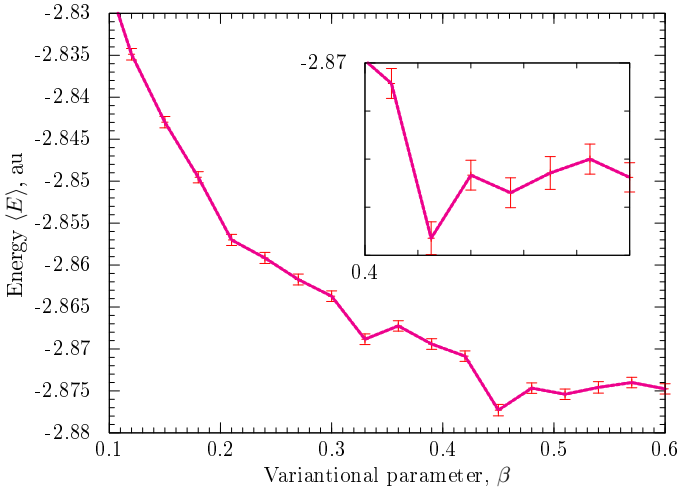

Figure 7.2: Energies for the He atom as a function of the variational parameter β executing a plain Python simulator. The experiment was carried out with 1×10^6 cycles, $dt = 0.1$ and $\alpha = 1.6785$.

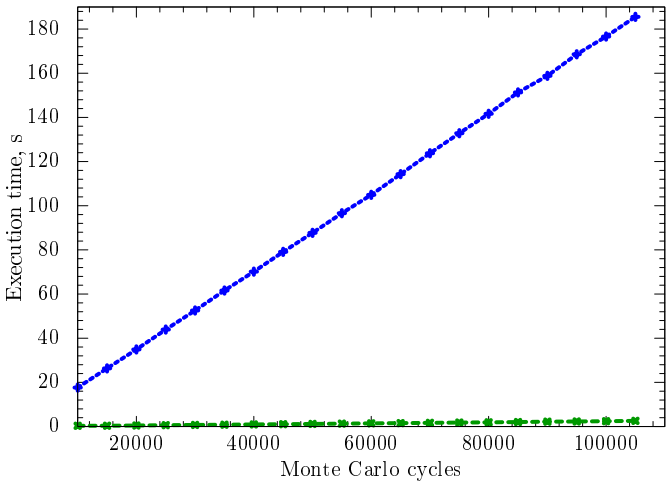

Figure 7.3: Execution time as a function of the number of Monte Carlo cycles for a Python (blue) and C++ (green) simulators implementing the QVMC method with importance sampling for the He atom.

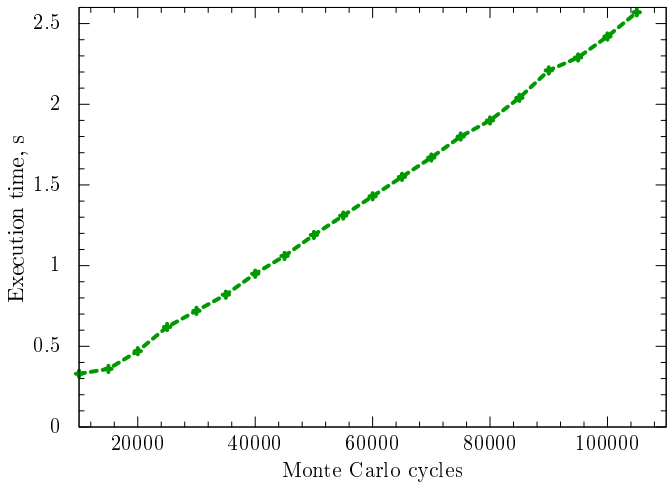

Figure 7.4: Details of the execution time for QVMC C++ simulator at the bottom of figure (7.3.

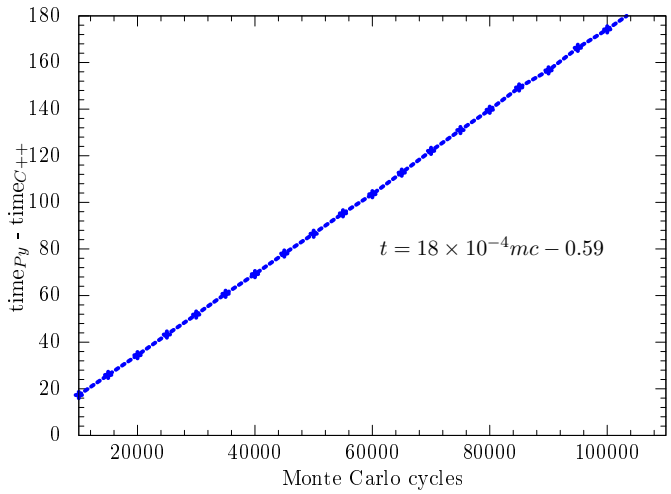

Figure 7.5: Evolution of the time delay of the Python implementation with respect to the C++ code as a function of the number of Monte Carlo cycles.

# calls	Total time	Cum. time	Class:function
1	9.153	207.061	MonteCarlo.py:(doMCISampling)
1	0.000	207.061	VMC.py:(doVariationalLoop)
1910014	23.794	159.910	Psi.py:(getPsiTrial)
100001	12.473	117.223	Psi.py:(getQuantumForce)
1910014	58.956	71.704	Psi.py:(getModelWaveFunctionHe)
50000	0.864	66.208	Energy.py:(getLocalEnergy)
1910014	57.476	64.412	Psi.py:(getCorrelationFactor)
50000	8.153	62.548	Energy.py:(getKineticEnergy)
6180049	21.489	21.489	:0(sqrt)
900002	4.968	4.968	:0(copy)
300010	2.072	2.072	:0(zeros)
50000	2.272	2.796	Energy.py:(getPotentialEnergy)

Table 7.2: Profile of a QVMC simulator with importance sampling for the He atom implemented in Python. The run was done with 50000 Monte Carlo cycles.

for He and Be atoms, a profile using 50000 Monte Carlo cycles was carried out.

The profiles[3] for He and Be implementations in tables 7.2 and 7.3 show that the [4] consuming part of a QVMC simulator are related to the evaluation of the trial wave function. For a single He atom, it takes 58.596 and 57.476 seconds for the determinant (getModelWaveFunction) and correlation part (getCorrelationFactor), respectively.

A function that frequently has to be computed in the QVMC algorithm is the square root, because of the dependence of the Slater determinant, Jastrow correlation function, kinetic and potential energies on the inter-electronic and electron-nucleus distances. In general, the evaluation of this quantity in Python is expensive, especially when it is not used correctly. Reference [?] recommends using math.sqrt() and numpy.sqrt() for scalar and vectorial arguments, respectively. Improvements of the actual algorithm in this sense are limited.

The fact that only one particle is moved at a time leads inevitably to an evaluation of the square root each time such a move happens, making the vectorization of the for-loops difficult. An alternative is to let Python run only the administrative part and move the rest of the simulator to C++. An advantage with this approach is that we do not need many functions to convert data transfered between Python and C++. Moreover, we can use the administrator class in Python to create objects in C++, introducing automatic garbage collection at a high level.

7.1.3 Comparing execution time of pure C++ vs mixed languages implementation

In this section we examine the perfomance of the pure C++ implementation with respect to the mixed Python/C++ code. We are interested in finding a possible delay in the execution

[3] The CPU time of a Python script increases with some factor (typically five [?]) when run under the administration of the profile module. It is assumed that the relative CPU time among functions is not affected by this overhead.

[4] Total time shows the total time spent in a specific function, while cum. time refers to the total time spent in the function and all the functions it calls.

# calls	Total time	Cum. time	Class:function
1	17.985	2417.743	MonteCarlo.py:(doMCISampling)
1	0.000	2417.743	VMC.py:(doVariationalLoop)
6220026	81.841	2305.124	Psi.py:(getPsiTrial)
200001	41.787	1828.758	Psi.py:(getQuantumForce)
6220026	532.861	1171.609	Psi.py:(getModelWaveFunctionBe)
6220026	921.182	1051.674	Psi.py:(getCorrelationFactor)
50000	0.912	477.214	Energy.py:(getLocalEnergy)
50000	15.313	467.341	Energy.py:(getKineticEnergy)
24880104	295.931	295.931	Psi.py:(phi2s)
63300273	220.166	220.166	:0(sqrt)
24880104	215.998	215.998	Psi.py:(phi1s)
6820036	45.579	45.579	:0(zeros)
1700002	9.369	9.369	:0(copy)
50000	7.108	8.961	Energy.py:(getPotentialEnergy)

Table 7.3: Profile of a QVMC simulator with importance sampling for the Be atom implemented in Python. The run was done with 50000 Monte Carlo cycles.

time as a function of the number of Monte Carlo cycles, using codes compiled with different optimization options. For the experiments we fix the variational parameters and set the time step $dt = 0.1$. During the timing of the mixed Python/C++ code, we only consider the time taken to make a variational loop, not the initialization of objects.

