Quantum Monte-Carlo Studies of Generalized Many-body Systems

by

Jørgen Høgberget

THESIS
for the degree of
MASTER OF SCIENCE

(Master in Computational Physics)

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

June 2013

Preface

During my stay at junior high school, I had no personal interest in mathematics. When the final year was finished, I sought an education within the first love of my life: Music. However, sadly, my application was turned down, forcing me to fall back on general topics. This is where I had my first encounter with physics, which should turn out not to be the last.

When high school ended, I applied for an education within structural engineering at the Norwegian University of Science and Technology, however, again I was turned down. Second on the list was the physics, meteorology and astronomy program at the University of Oslo. Never in my life have I been more grateful for being turned down, as starting at the University of Oslo introduced me to the second love of my life: Programming.

During the third year of my Bachelor I had a hard time figuring out which courses I should pick. Quite randomly, I attended a course on computational physics lectured by Morten Hjorth-Jensen. It immediately hit me that he had a genuine interest in the well-being of his students. It did not take long before I decided to apply for a Master's degree within the field of computational physics.

To summarize, me being here today writing this thesis is a result of extremely random events. However, I could not have landed in a better place, and I am forever grateful for my time here at the University of Oslo. So grateful in fact, that I am continuing my stay as a PhD student within the field of *multi-scale physics*.

I would like to thank my fellow Master students Sarah Reimann, Sigve Bøe Skattum, and Karl Leikanger for much appreciated help and good discussions. Sigve helped me with the minimization algorithm, that is, the darkest chapter of this thesis. A huge thanks to Karl for all those late nights where you gave me a ride home. Finally, thank you Sarah for withstanding my horrible German phrases all these years.

My supervisor, Morten Hjorth-Jensen, deserves a special thanks for believing in me, and filling me with confidence when it was much needed. This man is awesome.

Additionally I would like to thank the rest of the people at the computational physics research group, especially Mathilde N. Kamperud and Veronica K. B. Olsen whom with I shared my office, Svenn-Arne Dragly for helping me with numerous Linux-related issues, and "brochan" Milad H. Mobarhan for all the pleasant hours spent at Nydalen Kebab. You all contribute to making this place the best it can possibly be. Thank you!

Jørgen Høgberget

Oslo, June 2013

Contents

1

Introduction

Studies of general systems demand a general solver. The process of developing code aimed at a specific task is fundamentally different from the process of developing a general solver, simply due to the fact that the general equations need to be implemented *independent* of any specific properties a modelled system may contain. This is most commonly achieved through object oriented programming, which allows for the code to be structured into general implementations and specific implementations. The general - and specific implementations can then be interfaced through a functionality referred to as *polymorphism*. The aim of this thesis is to use object oriented C++ to build a general and efficient Quantum Monte-Carlo (QMC) solver, which can tackle several many-body systems, from confined electron systems, i.e. quantum dots, to bosons.

A constraint put on the QMC solver in this thesis is that the ansatz for the *trial wave function* consists of a single term, i.e. a single *Slater determinant*. This opens up the possibility to study systems consisting of a large number of particles, due to efficient optimizations in the single determinant. A simple trial wave function will also significantly ease the implementation of different systems, and thus make it easier to develop a general framework within the given time frame.

Given the simple ansatz for the wave function, the precision of Variational Monte-Carlo (VMC) is expected to be far from optimal, however, Diffusion Monte-Carlo (DMC) is supposed to withstand this problem, and thus yield a good final estimate nevertheless. To study this purposed power of DMC is another main focus of this thesis, in addition to pushing the limits regarding optimization of the code, and thus run ab-initio simulations of a large number of particles.

The two-dimensional quantum dot was chosen as the system of reference around which the code was planned. The reason for this is that all the current Master students are studying quantum dots at some level, which means that we can help each other reach a collective understanding of the system. Additionally, Sarah Reimann has studied two-dimensional quantum dots for up to 56 particles using a non-variational method called *Similarity Renormalization Group theory* [1]. Providing her with precise variational DMC benchmarks were considered to be of utmost importance. Coupled Cluster Singles and Doubles (CCSD) results are done up to 56 particles by Christoffer Hirth [2], however, for the lower frequencies, i.e. for higher correlations, CCSD struggles with convergence.

Depending on the success of the implementation, various additional systems could be implemented and studied in detail, such as atomic systems, three-dimensional - and double-well quantum dots.

Apart from benchmarking DMC ground state energies, the specific aim in the case of quantum dots is to study their behavior as the frequency is lowered. A lower frequency implies a higher correlation in the system. Understanding these correlated systems of electrons are of great importance to many-body theory in general. The effect of adding a third dimension is also of high interest. The advantage of DMC

compared to other methods is that the distribution is relatively easy to obtain.

Ground state energies for atomic systems can be benchmarked against experimental results [3–6], that is, precise calculations which are believed to be very close to the exact result for the given Hamiltonian, which yields an excellent opportunity to test the limits of DMC given a simple trial wave function. Going further to molecular systems, an additional aim is to explore the transition between QMC and molecular dynamics by parameterizing simple force field potentials [7].

Several former Master students, such as Christoffer Hirth [2] and Veronica K.B. Olsen [8], have studied two-dimensional quantum dots in the past, and have thus generated ground state energies to which the DMC energies can be compared. For three-dimensional quantum dots, few results are available for benchmarking.

The structure of the thesis

- The first chapter introduces the concept of object oriented programming, with focus on the methods used to develop the code for this thesis. The reader is assumed to have some background in programming, hence the very fundamentals of programming are not presented. A full documentation of the code is available in Ref. [9]. The code will thus not be covered in full detail. In addition to concepts from C++ programming, Python scripting will be introduced. General strategies regarding planning and structuring of code will also be covered in detail. The two most important Python scripts used in this thesis are documented in Appendix C and Appendix B.
- The second chapter serves as a theoretical introduction to QMC, discussing the necessary many-body theory in detail. Important theory which is required to understand the concepts introduced in later chapters are given the primary focus. The reader is assumed to have a basic understanding of Quantum Wave Mechanics. An introduction to the commonly used Dirac notation is given in Appendix A.
- Chapter 4 presents all the assumptions regarding the systems modelled in this thesis together with the aims regarding the generalization and optimization of the code. The strategies applied to achieve these aims will then be covered in high detail.
- Chapter 5 introduces the systems modelled in this thesis, that is, the quantum dots and atomic systems. The single-particle wave functions used to generate the trial wave functions for the different systems are presented together with the respective Hamiltonians.
- The results, along with the discussions and the conclusions mark the final part of this thesis. Results for up to 56 electrons in the two-dimensional quantum dot are presented and comparisons are made with two - and three-dimensional quantum dots for high and low frequency ranges. A brief display of a double-well quantum dot is then given before the atomic results are presented. The ground state energies of atoms up to krypton and molecules up to O_2 are then compared to experimental values. Concluding the results section, the molecular energies as a function of the separation of cores are compared to the Lennard-Jones 12-6 potential [10, 11]. Final remarks are then made regarding further work expanding on the work done in this thesis.