Figures 7.6 and 7.7 show that the delay between C++ and mixed code is not significant neither when increasing the number of Monte Carlo cycles nor when changing the optimization option used during compilation. The time used by Python to call compiled C++ code during one variational loop in the mixed code is negligible for this particular problem. In other applications, however, the effects of the optimization during compilation can be observable, and the advantage of mixed Python/C++ with respect to plain Python could be evident.

Because SWIG misses some functionalities for doing Python-C++ 100 % compatible, extra work was required to do the Python-C++ interface to work between the two languages. It can be tricky when combining Python with big libraries written in an ancient fashion or using constructs like operator overloading [?]. Moreover, the use of local variables inside functions called by for-loops leads to segmentation faults which were not observed with the original C++ code. This means that a developer of mixed code should be prepared for this kind of challenges.

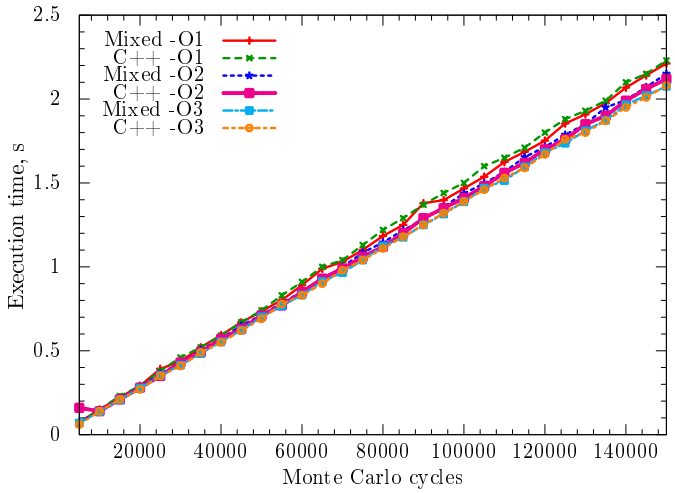

Figure 7.6: Execution time as a function of the number of Monte Carlo cycles for mixed Python/C++ and pure C++ simulators implementing the QVMC method with importance sampling for He atom.

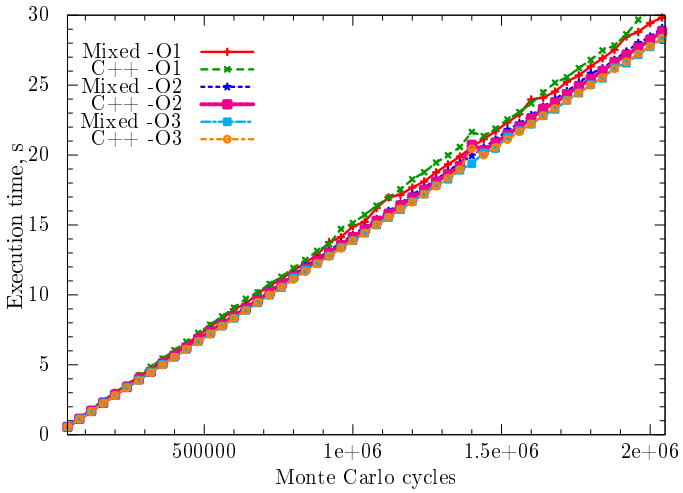

Figure 7.7: Execution time as a function of the number of Monte Carlo cycles for mixed Python/C++ and pure C++ simulators implementing the QVMC method with importance sampling for He atom.

System	dt	Monte Carlo cycles	Equilibration cycles
He	0.01	1×10^7	10 %
Be	0.01	1×10^7	10 %
2DHO2e ($\omega = 1.0$)	0.01	1×10^7	10 %
2DHO6e ($\omega = 1.0$)	0.01	1×10^7	10 %

Table 7.4: Experimental setup used to compute the dependence of the energy on the variational parameter α when no correlation and Coulomb interation are included. All runs were carried out on one computer, that is only the serial version of the code was used.

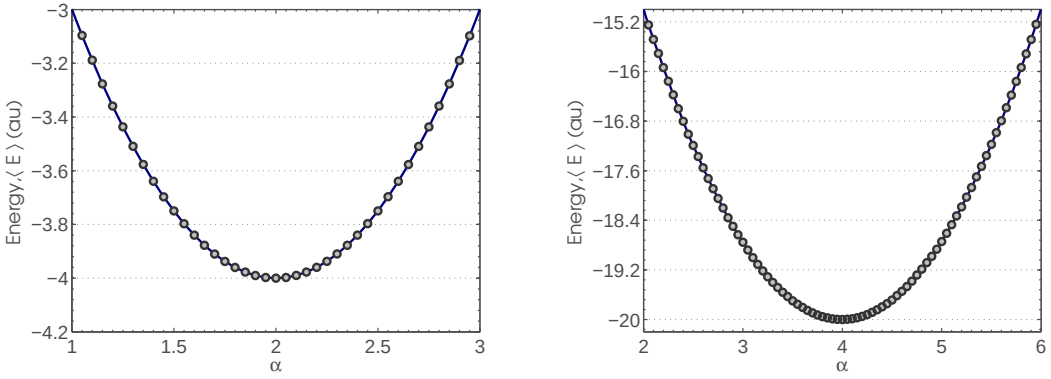

Figure 7.8: Dependence of the energy on the parameter α for He(left) and Be(right) atoms.

7.2 Validation of the optimized QVMC simulator implemented in pure C++

7.2.1 Dependence of the uncorrelated energy on the variational parameter α

We test the capability of the simulator to follow the variational principle when no Coulomb interaction nor correlation functions (Jastrow factor) are included in the models of He, Be and quantum dots (two dimensional harmonic oscillator) with two and six electrons (2DQDot2e and 2DQDot6e, in the following). For the experiment we use the setup of table 7.4, and compute the variational energy as a function of the parameter α. The step length dt was set to give an acceptance around 98 %.

The ground state energies computed in figures 7.8 to 7.9 and table 7.5 are in perfect agreement with the analytical values obtained in chapter 5 section 5.4 and fullfil the zero variance principle of section 5.5. It is an indication that the Slater determinant part of the simulator works well.

7.2.2 Graphical estimation of the ground state energies

To examinate the dependence of energy on the variational parameters α and β by graphical methods we set the number of Monte Carlo cycles to 1×10^7 and used 10 % of equilibration steps with the same dt given in table 7.4.

The estimated values of energy and variational parameters are summarized in table 7.6.

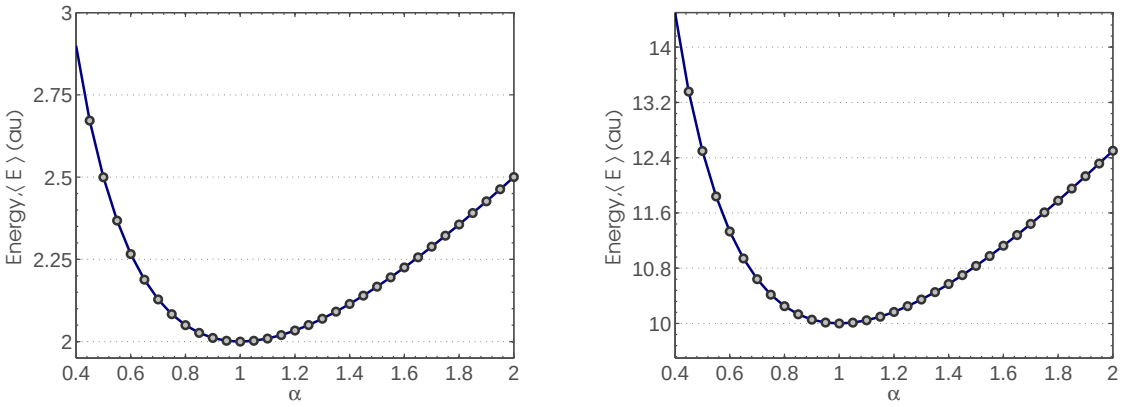

Figure 7.9: Dependence of the energy on the parameter α for a two dimensional harmonic oscillator with two and six electrons electrons, respectively.

System	Ground state energy, E (au)	$\alpha_{optimal}$
He	4.0	2.0
Be	20.0	4.0
2DHO2e ($\omega = 1.0$)	1.0	1.0
2DHO6e ($\omega = 1.0$)	10.0	1.0

Table 7.5: Analytical values of the ground state energy and corresponding variational parameter α computed for the Slater determinant. 2DHO2e stands for two-dimensional harmonic oscillator with two electrons.

The relatively big differences observed in energies for atoms and quantum dots are due to the scale length of each of these systems and to the fact that quantum dots do not have a nucleus which the electrons feel. Although these resulst are not exact, they are a good starting point as input parameters for the quasi-Newton method to be used later, and give an insight in which values to expect for energies and optimized variational parameters.

System	$\alpha_{optimal}$	$\beta_{optimal}$	Energy, $\langle E \rangle$ (au)
He	1.85	0.35	$-2.8901 \pm 1.0 \times 10{-}4$
Be	3.96	0.09	$-14.5043 \pm 4.0 \times 10^{-4}$
2DQDot2e ($\omega = 1.0$)	0.98	0.41	$3.0003 \pm 1.2 \times 10^{-5}$
2DQDot6e ($\omega = 1.0$)	0.9	0.6	$20.19 \pm 1.2 \times 10^{-4}$

Table 7.6: Values of the ground state (correlated) energy and corresponding variational parameters α and β estimated graphically. The shorthand 2DQDot2e stands for two-dimensional quantum dot with two electrons.

Attempting to optimize in this way is practical just for small systems, where the time of execution for the number of Monte Carlo iterations needed is not critical. Getting an estimate of the variational parameter α in this way is straightforward. For the parameter β, however, some extra work had to be done. Adjustments of the grids for both α and β were necessary to get a good estimate of the last. For systems with more than four electrons, it is not practical because of the computational cost involved.

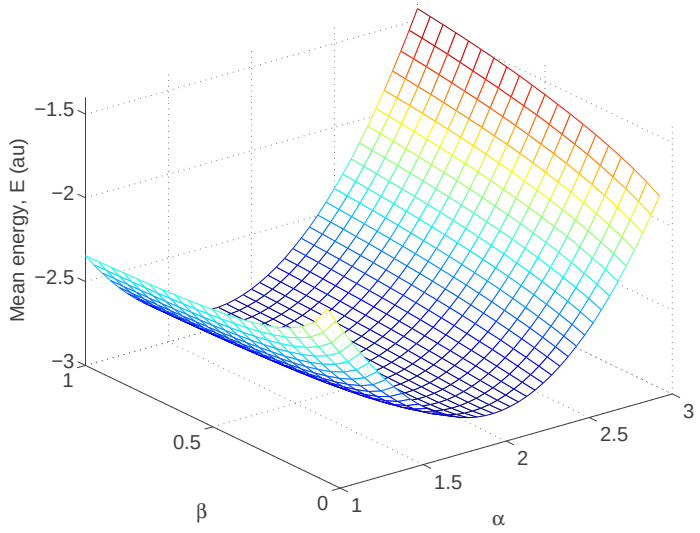

Figure 7.10: Dependence of energy on the parameters α and β for a He atom. The experimental setup is shown in table 7.4.

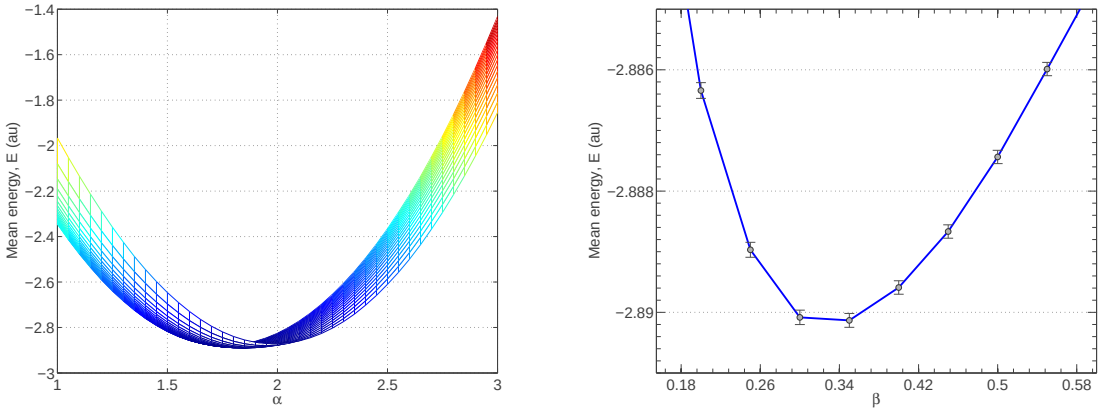

Figure 7.11: xz-view of α (left) in figure 7.10. To the right we show the dependence of the energy on β along the value of α that gives the minimum variational energy for a He atom. The parameters for these results are shown in table 7.4.

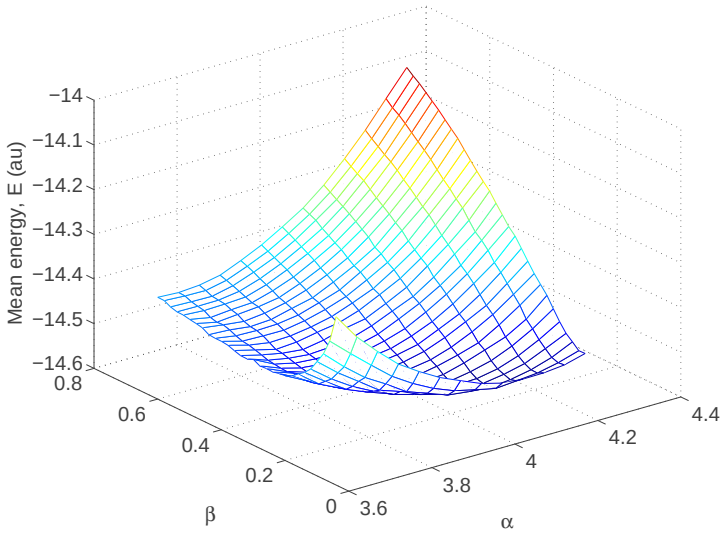

Figure 7.12: Dependence of energy on the parameters α and β for a Be atom. The parameters for these results are shown in table 7.4.

7.3 Optimization of the variational parameters by the quasi-Newton method.

This section deals with the validation of the quasi-Newton method as an optimization technique for the QVMC algorithm. All the experiments were performed with 1×10^7 Monte Carlo cycles, 10 % of equilibration steps in order to reach the most likely state and a time step $dt = 0.01$ for atoms and $dt = 0.06$ for quantum dots. The time step is used in the importance sampling part.

The evolution of the variational parameters α, β and energy during optimization of the

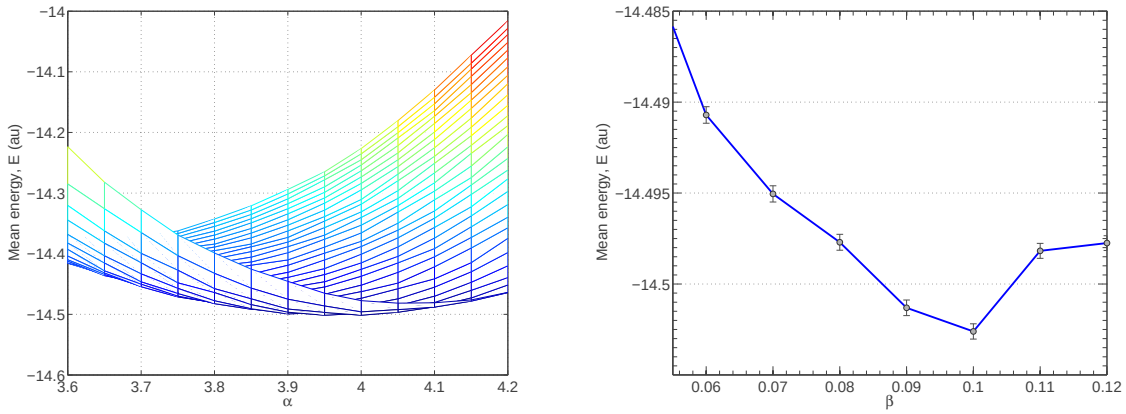

Figure 7.13: An xz-view of α (left) in figure 7.12. To the right we show the dependence of the energy on β along the value of α which gives the minumum variational energy for a Be atom. The parameters for these results are shown in table 7.4.