Part I

Theory

2

Scientific Programming

The introduction of the computer around World War II had a major impact on the mathematical fields of science. Previously unsolvable problems were now solvable. The question was no longer whether or not it was possible, but rather to what precision and with which method. The computer spawned a new branch of physics, *computational physics*, redefining the limits of our understanding of nature. The first major result of this synergy between science and computers came with the infamous atomic bombs *Little Boy* and *Fat Man*, a product of *The Manhattan Project* lead by *J. Robert Oppenheimer* [12].

2.1 Programming Languages

Programming is the art of writing computer programs. A program is a list of instructions for the computer, commonly referred to as *code*. It is in many ways similar to human-to-human instructions; for instance, different programming languages may be used to write instructions, such as *C++*, *Python* or *Java*, as long as the recipient is able to translate it. The instructions may be translated prior to the execution, i.e the code is *compiled*, or it may be translated *run-time* by an *interpreter*.

The native language of the computer is *binary*: Ones and zeros, which corresponds to high - and low voltage readings. Every character, number, color, etc. is on the lowest level represented by a sequence of binary numbers referred to as *bits*. In other words, programming languages serve as a bridge between the binary language of computers and a more manageable language for everyday programmers.

The closer the programming language at hand is to pure processor (CPU) instructions[1], the lower the *level* of the language is. This section will introduce high- and low level languages, focusing on C++ and Python, as these are the most commonly used languages throughout this thesis.

2.1.1 High-level Languages

Scientific programming involves a vast amount of different tasks, all from pure visualization and organization of data, to efficient memory allocation and processing. For less CPU-intensive tasks, the run time of the program is so small that the efficiency becomes irrelevant, leaving languages which prefer simplicity and structure over efficiency the optimal tool for the job. These languages are referred to as

[1]The CPU is the part of the computer responsible for flipping bits.

high-level languages [2].

High-level codes are often short snippets designed with a specific aim such as analyzing raw data, administrating input and output from different tools, creating a *Graphical User Interface* (GUI), or *gluing* different programs, which are meant to be run sequentially or in parallel, together into one. These short specific codes are referred to as *scripts*, hence high-level languages designed for these tasks are commonly referred to as *scripting languages* [13, 14].

Some examples of high-level languages are Python, Ruby, Perl, Visual Basic and UNIX shells. Excellent introductions to these languages are found throughout the World Wide Web.

Python

Named after the infamous *Monte Python's flying circus*, Python is an easy to learn open source interpreted programming language invented by Guido van Rossum around 1990. Python is designed for optimized development time by having a very clean and rigorous coding syntax [14, 15].

To demonstrate the simplicity of Python, consider the following simple expression

$$S = \sum_{i=1}^{100} i = 5050., \tag{2.1}$$

which is calculated in Python by the following expression:

```
1 print sum(range(101))
```

Executing the script yields the expected result:

```
~$ python Sum100Python.py
5050
```

For details regarding the basics of Python, see Ref. [15], or Ref. [13] for more advanced topics.

2.1.2 Low-level Languages

Scientific programming often involves solving complex equations. Complexity does not necessarily imply that the equations themselves are hard to understand; frankly, this is often not the case. In most cases of for example linear algebra, the problem at hand can be boiled down to solving $\mathbf{Ax} = \mathbf{b}$, however, the complexity lies in the dimensionality of the problem at hand. Matrix dimensions often range as high as millions. With each element being a double precision number (8 bytes or 64 bits), it is crucial to have full control of the memory and execute operations as efficiently as possible.

This is where lower level languages excel. Hiding few to none of the details, the power is in the hand of the programmer. This, however, comes at a price: More technical concepts such as memory pointers, declarations, compiling, linking, etc. makes the development process slower than that of a higher-level language.

[2] There are different definitions of the distinction between high- and low-level languages. Languages such as *assembly* is extremely complex and close to machine code, leaving all machine-independent languages as high-level in comparison. However, for the purpose of this thesis, the distinction will be set at a higher level than assembly.

Moreover, requesting access to an uninitialized element outside the bounds of an allocated array, Python will provide a detailed error message with proper traceback, whereas the compiled C++ code would simply crash at run-time, leaving nothing but a "segmentation fault" for the user. The payoff comes when the optimized program ends up running for days, in contrast to the high-level implementation which might end up running for months.

In addition, several options to optimize compiled machine code are available by having the compiler rearrange the way instructions are sent to the processor. Details regarding memory latency optimization will be discussed in Section 4.7.

C++

C++ is a programming language developed by Bjarne Stroustrup in 1983. It serves as an extension to the original *C* language, adding *object oriented* features, that is, classes etc. [16]. The following code is a C++ implementation of the sum in Eq. 2.1:

```
//Sum100C++.cpp
#include <iostream>

int main(){

        int S = 0;
        for (int i = 1; i <= 100; i++){
                S += i;
        }

        std::cout << S << std::endl;

        return 0;
}
```

```
~$ g++ Sum100C++.cpp -o sum100C++.x
~$ ./sum100C++.x
5050
```

Notice that, unlike Python, C++ requests an explicit declaration of S as an integer variable. This in turn tells the compiler the exact amount of memory needed to store the variable, opening up the possibility of efficient memory optimization.

Even though this is an extremely simply example, it illustrates the difference in coding styles between high- and low-level languages. The next section will cover the basics needed to understand object orientation in C++, and how it can be used to develop generalized coding frameworks.

2.2 Object Orientation

The concepts of classes and objects were introduced for the first time in the language *Simula 67*, developed by the Norwegian scientists Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Research Center [16]. Object orientation quickly became the state-of-the-art in programming, and has ever since enjoyed great success in numerous computational fields.

Object orientation ties everyday intuition into the programming language. Humans are object oriented without noticing it, in the sense that the focus is around *objects* of *classes*, for instance, an animal of a certain species, artifacts of a certain culture, people of a certain country, etc. This fact renders object

oriented codes extremely readable compared to what is possible with standard functions and variables. In addition to simple object structures, which in some sense can be achieved with standard C *structs*, classes provide functionality such as *inheritance* and accessibility control. These concepts will be the focus for the rest of the chapter, however, for the sake of completeness, a brief introduction to class syntax is given.

2.2.1 A Brief Introduction to Essential Concepts

Members

A class in its simplest representation is a collection of variables and functions unique to a specified object of the class[3]. When an object is created, it is uniquely identified by its own set of member variables.

An important member of a class is the object itself. In Python, this intrinsic mirror image is called `self`, and must, due to the interpreter nature of the language, be present in all function calls. In C++, it is available in any class member function as the `this` pointer. Making changes to `self` or `this` inside a function is equivalent to changing the object outside the function. It is nothing but a way for an object to have access to itself at any time.

Constructors

The constructor is the class function responsible for initializing new objects. When requesting a new instance of a class, a constructor is called with specific input parameters. In Python, the constructor is called `__init__()`, while in C++, the name of the constructor must match that of the class, e.g. `someClass::someClass()`[4].

Creating a new object is simply done by calling the constructor

```
1 someClass* someObject = new SomeClass("constructor argument");
```

The constructor can then assign values to member variables based on the input parameters.