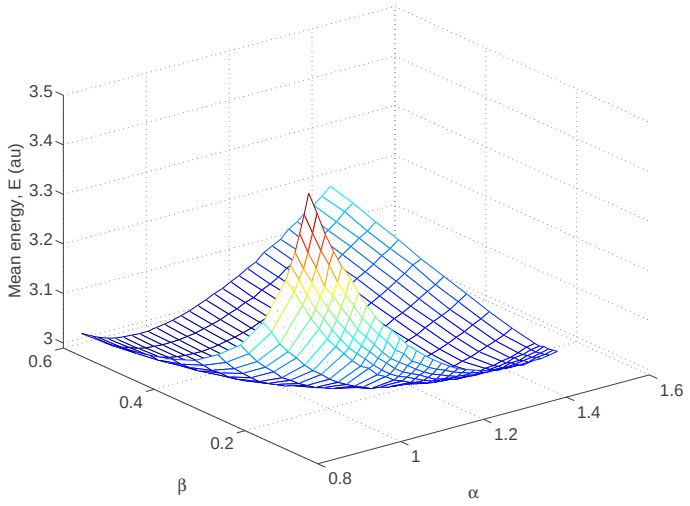

Figure 7.14: Dependence of the energy on the parameters α and β for a two dimensional quantum dot with two electrons. The experiment was carried out with 10^7 Monte Carlo cycles, 10% equilibration steps and $dt = 0.01$.

trial wave function of He and Be atoms with the quasi-Newton method is shown in figures 7.18 and 7.19. In both cases the optimizer makes a big jump during the three first iterations and then stabilizes around the optimal parameters and minimal energy. Table 7.7 summarizes the results. In general, these energies are in agreement with the previous computations from section 7.2.2, confirming that the algorithm suggested in section 4.10 for evaluating the numerical derivative of the energy with respect to the variational parameters is correct.

For quantum dots, the optimization of the trial wave function failed, becacuse it is almost constant with respect to the variational parameter β in the region where the minimum

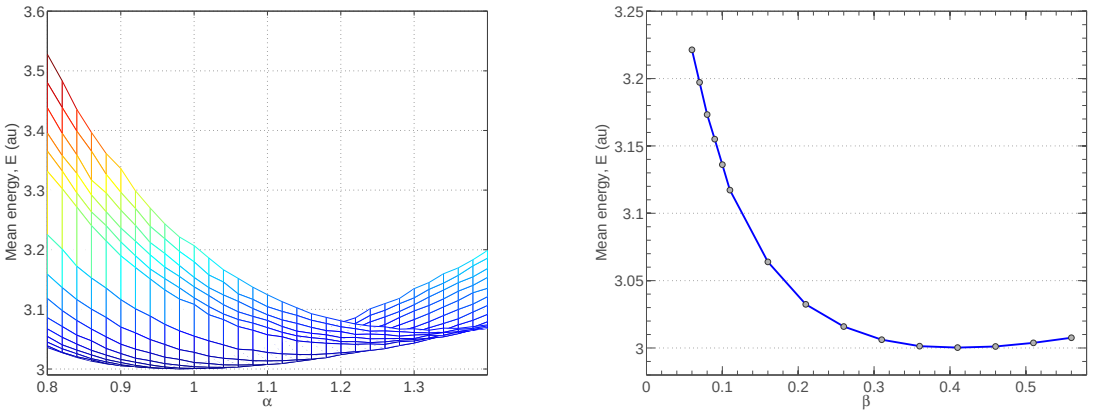

Figure 7.15: An xz-view of α (left) in figure 7.14. To the right we show the dependence of the energy on β along the value of α that gives the minumum variational energy for a two-dimensional quantum dot with two electrons. The parameters for these results are shown in table 7.4.

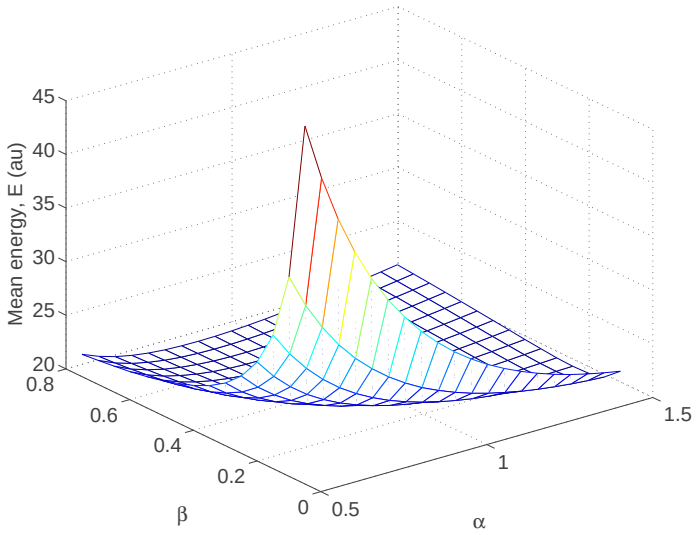

Figure 7.16: Dependence of energy on the parameters α and β for a two dimensional quantum dot with six electrons. The experiment was carried out with 10^7 Monte Carlo cycles, 10% equilibration steps and $dt = 0.01$.

is supposed to be located. Therefore, the approximated gradient computed in the quasi-Newton algorithm becomes zero or is very small for these points. This optimization method uses the gradient to build up the curvature information at each iteration to formulate a quadratic model problem, which explains the success when optimizing wave functions of atoms. In the following we use the variational parameters α and β reported by Albrigtsen [?].

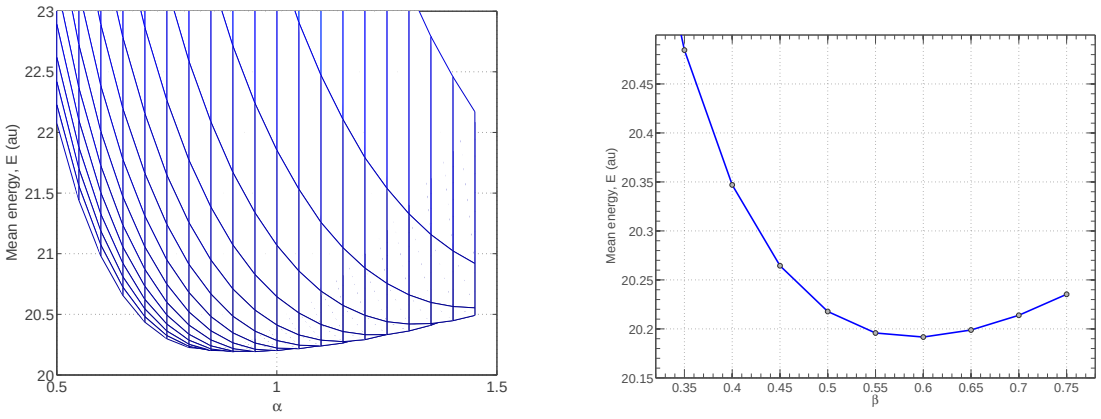

Figure 7.17: An xz-view of α (left) in figure 7.14. To the right we show the dependence of the energy on β along the value of α that gives the minumum variational energy for a two-dimensional quantum dot with six electrons. The parameters for these results are shown in table 7.4.

Deterministic methods are, in general, not suitable for problems affected by noise. They are either unable to reach an optimum or reach a false one [?]. A better option is the so called Stochastic Gradient Approximation (SGA). It is a probabilistic iterative method with variable step size, which uses the stochastic noise to make small jumps in the regions where the wave function is plane [?, ?]. Moreover, the number of configurations needed to reach an optimum is small [?].

System	α_0	β_0	α_{opt}	β_{opt}	Energy, (au)
He	1.564	0.134	1.838	0.370	-2.891
Be	3.85	0.08	3.983	0.104	-14.503

Table 7.7: Optimized variational parameters and corresponding energy minimization using a quasi-Newton method and the algorithm suggested in section 4.10. The rest of the parameters were: 10^7 Monte Carlo cycles with 10 % equilibration steps, $dt = 0.01$ for atoms.

7.4 Evaluating the ground state energy

In this section we evaluate the ground state energy of the quantum mechanical systems studied in this thesis. The Quantum Variational Monte Carlo method with importance sampling based on the Fokker-Planck algorithm is highly sensitive to the time step used for approximating the Langevin equation. Besides introducing numerical oscillations, a big dt will cause the particles to move frequently to the regions in configuration space with marginal probability. This means in turn that the contribution to the expectation value of say the mean energy are negligible. On the other hand, a short dt implies that the particles get stuck in the same place, and other possible configurations will not be sampled resulting in poorer statistics.

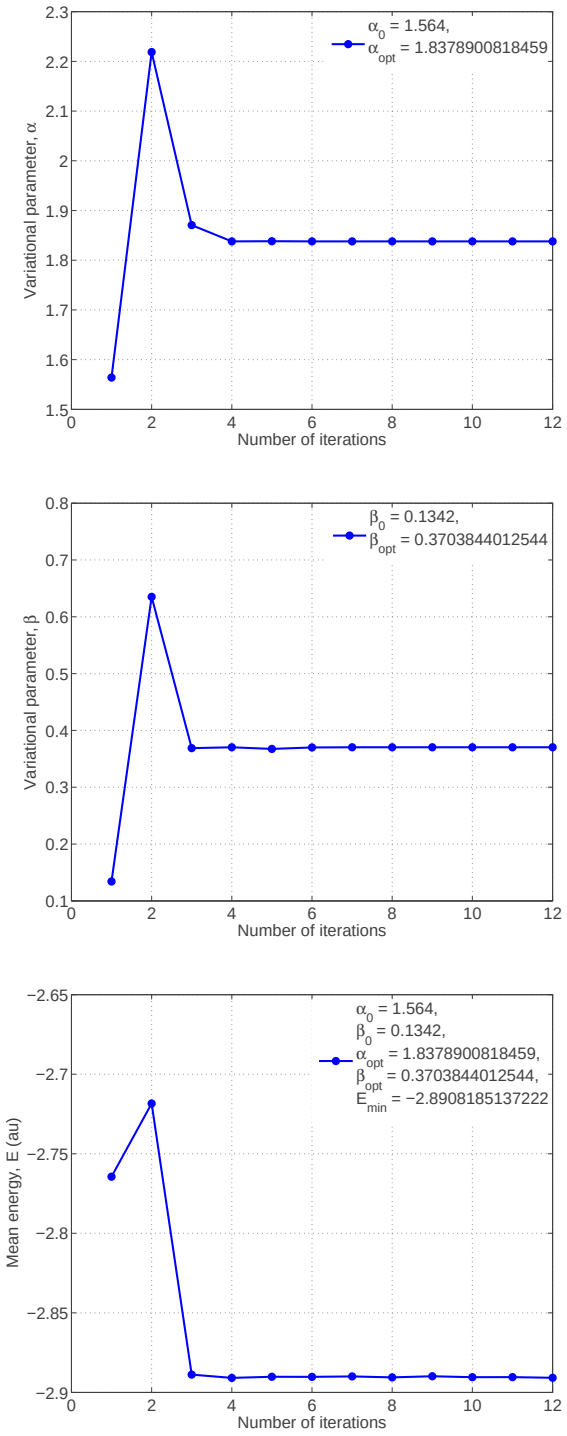

Figure 7.18: Evolution of the variational parameters α and β (top), energy (bottom) as a function of the number of iterations during the optimization phase with the quasi-Newton method of the trial wave function of He atom. The experiment was carried out with 10^7 Monte Carlo cycles and 10% equilibration steps in four nodes.

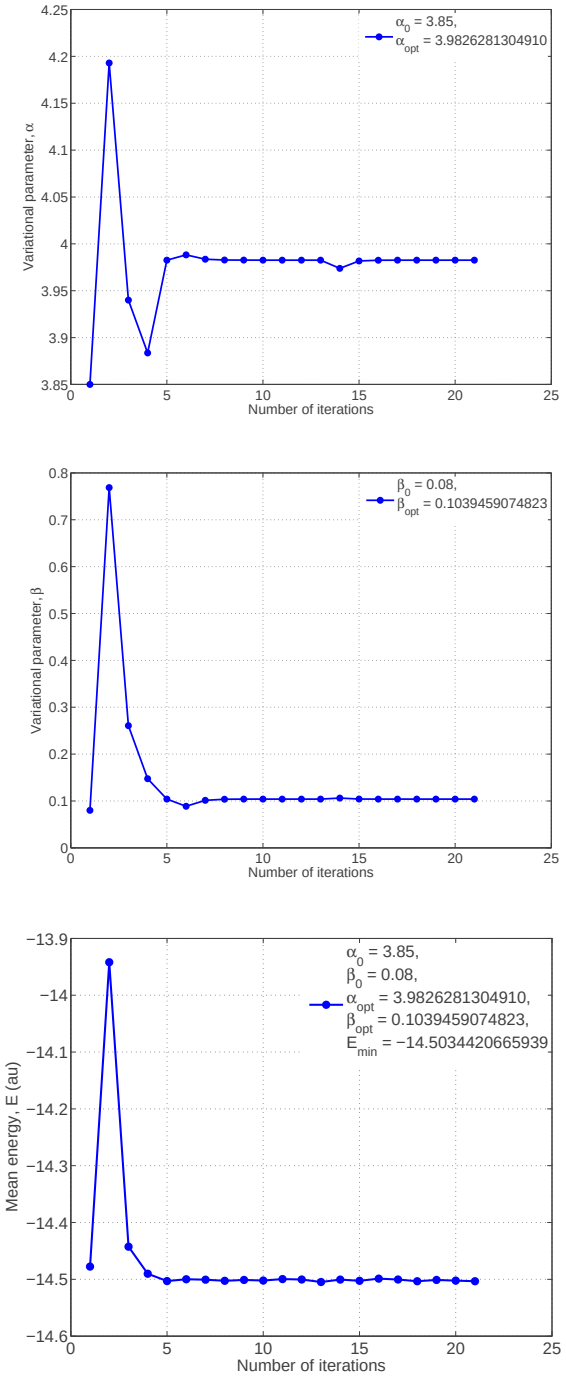

Figure 7.19: Evolution of the variational parameters α and β (top), energy (bottom) as a function of the number of iterations during the optimization stage with the quasi-Newton method of the trial wave function of Be atom. The experiment was carried out with 10^7 Monte Carlo cycles and 10% equilibration steps in four nodes.

Here we locate a $dt-$range in the $energy - dt$ plane where the energy is changing quasi linearly with the time step, as shown in figures 7.20 to 7.23 (left). This linearity reflects some kind of stability in the simulation, and allows us to compute a better estimate for

the ground state energy by extrapolating it to zero, that we extrapolate to $dt = 0$, as this parameter introduces a bias [?,?]. At that time we overcome the problem of choosing a very small time step dt.

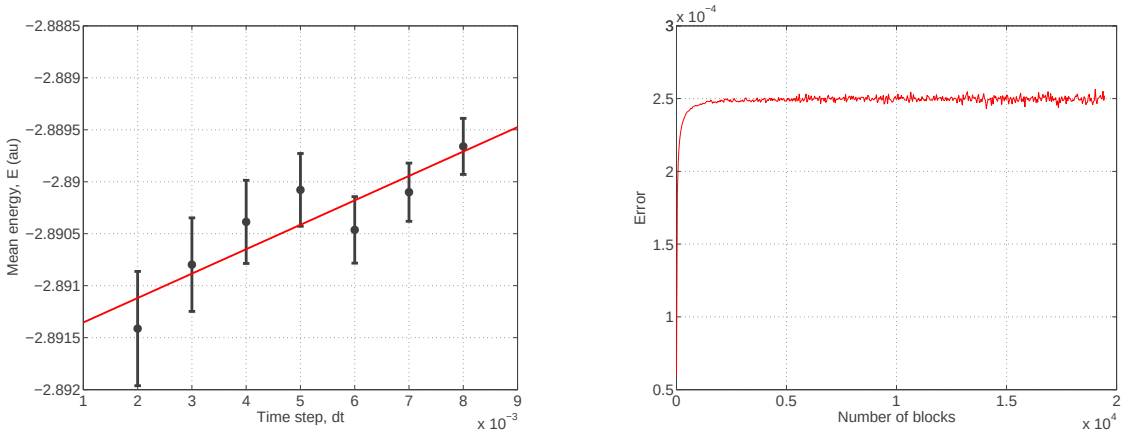

Figure 7.20: To the left we show the extrapolation to $dt-$zero of the energy for a He atom. To the right we display the details of the blocking analysis at $dt = 0.01$ where the energy $-2.89039 \pm 2.5 \times 10^{-4}\, au$. Experimental setup: 10^7 Monte Carlo cycles, 10% equilibration steps with four nodes, with $\alpha = 1.8379$, $\beta = 0.3704$.

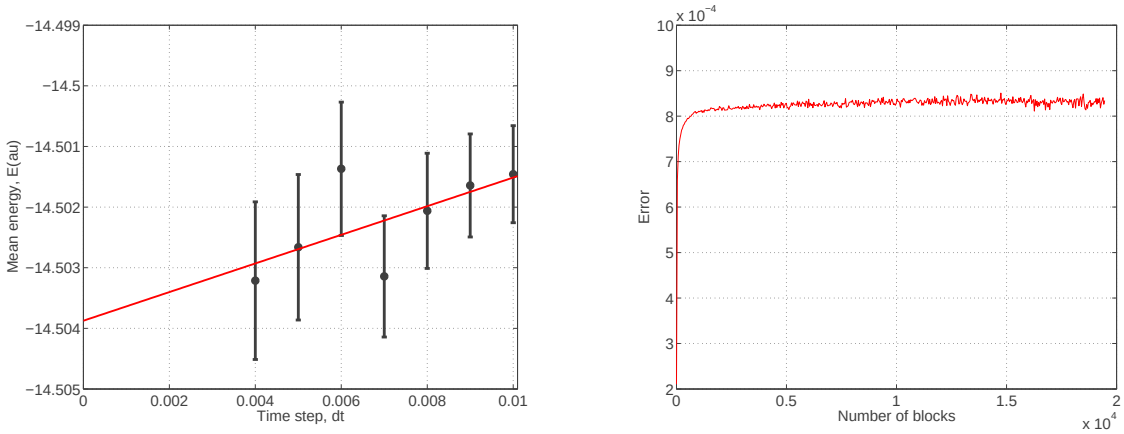

Figure 7.21: Extrapolation to $dt-$zero of the energy (left) and blocking analysis at $dt = 0.01$ for the Be atom where the energy $E = -14.50146 \pm 8.5 \times 10^{-4}\, au$. Experimental setup: 10^7 Monte Carlo cycles, 10% equilibration steps for four nodes, with $\alpha = 3.983$ and $\beta = 0.103$.