Destructors

Opposite to constructors, destructors are responsible for deleting an object. In Python this is automatically done by the *garbage collector*, however, in C++ this is sometimes important in order to avoid memory leaks. The destructor is implemented as the function `someClass::~someClass()`, and is invoked by typing e.g. `delete someObject;`.

Reference [15] is suggested for further introduction to basic concepts of classes.

2.2.2 Inheritance

Consider the abstract idea of a keyboard: A board and keys (obviously). In object orientation terms, the keyboard *superclass* describes a board with keys. It is *abstract* in the sense that no specific information regarding the formation or functionality of the keys is needed in order to define the concept of a keyboard.

[3] Members can be shared by all class instances in C++ by using the static keyword. This will make the variables and function obtainable without initializing an object as well.

[4] The double colon notation means "someClass' member someClass()".

On the other hand, there exist different specific kinds of keyboards, e.g. computer keyboards or musical keyboards. Although quite different in design and function, they both relate to the same concept of a keyboard described previously: They are both *subclasses* of the same superclass, inheriting the basic concepts, but overloading the abstract parts with specific implementations.

Consider the following Python implementation of a keyboard superclass:

```
class Keyboard():

    #Set member variables keys and the number of keys
    #A subclass will override these with their own representation
    keys = None
    nKeys = 0

    #Constructor (function called when creating an object of this class)
    #Sets the number of keys and calls the setup function,
    #ensuring that no object of this abstract class can be created.
    def __init__(self, nKeys):
        self.nKeys = nKeys
        self.setupKeys()

    def setupKeys(self):
        raise NotImplementedError("Override me!")

    def pressKey(self, key):
        raise NotImplementedError("Override me!")

    def releaseKey(self, key):
        raise NotImplementedError("Override me!")
```

This class does not function on its own, and is clearly an abstract class meant for sub-classing. Examples of subclasses of keyboards are as mentioned computer - and musical keyboards. An easy way to visualize this inheritance relation is by drawing an *inheritance diagram* as in Figure 2.1. A python implementation of these subclasses are given on the next page.

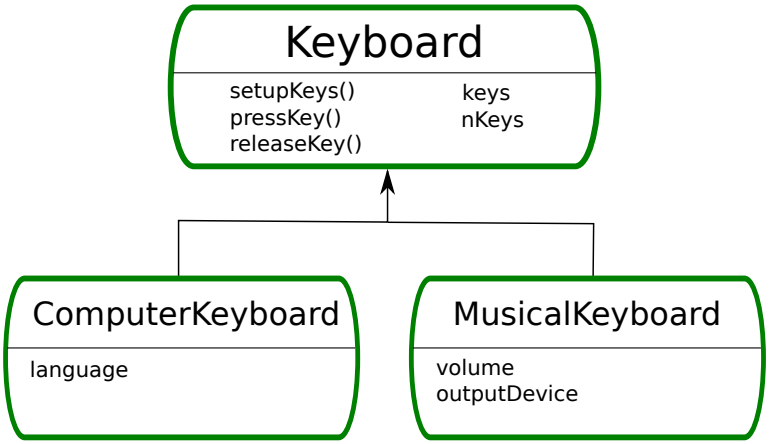


Figure 2.1: Inheritance diagram for a keyboard superclass. Class members are listed below the class name.

```
#The (keyboard) spesifies inheritance from Keyboard
class ComputerKeyboard(Keyboard):

    def __init__(self, language, nKeys):

        self.language = language

        #Use the superclass contructor to set the number of keys
        super(ComputerKeyboard, self).__init__(nKeys)


    def setupKeys(self):

        if self.language == "Norwegian":
            "Set up norwegian keyboard style"
        elif ...


    def pressKey(self, key):
        return self.keys[key]



#Dummy import for illustration purposes
from myDevices import Speakers

class MusicalKeyboard(Keyboard):

    def __init__(self, nKeys, volume):

        #Set the ouput device to speakers implemented elsewhere
        self.outputDevice = Speakers()
        self.volume = volume

        super(ComputerKeyboard, self).__init__(nKeys)


    def setupKeys(self):
        lowest = 27.5 #Hz
        step = 1.06 #Relative increase in Hz (neighbouring keys)

        self.keys = [lowest + i*step for i in range(self.nKeys)]


    def pressKey(self, key):

        #Returns a harmonic wave with frequency and amplitude
        #extracted from the pressed key and the volume level.
        outout = ...
        self.outputDevice.play(key, output)


   #Fades out the playing tune
    def releaseKey(self, key):
        self.outputDevice.fade(key)
```

It is clear from looking at the superclass that two keyboards are differentiated by the way their keys are set up. Not overriding the `setupKeys()` function would cause the generic superclass constructor to call the function which would raise an exception and close the program. These kinds of superclass member functions, which requires an implementation in order for the object to be constructed, are referred to as *pure virtual functions*. The other two functions do not necessarily have to be implemented, and are thus referred to as *virtual* functions. These topics will be discussed in more detail in the next section.

2.2.3 Pointers, Typecasting and Virtual Functions

A pointer is a hexadecimal number representing a memory address where some *type* of object is stored, for instance, an `int` at `0x7fff0882306c` (`0x` simply implies hexadecimal). Higher level languages like Python handles all the pointers and typesetting automatically. In low-level languages like C++, however, you need to control everything. This is commonly referred to as *type safety*.

Memory addresses are *global*, that is, they are shared throughout the program. This implies that changes done to a pointer to an object, e.g. `Keyboard* myKeyboard`, are applied everywhere the object is in use. This is particularly handy (or dangerous) when passing a pointer as an argument to a function, as changes applied to the given object will cause the object to remain changed after exiting the function. Passing non-pointer arguments would only change a local copy of the object which is destroyed upon exiting the function. The alternative would be to pass the reference to the object, e.g. `&myObject`, which passes the address just as in the case of pointers.

Creating a pointer to a keyboard object in C++ can be done in several ways. Following are three examples:

```
MusicalKeyboard* myKeyboard1 = new MusicalKeyboard(...);
Keyboard* myKeyboard2 = new MusicalKeyboard(...);
```

the second which is identical to

```
Keyboard* myKeyboard2 = (Keyboard*) myKeyboard1;
```

For reasons which will be explained in Section 2.2.4, the second expression is extremely handy. Technically, the subclass object is *type cast* to the superclass type. This is allowed since *any subclass is type-compatible with a pointer to it's superclass*. Any standard functions implemented in the subclass will not be directly callable from outside the object itself, and identical functions will be overwritten by the corresponding superclass functions, unless the functions are *virtual*. Flagging a function as virtual in the superclass will then tell the compiler not to overwrite this particular function when a subclass object is typecast to the superclass type.

Python does not support virtual functions in the same strict sense as C++, since typecasting is *automagic* in a language which is not type safe. The following example should bring some clarity to the current topic:

```
#include <iostream>
using namespace std;

class superClass{
public:
        // virtual = 0 implies pure virtual
        virtual void pureVirtual() = 0;
        virtual void justVirtual() {cout << "superclass virtual"    << endl;}
        void notVirtual()          {cout << "superclass notVirtual" << endl;}
};

class subClass : public superClass{
public:
        void pureVirtual() {cout << "subclass pure virtual override"     << endl;}
        void justVirtual() {cout << "subclass standard virtual override" << endl;}
        void notVirtual()  {cout << "subclass non virtual"               << endl;}
};

//Testfunc retrieves a superClass pointer, then calls all the functions.
void testFunc(superClass* someObject){
        someObject->pureVirtual(); someObject->justVirtual(); someObject->notVirtual();
}
```

```
1  int main(){
2
3          cout << "-Calling subClass object of type superClass*" << endl;
4          superClass* object = new subClass(); testFunc(object);
5
6          cout << endl << "-Calling subClass object of type subClass*" << endl;
7          subClass* object2 = new subClass(); testFunc(object);
8
9          cout << endl << "-Directly calling object of type subclass*" << endl;
10         object2->pureVirtual(); object2->justVirtual(); object2->notVirtual();
11
12         return 0;
13 }
```

Listing 2.1: This code demonstrates two different ways of creating an object of a subclass. In line 4, the object is created as a superclass type. Passing it to the function will cause the function to access the superclass functions unless they are declared as virtual. In line 7, the object is created as a subclass type, however, by passing it to the function, the object is typecast to the superclass type, rendering the two methods identical. In line 10, the functions are directly called, which ensures that only the subclass' functions are called regardless of them being virtual or not.

Executing the above code yields

```
~$ ./virtualFunctionsC++.x
-Calling subClass object of type superClass*
subclass pure virtual override
subclass standard virtual override
superclass notVirtual

-Calling subClass object of type subClass*
subclass pure virtual override
subclass standard virtual override
superclass notVirtual

-Directly calling object of type subclass*
subclass pure virtual override
subclass standard virtual override
subclass non virtual
```

As introduced in the keyboard example, the superclass in this case has a pure virtual function. Creating an object of the superclass would raise an error in the compilation, claiming that the superclass is abstract. Implemented subclasses must overload pure virtual functions in order to compile.

2.2.4 Polymorphism

The previous example involved a concept referred to as *polymorphism*, which is a concept closely connected to virtual functions and type casting. Because of the virtual nature of the superclass' functions, the function `testFunc()` does not a priori know its exact task. All it knows is that the received object has three functions (see the previous example). Exploiting this property is referred to as polymorphism.

Using polymorphism, codes can be written in an organized and versatile fashion. To further illustrate this, consider the following example from the Quantum Monte-Carlo (QMC) code developed in this thesis:

```
1  class Potential {
2  protected:
3      int n_p;
4      int dim;
5
6  public:
7      Potential(int n_p, int dim);
8
9      //Pure virtual function
10     virtual double get_pot_E(const Walker* walker) const = 0;
11
12 };
13
14 class Coulomb : public Potential {
15 public:
16
17     Coulomb(GeneralParams &);
18
19     //Returns the sum 1/r_i
20     double get_pot_E(const Walker* walker) const;
21
22 };
23
24 class Harmonic_osc : public Potential {
25 protected:
26     double w;
27
28 public:
29
30     Harmonic_osc(GeneralParams &);
31
32     //return the sum 0.5*w*r_i^2
33     double get_pot_E(const Walker* walker) const;
34
35 };
36
37 ...
```

Listing 2.2: An example from the QMC code. The superclass of potentials is defined by a number of particles (line 3), a dimension (line 4) and a pure virtual function for extracting the local energy of a given walker (line 10). Specific potentials are implemented as subclasses (line 14 and 24), simply overriding the pure virtual function with their own implementations.

Assume that an object `Potential* potential` is sent to an energy function. Since `get_pot_E()` is virtual, the potential can take any form; the energy function only checks whether it has an implementation or not. The code can easily be adapted to handle any combination of any potentials by storing the potential objects in a vector and simply accumulate the contributions:

```
1  //Simple compiler definition to clean up the code
2  #define potvec std::vector<Potential*>
3
4  class System {
5
6      double get_potential_energy(const Walker* walker) const;
7      potvec potentials;
8      ...
9  };
10
11 double System::get_potential_energy(const Walker* walker) const {
12     double potE = 0;
13
14     //Iterates through all loaded potentials and accumulate energies.
15     for (potvec::iterator pot = potentials.begin(); pot != potentials.end(); ++pot) {
16         potE += (*pot)->get_pot_E(walker);
17     }
18
19     return potE;
20 }
```

2.2.5 Const Correctness

In the previous `Potential` code example, function declarations with the `const` flag were used. As mentioned in the section on pointers, passing pointers to functions are dangerous business. If an object is flagged with `const` on input, e.g. `void f(const x)`, the function itself cannot alter the value of `x`. If it does, the compiler will abort. This property is referred to as *const correctness*, and serve as a safeguard guaranteeing that nothing will happen to `x` as it passes through `f`. This is practical in situations where changes to an object are unwanted.

If you declare a member function itself with `const` on the right hand side, e.g. `void class::f(x) const`, no changes may be applied to class members inside this specific function. For instance, in the potential energy functions, all that is requested is to evaluate a function at a given set of coordinates; there is no need to change anything, hence the const correctness is applied to the function.

In other words: const correctness works as a safeguard preventing changes to values which should remain unchanged. A change in such a variable is then followed by a compiler error instead of unforeseen consequences.

2.2.6 Accessibility levels and Friend classes

When a C++ class is declared, each member needs to be related to an *accessibility level*. The three accessibility levels in C++ are

(i) **Public:** The variable or function may be accessed from anywhere the object is available.

(ii) **Private:** The variable or function may be accessed only from within the class itself.

(iii) **Protected:** As for private, but also accessible from subclasses of the class.

As an example, any standardized application (app) needs the `app::execute_app()` function to be public, i.e. accessible from the main file. On the other hand, `app::dump_progress()` should be controlled by the application itself, and should thus be private, or protected in case the application has subclasses.

There is one exception to the rule of protected - and private variables. In certain situations where a class needs to access private variables from another class, but going full public is undesired, the latter class can *friend* the first class. This implies that the first class has access to the second class' private members.

In the QMC code developed in this thesis, the distribution is calculated by a class `Distrubution`. In order to achieve this, the protected members of `QMC` need to be made available to the `Distrubution` class. This is implemented in the following fashion:

```
1
2  class QMC {
3  protected:
4
5      arma::mat dist; //!< Matrix holding positional data for the distribution.
6      int last_inserted; //!< Index of last inserted positional data.
7      ...
8
9  public:
10     ...
11
12     //Gives Distribution access to protected members of QMC.
13     friend class Distribution;
14
15 };
16
17 void Distribution::finalize() {
18
19     //scrap out all the over-allocated space (DMC)
20     qmc->dist.resize(qmc->last_inserted, dim);
21
22     if (dim == 3) {
23         generate_distribution3D(qmc->dist, qmc->n_p);
24     } else {
25         generate_distribution2D(qmc->dist, qmc->n_p);
26     }
27
28     qmc->dist.reset();
29
30 }
```

Listing 2.3: An example from the QMC code. The distribution class needs access to the private members of QMC. This is achieved in line 13 by friending the distribution class.