Allocating the region with quasi-linear $energy - dt$ dependence is computationally costly because of the number of evaluations (in parallel) required to explore the domain of dt. Fixing the seed in the random generator and reducing the number of Monte Carlo cycles is a straigforward way of allocating such a domain. This exploration could be done as a serial computation as well. It could be used later to produce results with a random seed and in parallel.

Tables 7.8 to 7.11 present values of energies as a function of dt as well as the percentage of accepted moves for the systems studied in this thesis. This percentage is inversly proportional with dt in the region with a quasi-linear $energy-dt$ dependence. Moreover, it remains above 99 %. In the same domain, the step sizes observed for quantum dots are longer than for atoms, indicating that systems with bigger length scales require longer jumps in dt to cover a bigger configuration space during the sampling process.

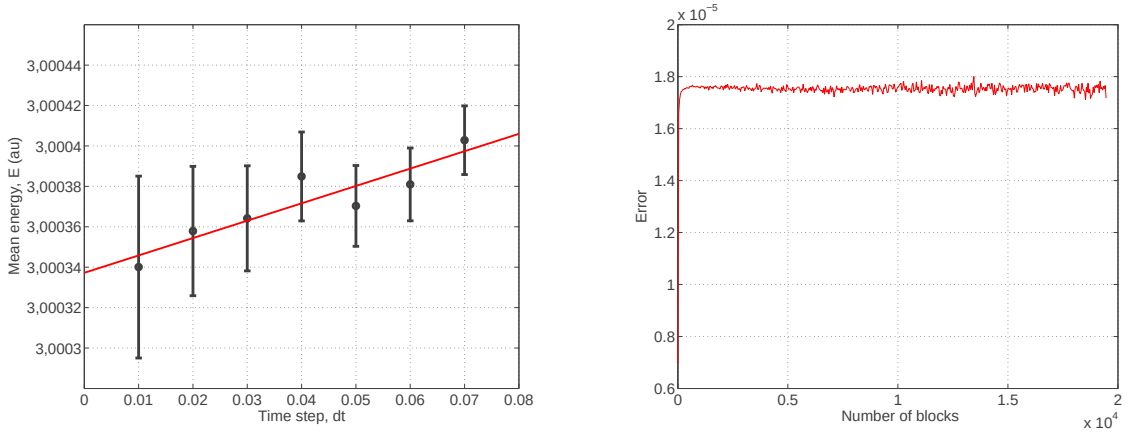

Figure 7.22: Extrapolation to $dt-$zero of the energy for a two-dimensional quantum dot with two electrons (left) and mean energy error ($E_{min} = 3.000381 \pm 1.8 \times 10^{-5}\, au$) at $dt = 0.06$ as a function of the number of blocks for a 2DQDot2e (right). Parameters of simulation: 1×10^7 Monte Carlo steps with 10 % equilibration steps, $\alpha = 0.99044$ and $\beta = 0.39994$. The error varies from 1.7×10^{-5} at $dt = 0.07$ to 4.5×10^{-5} at $dt = 0.01$.

Time step	Energy, (au)	Error	Accepted moves, (%)
0.002	-2.891412637854	5.5e-4	99.97
0.003	-2.890797649433	4.5e-4	99.94
0.004	-2.890386198895	4.0e-4	99.91
0.005	-2.890078440930	3.5e-4	99.88
0.006	-2.890463490951	3.2e-4	99.84
0.007	-2.890100432462	2.8e-4	99.81
0.008	-2.889659923905	2.7e-4	99.77

Table 7.8: Energy computed for the He atom and the error associated as a function of the time step. Parameters: 10^7 Monte Carlo cycles with 10 % equilibration steps, $\alpha = 1.8379$ and $\beta = 0.3704$.

The results for the energy of atoms and 2DQDot2e using extrapolation to zero dt shown in table 7.12 are in correspondence with values listed in the scientific literature, see for example Refs. [?,?]. In general, the difference between the lowest and highest energies were small. Doing an extrapolation to zero can be expensive in terms of the computational time involved in doing blocking for each dt, and is practical for big systems. One should evaluate

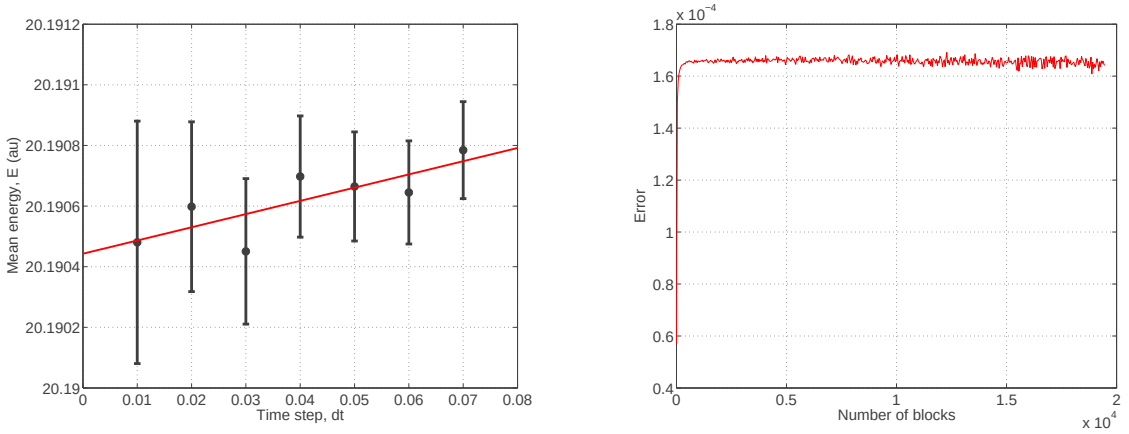

Figure 7.23: Extrapolation to $dt-$zero of the energy for a two-dimensional quantum dot with six electrons (left) and mean energy error ($E_{min} = 20.190645 \pm 1.7 \times 10^{-4}\, au$) at $dt = 0.06$ as a function of the number of blocks for a 2DQDot6e (right). Parameters of simulation: 10^7 Monte Carlo steps with 10 % equilibration steps, $\alpha = 0.926273$ and $\beta = 0.561221$ running with four nodes.

Time step	Energy, (au)	Error	Accepted moves, (%)
0.004	-14.50321303316	1.3e-3	99.61
0.005	-14.50266236227	1.2e-3	99.47
0.006	-14.50136820967	1.1e-3	99.32
0.007	-14.50314292468	1.0e-3	99.17
0.008	-14.50206184582	9.5e-4	99.01
0.009	-14.50164368104	8.5e-4	98.85
0.01	-14.50145748870	8.0e-4	98.68

Table 7.9: Energy computed for the Be atom and the error associated as a function of the time step. Parameters: 10^7 Monte Carlo cycles with 10 % equilibration steps, $\alpha = 3.983$ and $\beta = 0.103$.

the gain of getting some decimals digits in energy against the computational cost required by the computation.

In relation with the figures 7.20 to 7.23(right), in all the cases the amplitude of the oscillations in the error increases with the number of blocks, reflecting the fact that the greater the number of blocks is, the greater the sequential correlation in the set of correlated energy data becomes, because of the reduction of the distance between the $i^{th}-$entry in each block.

Later, the blocking analysis shows also that the greater the system, the greater the error in energy is for equal number of Monte Carlo cycles and spatial dimensions, simply due to the fact that the number of degrees of freedom and the role of correlations increase with the size of the system.

The effect of the interelectronic correlaton will, however, have less impact in higher di-

Time step	Energy, (au)	Error	Accepted moves, (%)
0.01	3.000340072477	4.5e-5	99.95
0.02	3.000357900850	3.2e-5	99.87
0.03	3.000364180564	2.6e-5	99.77
0.04	3.000384908560	2.2e-5	99.65
0.05	3.000370330692	2.0e-5	99.52
0.06	3.000380980039	1.8e-5	99.37
0.07	3.000402836533	1.7e-5	99.21

Table 7.10: Results of a blocking analysis for several time steps. The system was a two-dimensional quantum dot with two electrons and $\omega = 1.0$. The rest of the parameters were: 10^7 Monte Carlo cycles with 10 % equilibration steps, and $\alpha = 0.99044$ and $\beta = 0.39994$ taken from reference [?].