Codes could be developed without using `const` flags and with solely public members, however, in that case it is very easy to put together a very disorganized code, with pointers going everywhere and functions being called in all sorts of contexts. This is especially true if there are several developers on the same project.

Clever use of accessibility levels will make codes easier to develop in an organized and intuitive fashion Put in other words: If you have to break an accessibility level to implement a desired functionality, there probably exists a better way of implementing it.

2.2.7 Example: PotionGame

To conclude this section on object orientation, consider the following code for a player vs. player game:

```
#playerClass.py
import random

class Player:

    def __init__(self, name):
        #Player initialized at full health and full energy.
        self.health = self.energy = 100

        self.name = name
        self.dead = False

        #Player initialized with no available potions.
        self.potions = []


    def addPotion(self, potion):
        self.potions.append(potion)


    #Selects the given potion and consumes it. The potion needs to know
    #the player it should affect, hence we send 'self' as an argument.
    def usePotion(self, potionIndex):
        print "%s consumes %s." % (self.name, self.potions[potionIndex].name)

        selectedPotion = self.potions[potionIndex]
        selectedPotion.applyPotionTo(self)          #Self explainatory!
        self.potions.pop(potionIndex)               #Removed the potion.


    def changeHealth(self, amount):
        self.health += amount

        #Cap health at [0,100].
        if self.health > 100: self.health = 100
        elif self.health <= 0:
            self.health = 0
            self.dead = True;


    def changeEnergy(self, amount):
        self.energy += amount

        #Cap energy at [0,100].
        if self.energy > 100: self.energy = 100
        elif self.energy < 0: self.energy = 0


    #Lists the potions to the user.
    def displayPotions(self):
        if not self.potions:
            print "  No potions available"

        for potion in self.potions: print "  " + potion.name


    def attack(self, targetPlayer):

        energyCost = 55

        if self.energy < energyCost:
            print "%s: Insuficcient energy to attack." % self.name; return

        damage = 40 + random.randint(-10, 10)

        targetPlayer.changeHealth(-damage)
        self.changeEnergy(-energyCost)

        print "%s hit %s for %s using %s energy" % (self.name,
                                                    targetPlayer.name,
                                                    damage, energyCost)
```

```
1  #potionClass.py
2
3  class Potion:
4
5      def __init__(self, amount):
6          self.amount = amount
7          self.setName()
8
9      def applyPotionTo(self, player):
10         raise NotImplementedError("Member function applyPotion not implemented.")
11
12     #This function should be overwritten
13     def setName(self):
14         self.name = "Undefined"
15
16
17 class HealthPotion(Potion):
18
19     #Constructor is inherited
20
21     #Calls back to the player object's functions to change the health
22     def applyPotionTo(self, player):
23         player.changeHealth(self.amount)
24
25     def setName(self):
26         self.name = "Health Potion (%d)" % self.amount
27
28
29 class EnergyPotion(Potion):
30
31     def applyPotionTo(self, player):
32         player.changeEnergy(self.amount)
33
34     def setName(self):
35         self.name = "Energy Potion (%d)" % self.amount
```

The `Player` class keeps track of everything a player needs of personal data, such as the name (line 10), health- and energy levels (line 8), potions etc. Bringing another player into the game is simply done by creating another `Player` object. A player holds a number of `Potion` objects in a list (line 14). These objects are subclass implementations of the abstract potion class, which overwrites the virtual function describing the potion's effect on a given player object. This is demonstrated in lines 23 and 32. This subclass hierarchy of potions makes it incredibly easy to implement new ones.

The power of object orientation shines through in this simple example. The readability is very good, and does not falter if numerous potions or players are brought to the table.

In this section the focus has not been solely on scientific computing, but rather on the use of object orientation in general. The interplay between the potions and the players in the current example closely resembles the interplay between the QMC solver and the potentials introduced previously. Whether games or scientific programs are at hand, the methods used in the programming remain the same.

On the following page, a game is constructed using the `Player` and `Potion` classes. In lines 15-22, three players are initialized with a set of potions, from where they battle each other one round. The syntax is extremely transparent. Adding a fourth player is simply a matter of adding a new line of code. The output of the game is displayed below the code.

```
#potionGameMain.py

from potionClass import *
from playerClass import *

def roundOutput(n, *players):
        header= "Round %d: " % n
        print header.replace('0','start')
        for player in players:
                print "  %s (hp/e=%d/%d):" % (player.name, player.health, player.energy)
                player.displayPotions()
                print


Sigve = Player('Sigve');
Sigve.addPotion(EnergyPotion(10));

Jorgen = Player('Jorgen')
Jorgen.addPotion(HealthPotion(20)); Jorgen.addPotion(EnergyPotion(20))

Karl = Player('Karl')
Karl.addPotion(HealthPotion(20))

#Initial output
roundOutput(0, Sigve, Jorgen, Karl)

#Round one: Each player empties their arsenal
Sigve.attack(Jorgen); Sigve.attack(Karl); Sigve.usePotion(0); Sigve.attack(Karl)
print

Karl.usePotion(0); Karl.attack(Sigve)
print

Jorgen.attack(Karl); Jorgen.usePotion(1); Jorgen.attack(Sigve)
print

roundOutput(1, Sigve, Jorgen, Karl)
#Round one end.
#...
```

```
Round start:
  Sigve (hp/e=100/100):
  Energy Potion (10)

  Jorgen (hp/e=100/100):
  Health Potion (20)
  Energy Potion (20)

  Karl (hp/e=100/100):
  Health Potion (20)

Sigve hit Jorgen for 40 using 55 energy
Sigve: Insuficcient energy to attack.
Sigve consumes Energy Potion (10).
Sigve hit Karl for 35 using 55 energy

Karl consumes Health Potion (20).
Karl hit Sigve for 41 using 55 energy

Jorgen hit Karl for 44 using 55 energy
Jorgen consumes Energy Potion (20).
Jorgen hit Sigve for 47 using 55 energy

Round 1:
  Sigve (hp/e=12/0):
  No potions available

  Jorgen (hp/e=60/10):
  Health Potion (20)

  Karl (hp/e=41/45):
  No potions available
```

2.3 Structuring the code

Structuring a code is a matter of making choices based on the complexity of the code. If the code is short and has a direct purpose, for instance, to calculate the sum from Eq. (2.1), the structure is not an issue at all, given that reasonable variable names are used. However, if the code is more complex and the methods used are specific implementations of a more general case, e.g. potentials, code structuring becomes very important. For details about the structuring of the code used in this thesis, see the documentation provided in Ref. [9].

2.3.1 File Structures

Not only does the potion game example demonstrate clean object orientation, but also standard file structuring by splitting the different classes and the main application into separate files. In a small code, like for example the potion game, the gain of transparency is not immense, however, when the class structures span thousands of lines, having a good structure is crucial to the development process, the code's readability, and the management in general.

Developing codes in scientific scenarios often involve large frameworks. For example, when coding molecular dynamics, several collision models, force models etc. are implemented alongside the main solver. In the case of Markow Chain Monte Carlo methods, different diffusion models (sampling rules) may be selectable. Even though these models are implemented using polymorphism, the code still gets messy when the amount of classes gets large.

In these scenarios, it is common to gather the implementations of the different classes in separate files (as for the potion game). This would be for purely cosmetic reasons if the process of writing code was linear, however, empirical evidence suggests otherwise: At least half the time is spent debugging, going back and forth between files.

A standard way to organize code is to have all the source code gathered in an *src* folder, with one folder per distinct class. Subclasses should appear as folders inside the superclass folder. Figure 2.2 shows an example setup for the framework of an object oriented code.

Another gain by this structuring files this way, is that tools such as Make, QMake, etc. ensures that only the files that actually changed will be recompiled. This saves a lot of time in the development process once the total compilation time starts taking several minutes.

2.3.2 Class Structures

In scientific programming, the simulated process often has a physical or mathematical interpretation. Some examples are, for instance, atoms in molecular dynamics and Bose-Einstein condensates, random walkers in diffusion processes, etc. Implementing classes representing these quantities will shorten the gap between the mathematical formulation of the problem and the implementation.

In addition, quantities such as the energy, entropy and temperature, are all calculated based on equations from statistical mechanics, quantum mechanics, or similar. Having class methods representing these calculations will again shorten the gap. There is no question what is done when the `system::get_potential_E` method is called, however, if some random loop appears in the main solver, an initial investigation is required in order to understand the flow of the code.

As described in Section 2.2.4, abstracting for example the potential energy function into a system object opens up the possibility of generalizing the code to any potential without altering the main solver. Structure is in other words vital if readability and versatility is desired.

Planning the code structure comes in as a key part of any large coding project. For details regarding the planning of the code in this thesis, see Section 4.1.

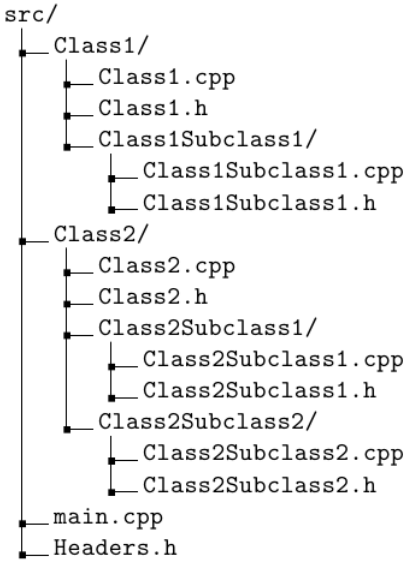


Figure 2.2: An illustration of a standard way to organize source code. The file endings represent C++ code.

3

Quantum Monte-Carlo

Quantum Monte-Carlo (QMC) is a method for solving Schrödinger's equation using statistical *Markov Chain* (random walk) simulations. The statistical nature of Quantum Mechanics makes Monte-Carlo methods the perfect tool not only for accurately estimating observables, but also for extracting interesting quantities such as densities, i.e. probability distributions.

There are multiple strategies which can be used in order to deduce the virtual[1] dynamics of QMC, some of which are more mathematically complex than others. In this chapter the focus will be on modelling the Schrödinger equation as a diffusion problem in complex (Wick rotated) time. Other more condensed mathematical approaches does not need the Schrödinger equation at all, however, for the purpose of completeness, this approach will be mentioned only briefly in Section 3.2.3.

In this chapter, *Dirac Notation* will be used. See Appendix A for an introduction. The equations will be in atomic units, i.e. $\hbar = m_e = e = 4\pi\epsilon_0 = 1$, where m_e and ϵ_0 are the electron mass and the vacuum permittivity, respectively.

3.1 Modelling Diffusion

Like any phenomena involving a probability distribution, Quantum Mechanics can be modelled by a diffusion process. In Quantum Mechanics, the distribution is given by $|\Phi(\mathbf{r}, t)|^2$, the wave function squared. The diffusing elements of interest are the particles in the system at hand.

The basic idea is to introduce an ensemble of *random walkers*, in which each walker is represented by a position in space at a given time. Once the walkers reach equilibrium, averaging values over the paths of the ensemble will yield average values corresponding to the probability distribution governing the movement of individual walkers. In other words: Counting every walker's contribution within a small volume $\mathrm{d}\mathbf{r}$ will correspond to $|\Phi(\mathbf{r}, t)|^2\mathrm{d}\mathbf{r}$ in the limit of infinite walkers.

Such random movement of walkers are referred to as a *Brownian motion*, named after the British botanist R. Brown, originating from his experiments on plant pollen dispersed in water. Markov chains are a subtype of Brownian motion, where a walkers next move is independent of previous moves. This is the stochastic process in which QMC is described.

[1] As will be shown, the time parameter in QMC does not correspond to physical time, but rather an imaginary axis at a fixed point in time. Whether nature operates on a complex time plane or not is not testable in a laboratory, and the origin of the probabilistic nature of Quantum Mechanics will thus remain a philosophical problem.

The purpose of this section is to motivate the use of diffusion theory in Quantum Mechanics, and to derive the sampling rules needed in order to model Quantum Mechanical distributions by diffusion of random walkers correctly.

3.1.1 Stating the Schrödinger Equation as a Diffusion Problem

Consider the time-dependent Schrödinger equation for an arbitrary wave function $\Phi(\mathbf{r}, t)$ using an arbitrary energy shift E'

$$-\frac{\partial\Phi(\mathbf{r},t)}{i\partial t} = (\widehat{\mathbf{H}} - E')\Phi(\mathbf{r},t). \tag{3.1}$$

Given that the Hamiltonian is time-independent, the formal solution is found by separation of variables in $\Phi(\mathbf{r}, t)$ [17]

$$\widehat{\mathbf{H}}\Phi(\mathbf{r},t_0) = E\Phi(\mathbf{r},t_0), \tag{3.2}$$

$$\Phi(\mathbf{r},t) = \exp\left(-i(\widehat{\mathbf{H}} - E')(t-t_0)\right)\Phi(\mathbf{r},t_0). \tag{3.3}$$

From Eq. (3.3) it is apparent that the time evolution operator is on the form

$$\widehat{\mathbf{U}}(t,t_0) = \exp\left(-i(\widehat{\mathbf{H}} - E')(t-t_0)\right). \tag{3.4}$$

The time-independent equation is solved for the ground state energy through methods such as *Full Configuration Interaction* [18] or similar methods based on diagonalizing the Hamiltonian. The time-dependent equation is used by methods such as *Time-Dependent Multi-Configuration Hartree-Fock* [19] in order to obtain the time-development of quantum states. However, neither of the equations originate from, or resemble, diffusion equations.

The original Schrödinger equation, however, does resemble a diffusion equation in complex time[2]. It can not be treated as a true diffusion equation, since the time evolved quantity, the wave function, is not a probability distribution unless it is squared. However, the equation involves a time derivative and a Laplacian, strongly indicating some sort of connection to a diffusion process.