Time step	Energy, (au)	Error	Accepted moves, (%)
0.01	20.19048030567	4.0e-4	99.90
0.02	20.19059799459	2.8e-4	99.75
0.03	20.19045049792	2.4e-4	99.55
0.04	20.19069748408	2.0e-4	99.34
0.05	20.19066469178	1.8e-4	99.10
0.06	20.19064491561	1.7e-4	98.85
0.07	20.19078449010	1.6e-4	98.58

Table 7.11: Results of a blocking analysis for several time steps. The system was a two-dimensional quantum dot with six electrons and $\omega = 1.0$. The rest of the parameters were: 10^7 Monte Carlo cycles with 10 % equilibration steps, and $\alpha = 0.926273$ and $\beta = 0.561221$ taken from reference [?].

System	Energy, (au)
He	-2.8913
Be	-14.5039
2DQDot2e	3.0003
2DQDot6e	20.1904

Table 7.12: Energies estimated using zero-dt extrapolation in figures 7.8 to 7.11. $\omega = 1.0$ for quantum dots.

mensions. At lower dimensions the particles have less degrees of freedom to avoid each other, but make it more difficult for more than two particles to get close to each other.

7.5 Error as a function of the number of Monte Carlo cycles in Be atom

In this experiment we were interested in finding the behaviour of the error as a function of the number of Monte Carlo cycles. The runs were executed with a Be atom with fixed seed (in the random number generator). Figures 7.24 (right and left) show the error decaying exponentially after which it stabilizes, indicating that no much gain in the reduction of the error in reached by increasing the number of Monte Carlo cycles (and computational cost).

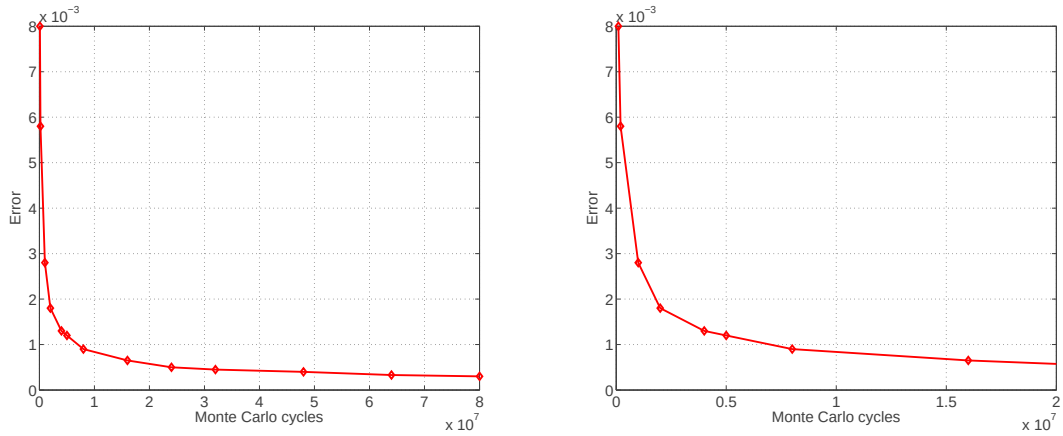

Figure 7.24: Error in the energy (left) and zoom(right) obtained by blocking for a Be atom as a function of the number of Monte Carlo cycles. The set up for the experiment was: $dt = 0.01$, $\alpha = 3.981$, $\beta = 0.09271$ and 10 % equilibration steps by run.

Chapter 8

Conclusions and perspectives

In this thesis we have shown how to use low level programming languages like Python to prototype big codes in scientific computing. The extension of C++ with scripting capabilities via an interface with Python has also been discussed. Implementing the prototype in C++ requires twice as many lines than the version in Python and is, in general, a bit more complicated.

Later we have implemented a software in plain C++ (about 8000 lines of code) capable of computing the ground state energies of several closed shell quantum mechanical systems by using the variational Monte Carlo method. Applications include the helium and beryllium atoms and quantum dots with two and six electrons. Analytical expressions for code validation have been derived.

Moreover, a new method to efficiently compute the analytical derivative of the energy with respect to the variational parameters has been suggested.

8.1 Comparing Python with C++: a first impression

Python is a good programming language for prototyping and to structure big codes. In some applications it is flexible enough to be combined with other languages, but it needs some improvements. Furthermore, Python has an automatic garbage collection, which is a big productivity gain. Therefore, letting Python create the objects of C++ is convenient, because it will take care of the pointers being created and destroyed. In this way, most of the errors associated with memory management can be avoided. Moreover, it has a cleaner syntax than C++ and the number of lines neccesary to solve the same problem is significantly lower.

C++ is harder to learn than Python. It can take about one year to get to an acceptable level in C++ and only some weeks to start writing productive codes in Python. Altough it is markedly slower than C++, its interpreted execution and dynamic typing are productive gains, especially in the design phase for making prototypes of big libraries, but also during the test stage. It is also relatively easy and convenient to generate scripts for running big simulations almost automatically.

Calling C++ from Python is relatively easy when it can be done automatically by tools like SWIG. However, it will involve some manual job if the data to be transferred are arrays or if it is C++ that calls Python. Because Python is a "typeless" language, i.e., it does not declare explicitely variables as integers, floating number, etc, it should be indicated what datatype is transferred. Accomplishing it with the C-API is far from a straighforward job for an unexperienced programmers, and good tutorials are missing. Frequently it is an error prone work. In general, one should encapsulate this information in classes like convert.cpp from reference [?] used in this thesis.

As rule of thumb before attempting to combine programming techniques is that one should always profile and then identify the parts to be moved to other languages. In reference to the QVMC algorithm used in this thesis, it is convenient to let the evaluation of the wave function and the Hamiltonian be done in C++. Calling Python functions from C++ is known to be very slow and should be avoided. In fact, the gain obtained in combining Python and C++ in this particular problem does not seem to give a big gain, but we should remark that it was very useful in the prototyping stage. In any case one should find a balance between computational cost, flexibility of the implementation and facility to be used by the end user.

8.2 Parametric optimization of trial wave functions

Graphical methods for the estimation of variational parameters and energies are practical just for very small systems where the computational cost is not high. Nevertheless, they are good starting points to validate a code and give some insight in the kind of data output expected. On the other hand, the success of the quasi-Newton method in finding a minimum depends greatly on the starting values given to the variational parameters and on the kind of wave function being considered.

The Quasi-Newton method gives acceptable results for wave functions of atomic systems such as helium and berylium, but fails to find an optimum for quantum dot wave functions. The higher correlation in atoms is reflected in the pronounced curvature of the $energy - \beta$ graphs of these systems and improves the accuracy of the method in finding a minima. Good starting values for α are near the nuclear charge[1]. The tuning of β is a bit more difficult, and should be done with several runs.

Because the method is designed to handle quadratic functions, it converges poorly to a global minima in regions with poor curvature, as happens in the quantum dot case. Its big weakness is its high sensibility to stochastic noise, requiring a relatively high number of Monte Carlo cycles to get stable interactons.

[1] For quantum dots, the optimal value of α was always located near $\alpha = 1.0$ when $\omega = 1.0$.

8.3 Energy computations

For the systems studied in this thesis, the QVMC simulator developed gives good results for the expectation value of the energy when optimal parameters are provided. Doing extrapolation to dt zero is computationally expensive, especially for big systems. By observing the evolution of the percent of accepted moves with time step using just a few Monte Carlo cycles, one could efficiently locate the region with quasi linear $energy - dt$ behaviour, before starting the production phase, at a lower computational cost. For the algorithm described in this thesis, the percent of accepted steps should be, at least, 96 %.

8.4 Further work

In connection with mixing Python with C++, work is still needed in the Single Wrapper Interface Generator (SWIG) to improve the compatibility of the two languages. Efforts should also be done to make Python faster.

Alternative algorithms to reduce the evaluation of mathematical functions should be tried. Reference [?] suggests a convenient expression for the kinetic energy. Moreover, other versions of the quasi-Newton method adapted to handle stochastic noise should be examined. One possibility is the so-called Stochastic Gradient Approximation [?].

The way the current algorithm fixes the time step for atoms, introduces some bias. Core electrons should be sampled with shorter dt than the others lying in the valence shells because the region to sample is smaller. Giving the same dt to both gives more acceptance to the core electrons. On the other hand, increasing the step size to sample the valence region better, sacrifices an optimal sampling of the core electrons.