Substituting complex time with a parameter τ and choosing the energy shift E' equal to the true ground state energy of $\widehat{\mathbf{H}}$, E_0, the time evolution operator in Eq. (3.4) becomes the *projection operation* $\widehat{\mathbf{P}}(\tau)$, whose choice of name will soon be apparent. In other words:

$$t \to it \equiv \tau,$$
$$\widehat{\mathbf{U}}(t,0) \to \exp\left(-(\widehat{\mathbf{H}} - E_0)\tau\right) \equiv \widehat{\mathbf{P}}(\tau).$$

Consider an arbitrary wave function $\Psi_T(\mathbf{r})$. Applying the new operator yields a new wave function $\Phi(\mathbf{r}, \tau)$ in the following manner

[2]The physical time diffusion equation evolves the squared wave function, and can be deduced from the quantum continuity equation combined with Fick's laws of diffusion [20].

$$\begin{aligned}\Phi(\mathbf{r},\tau) &= \langle \mathbf{r}|\, \widehat{\mathbf{P}}(\tau)\, |\Psi_T\rangle \\ &= \langle \mathbf{r}|\exp\left(-(\widehat{\mathbf{H}}-E_0)\tau\right)|\Psi_T\rangle\,.\end{aligned} \tag{3.5}$$

Expanding the arbitrary state in the eigenfunction of $\widehat{\mathbf{H}}$, $|\Psi_i\rangle$, yields

$$\begin{aligned}\Phi(\mathbf{r},\tau) &= \sum_i C_i \langle r|\exp\left(-(\widehat{\mathbf{H}}-E_0)\tau\right)|\Psi_i\rangle \\ &= \sum_i C_i\Psi_i(\mathbf{r})\exp\left(-(E_i-E_0)\tau\right) \\ &= C_0\Psi_0(x)+\sum_{i=1}^{\infty} C_i\Psi_k(x)e^{-\delta E_i\tau},\end{aligned} \tag{3.6}$$

where $C_i = \langle\Psi_i|\Psi_T\rangle$ and $\delta E_i = E_i - E_0 \geq 0$. In the limit where τ goes to infinity, the ground state is the sole survivor of the expression, hence the name projection operator. In other words:

$$\begin{aligned}\lim_{\tau\to\infty}\Phi(\mathbf{r},\tau) &= \lim_{\tau\to\infty}\langle \mathbf{r}|\,\widehat{\mathbf{P}}(\tau)\,|\Psi_T\rangle \\ &= C_0\Psi_0(x).\end{aligned} \tag{3.7}$$

The projection operator transforms an arbitrary wave function $\Psi_T(\mathbf{r})$, from here on referred to as the *trial wave function*, into the true ground state, given that the overlap C_0 is non-zero.

In order to model the projection with Markov chains, the process needs to be split into subprocesses which in turn can be described as transitions in the Markov chain. Introducing a time-step $\delta\tau$, the projection operator can be rewritten as

$$\widehat{\mathbf{P}}(\tau) = \prod_{k=1}^{n}\exp\left(-(\widehat{\mathbf{H}}-E_0)\delta\tau\right), \tag{3.8}$$

where $n = \tau/\delta\tau$. An important property to notice is that

$$\widehat{\mathbf{P}}(\tau+\delta\tau) = \exp\left(-(\widehat{\mathbf{H}}-E_0)\delta\tau\right)\widehat{\mathbf{P}}(\tau). \tag{3.9}$$

Using this relation in combination with Eq. (3.5), the effect of the projection operator during a single time-step is revealed:

$$\begin{aligned}\Phi(\mathbf{r},\tau+\delta\tau) &= \langle \mathbf{r}|\,\widehat{\mathbf{P}}(\tau+\delta\tau)\,|\Psi_T\rangle \\ &= \langle \mathbf{r}|\exp\left(-(\widehat{\mathbf{H}}-E_0)\delta\tau\right)\widehat{\mathbf{P}}(\tau)\,|\Psi_T\rangle \\ &= \langle \mathbf{r}|\exp\left(-(\widehat{\mathbf{H}}-E_0)\delta\tau\right)|\Phi(\tau)\rangle \\ &= \int_{\mathbf{r}'}\langle \mathbf{r}|\exp\left(-(\widehat{\mathbf{H}}-E_0)\delta\tau\right)|\mathbf{r}'\rangle\,\langle\mathbf{r}'|\Phi(\tau)\rangle\,\mathrm{d}\mathbf{r}' \\ &= \int_{\mathbf{r}'}\langle \mathbf{r}|\exp\left(-(\widehat{\mathbf{H}}-E_0)\delta\tau\right)|\mathbf{r}'\rangle\,\Phi(\mathbf{r}',\tau)\mathrm{d}\mathbf{r}',\end{aligned} \tag{3.10}$$

where a complete set of position states were introduced.

For practical purposes, E_0 needs to be substituted with an approximation E_T to the ground state energy, commonly referred to as the *trial energy*, in order to avoid self consistency. From Eq. (3.6) it is apparent that the projection will still converge as long as $E_T < E_1$, that is, the trial energy is less than that of the first excitation. The resulting expression reads:

$$\Phi(\mathbf{r}, \tau + \delta\tau) = \int_{\mathbf{r}'} \langle \mathbf{r}| \exp\left(-(\widehat{\mathbf{H}} - E_T)\delta\tau\right) |\mathbf{r}'\rangle \, \Phi(\mathbf{r}', \tau) \mathrm{d}\mathbf{r}' \tag{3.11}$$

$$\equiv \int_{\mathbf{r}'} G(\mathbf{r}, \mathbf{r}'; \delta\tau)\Phi(\mathbf{r}', \tau)\mathrm{d}\mathbf{r}'. \tag{3.12}$$

The equations above are well suited for Markov Chain models, as an ensemble of walkers can be iterated by transitioning between configurations $|\mathbf{r}\rangle$ and $|\mathbf{r}'\rangle$ with probabilities given by the *Green's function*, $G(\mathbf{r}, \mathbf{r}'; \delta\tau)$.

The effect of the Green's function from Eq. (3.12) on individual walkers is not trivial. In order to relate the Green's function to well-known processes, the exponential is split into two parts, one containing only the kinetic energy operator $\widehat{\mathbf{T}} = -\frac{1}{2}\nabla^2$, and the second containing the potential energy operator $\widehat{\mathbf{V}}$ and the energy shift. This is known as the *short time approximation* [21]

$$G(\mathbf{r}, \mathbf{r}'; \delta\tau) = \langle \mathbf{r}| \exp\left(-(\widehat{\mathbf{H}} - E_T)\delta\tau\right) |\mathbf{r}'\rangle \tag{3.13}$$

$$= \langle \mathbf{r}| e^{-\widehat{\mathbf{T}}\delta\tau} e^{-(\widehat{\mathbf{V}} - E_T)\delta\tau} |\mathbf{r}'\rangle + \frac{1}{2}[\widehat{\mathbf{V}}, \widehat{\mathbf{T}}]\delta\tau^2 + \mathcal{O}(\delta\tau^3). \tag{3.14}$$

The first exponential describes a transition of walkers governed by the Laplacian, which is a diffusion process. The second exponential is linear in position space and is thus a weighing function responsible for distributing the correct weights to the corresponding walkers. In other words:

$$G_{\mathrm{Diff}} = e^{\frac{1}{2}\nabla^2\delta\tau}, \tag{3.15}$$

$$G_{\mathrm{B}} = e^{-(\widehat{\mathbf{V}} - E_T)\delta\tau}, \tag{3.16}$$

where B denotes *branching*. The reasons for this name together with the complete process of modelling weights by branching will be covered in detail in Section 3.5.