Implementing the single particle wave functions, the single-particle potential, the inter-particle potential and the Jastrow function as functors (classes behaving as functions) combined with templates would make a more robust code. Adding a load balance checker to the Parallelizer class would improve the use of computational resources when working in parallel.

For more clarity in the code, one should separate the statistical data analysis in an Observable with a member pointer to Energy class. It would help to automatize the blocking analysis, since it is a part taking long time when done by hand. Extensions to load wave functions from Hartree-Fock simulations could be convenient, as well as improvements to try open shell problems, and some extra functionality to deal with optimization of variance and variance/energy.

Appendix A

Derivation of the algorithm for updating the inverse Slater matrix

The following derivation follows the ideas given in Ref. [?]. As starting point we may consider that each time a new position is suggested in the Metropolis algorithm, a row of the current Slater matrix experiences some kind of perturbation. Hence, the Slater matrix with its orbitals evaluated at the new position equals the old Slater matrix plus a perturbation matrix,

$$D_{jk}(\boldsymbol{x}^{\boldsymbol{new}}) = D_{jk}(\boldsymbol{x}^{\boldsymbol{cur}}) + \Delta_{jk}. \tag{A-1}$$

where

$$\Delta_{jk} = \delta_{ik}[\phi_j(\boldsymbol{x}_{\boldsymbol{i}}^{\boldsymbol{new}}) - \phi_j(\boldsymbol{x}_{\boldsymbol{i}}^{\boldsymbol{old}})] = \delta_{ik}(\Delta\phi)_j \tag{A-2}$$

Computing $(B^T)^{-1}$ we arrive to

$$D_{kj}(\boldsymbol{x}^{\boldsymbol{new}})^{-1} = [D_{kj}(\boldsymbol{x}^{\boldsymbol{cur}}) + \Delta_{kj}]^{-1}. \tag{A-3}$$

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

$$\begin{aligned}
[\boldsymbol{D}^T(\boldsymbol{x}^{\boldsymbol{new}})]^{-1} &= [\boldsymbol{D}^T(\boldsymbol{x}^{\boldsymbol{cur}}) + \Delta^T]^{-1} \\
&= [\boldsymbol{D}^T(\boldsymbol{x}^{\boldsymbol{cur}})]^{-1} - [\boldsymbol{D}^T(\boldsymbol{x}^{\boldsymbol{cur}}) + \Delta^T]^{-1}\Delta^T[\boldsymbol{D}^T(\boldsymbol{x}^{\boldsymbol{cur}})]^{-1} \\
&= [\boldsymbol{D}^T(\boldsymbol{x}^{\boldsymbol{cur}})]^{-1} - \underbrace{[\boldsymbol{D}^T(\boldsymbol{x}^{\boldsymbol{new}})]^{-1}}_{\text{By Eq.}A-3}\Delta^T[\boldsymbol{D}^T(\boldsymbol{x}^{\boldsymbol{cur}})]^{-1}.
\end{aligned}$$

Using index notation, the last result may be expanded by

$$
\begin{aligned}
D^{-1}_{kj}(\boldsymbol{x}^{\boldsymbol{new}}) &= D^{-1}_{kj}(\boldsymbol{x}^{\boldsymbol{cur}}) - \sum_l \sum_m D^{-1}_{km}(\boldsymbol{x}^{\boldsymbol{new}}) \Delta^T_{ml} D^{-1}_{lj}(\boldsymbol{x}^{\boldsymbol{cur}}) \\
&= D^{-1}_{kj}(\boldsymbol{x}^{\boldsymbol{cur}}) - \sum_l \sum_m D^{-1}_{km}(\boldsymbol{x}^{\boldsymbol{new}}) \Delta_{lm} D^{-1}_{lj}(\boldsymbol{x}^{\boldsymbol{cur}}) \\
&= D^{-1}_{kj}(\boldsymbol{x}^{\boldsymbol{cur}}) - \sum_l \sum_m D^{-1}_{km}(\boldsymbol{x}^{\boldsymbol{new}}) \underbrace{\delta_{im}(\Delta\phi)_l}_{\text{By Eq. } A-2} D^{-1}_{lj}(\boldsymbol{x}^{\boldsymbol{cur}}) \\
&= D^{-1}_{kj}(\boldsymbol{x}^{\boldsymbol{cur}}) - D^{-1}_{ki}(\boldsymbol{x}^{\boldsymbol{new}}) \sum_{l=1}^{N} (\Delta\phi)_l D^{-1}_{lj}(\boldsymbol{x}^{\boldsymbol{cur}}) \\
&= D^{-1}_{kj}(\boldsymbol{x}^{\boldsymbol{cur}}) - D^{-1}_{ki}(\boldsymbol{x}^{\boldsymbol{new}}) \sum_{l=1}^{N} \underbrace{[\phi_l(\boldsymbol{r}_{\boldsymbol{i}}^{\boldsymbol{new}}) - \phi_l(\boldsymbol{r}_{\boldsymbol{i}}^{\boldsymbol{old}})]}_{\text{By Eq. } A-2} D^{-1}_{lj}(\boldsymbol{x}^{\boldsymbol{cur}}).
\end{aligned}
$$

From (4.5),

$$
\boldsymbol{D}^{-1}(\boldsymbol{x}^{\boldsymbol{cur}}) = \frac{adj\boldsymbol{D}}{|\boldsymbol{D}(\boldsymbol{x}^{\boldsymbol{cur}})|} \quad \text{and} \quad \boldsymbol{D}^{-1}(\boldsymbol{x}^{\boldsymbol{new}}) = \frac{adj\boldsymbol{D}}{|\boldsymbol{D}(\boldsymbol{x}^{\boldsymbol{new}})|}.
$$

Dividing these two equations we get

$$
\frac{\boldsymbol{D}^{-1}(\boldsymbol{x}^{\boldsymbol{cur}})}{\boldsymbol{D}^{-1}(\boldsymbol{x}^{\boldsymbol{new}})} = \frac{|\boldsymbol{D}(\boldsymbol{x}^{\boldsymbol{new}})|}{|\boldsymbol{D}(\boldsymbol{x}^{\boldsymbol{cur}})|} = R \Rightarrow D^{-1}_{ki}(\boldsymbol{x}^{\boldsymbol{new}}) = \frac{D^{-1}_{ki}(\boldsymbol{x}^{\boldsymbol{cur}})}{R}.
$$

Therefore,

$$
D^{-1}_{kj}(\boldsymbol{x}^{\boldsymbol{new}}) = D^{-1}_{kj}(\boldsymbol{x}^{\boldsymbol{cur}}) - \frac{D^{-1}_{ki}(\boldsymbol{x}^{\boldsymbol{cur}})}{R} \sum_{l=1}^{N} [\phi_l(\boldsymbol{r}_{\boldsymbol{i}}^{\boldsymbol{new}}) - \phi_l(\boldsymbol{r}_{\boldsymbol{i}}^{\boldsymbol{old}})] D^{-1}_{lj}(\boldsymbol{x}^{\boldsymbol{cur}}),
$$

or

$$
\begin{aligned}
D^{-1}_{kj}(\boldsymbol{x}^{\boldsymbol{new}}) = D^{-1}_{kj}(\boldsymbol{x}^{\boldsymbol{cur}}) \quad &- \quad \frac{D^{-1}_{ki}(\boldsymbol{x}^{\boldsymbol{cur}})}{R} \sum_{l=1}^{N} \phi_l(\boldsymbol{r}_{\boldsymbol{i}}^{\boldsymbol{new}}) D^{-1}_{lj}(\boldsymbol{x}^{\boldsymbol{cur}}) \\
&+ \quad \frac{D^{-1}_{ki}(\boldsymbol{x}^{\boldsymbol{cur}})}{R} \sum_{l=1}^{N} \phi_l(\boldsymbol{r}_{\boldsymbol{i}}^{\boldsymbol{old}}) D^{-1}_{lj}(\boldsymbol{x}^{\boldsymbol{cur}}) \\
= D^{-1}_{kj}(\boldsymbol{x}^{\boldsymbol{cur}}) \quad &- \quad \frac{D^{-1}_{ki}(\boldsymbol{x}^{\boldsymbol{cur}})}{R} \sum_{l=1}^{N} D_{il}(\boldsymbol{x}^{\boldsymbol{new}}) D^{-1}_{lj}(\boldsymbol{x}^{\boldsymbol{cur}}) \\
&+ \quad \frac{D^{-1}_{ki}(\boldsymbol{x}^{\boldsymbol{cur}})}{R} \sum_{l=1}^{N} D_{il}(\boldsymbol{x}^{\boldsymbol{cur}}) D^{-1}_{lj}(\boldsymbol{x}^{\boldsymbol{cur}}).
\end{aligned}
$$

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

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