The flow of QMC is then to use these Green's functions to propagate the ensemble of walkers into the next time-step. The final distribution of walkers will correspond to that of the direct solution of the Schrödinger equation, given that the time-step is sufficiently small, and the number of cycles n are sufficiently large. These constraints will be covered in more detail later.

Incorporating only the effect of Eq. (3.15) results in a method called *Variational Monte-Carlo* (VMC). Including the branching term as well results in *Diffusion Monte-Carlo* (DMC). These methods will be discussed in Sections 3.8 and 3.9, respectively. In either of these methods, diffusion is a key process.

3.2 Solving the Diffusion Problem

The diffusion problem introduced in the previous section uses a symmetric kinetic energy operator implying an *isotropic diffusion*, however, a more efficient kinetic energy operator can be introduced without

violating the original equations, resulting in an *anisotropic diffusion* governed by the *Fokker-Planck equation*. These models will be the topic of this section.

For details regarding the transition from isotropic to anisotropic diffusion, see Section 3.2.3.

3.2.1 Isotropic Diffusion

Isotropic diffusion is a process in which diffusing particles sees all directions as an equally probable path. Eq. (3.17) is an example of this. The isotropic diffusion equation is

$$\frac{\partial P(\mathbf{r},t)}{\partial t} = D\nabla^2 P(\mathbf{r},t). \tag{3.17}$$

This is the simplest form of a diffusion equation, that is, the case with a linear *diffusion constant*, D, and no drift terms.

From Eq. (3.15) it is clear that the value of the diffusion constant is $D = \frac{1}{2}$, originating from the term scaling the Laplacian in the Schrödinger Equation. An important point is that closed form expressions for the Green's function exists. This closed form expression in the isotropic case is a Gaussian distribution with variance $2D\delta t$ [21]

$$G^{\text{ISO}}_{\text{Diff}}(i \rightarrow j) \propto e^{-|\mathbf{r}_i - \mathbf{r}_j|^2/4D\delta\tau}. \tag{3.18}$$

These equations describe the diffusion process theoretically, however, in order to achieve specific sampling rules for the walkers, a connection between the time-dependence of the distribution and the time-dependence of an individual walker's components in configuration space is needed. This connection is given in terms of a stochastic differential equation called *The Langevin Equation*.

The Langevin Equation for isotropic diffusion

The Langevin Equation is a stochastic differential equation used in physics to relate the time dependence of a distribution to the time-dependence of the degrees of freedom in a system. For isotropic diffusion, solving the Langevin equation using a Forward Euler approximation for the time derivative results in the following relation:

$$x_{i+1} = x_i + \xi, \qquad \begin{aligned} \text{Var}(\xi) &= 2D\delta t, \\ \langle \xi \rangle &= x_i, \end{aligned} \tag{3.19}$$

where ξ is a normal distributed number whose variance matches that of the Green's function in Eq. (3.18). This relation is in agreement with the isotropy of Eq. (3.17) in the sense that the displacement is symmetric around the current position.

3.2.2 Anisotropic Diffusion and the Fokker-Planck equation

Anisotropic diffusion, in contrast to isotropic diffusion, does not see all directions as equally probable. An example of this is diffusion according to the *Fokker-Planck Equation*, that is, diffusion with a drift term, $\mathbf{F}(\mathbf{r},t)$, responsible for pushing the walkers in the direction of configurations with higher probabilities, and thus closer to an equilibrium state. The Fokker-Planck equation reads:

$$\frac{\partial P(\mathbf{r},t)}{\partial t} = D\nabla \cdot \Big[\Big(\nabla - \mathbf{F}(\mathbf{r},t)\Big)P(\mathbf{r},t)\Big]. \tag{3.20}$$

As will be derived in detail in Section 3.2.3, using the Fokker-Planck equation does not violate the original Schrödinger equation, but changes the representation of the ensemble of walkers to a mixed density. This means that QMC can be run with Fokker-Planck diffusion, leading to a more optimized way of sampling due to the drift term.

As mentioned introductory, the goal of the Markov process is convergence to a stationary state. Using this criteria, the expression for the drift term can be found. A stationary state is obtained when the left hand side of Eq. (3.20) is zero. This yields:

$$\nabla^2 P(\mathbf{r},t) = P(\mathbf{r},t)\nabla \cdot \mathbf{F}(\mathbf{r},t) + \mathbf{F}(\mathbf{r},t) \cdot \nabla P(\mathbf{r},t).$$

In order to get cancellation in the remaining terms, the Laplacian term on the right-hand side must cancel out the terms on the left. This implies that the drift term needs to be on the form $\mathbf{F}(\mathbf{r},t) = g(\mathbf{r},t)\nabla P(\mathbf{r},t)$. Inserting this yields

$$\nabla^2 P(\mathbf{r},t) = P(\mathbf{r},t)\frac{\partial g(\mathbf{r},t)}{\partial P(\mathbf{r},t)}\Big|\nabla P(\mathbf{r},t)\Big|^2 + P(\mathbf{r},t)g(\mathbf{r},t)\nabla^2 P(\mathbf{r},t) + g(\mathbf{r},t)\Big|\nabla P(\mathbf{r},t)\Big|^2.$$

The factors in front of the Laplacian suggests using $g(\mathbf{r},t) = 1/P(\mathbf{r},t)$. A quick check reveals that this also cancels the gradient terms. The resulting expression for the drift term becomes

$$\begin{aligned} \mathbf{F}(\mathbf{r},t) &= \frac{1}{P(\mathbf{r},t)}\nabla P(\mathbf{r},t) \\ &= \frac{2}{|\psi(\mathbf{r},t)|}\nabla|\psi(\mathbf{r},t)|. \end{aligned} \tag{3.21}$$

In QMC, the drift term is commonly referred to as the *quantum force*. This is due to the fact that it is responsible for pushing the walkers into regions of higher probabilities, analogous to a force in Newtonian mechanics.

Another strength of the Fokker-Planck equation is that even though the equation itself is more complicated, its Green's function still has a closed form solution. This means that it can be evaluated efficiently. If this was not the case, the practical value would be reduced dramatically. The reason for this will become clear in Section 3.4. The closed form solution reads [21]

$$G^{\mathrm{FP}}_{\mathrm{Diff}}(i \to j) \propto e^{-(x_i - x_j - D\delta\tau F(x_i))^2/4D\delta\tau}. \tag{3.22}$$

As expected, the Green's function is no longer symmetric.

The Langevin Equation for the Fokker-Planck equation

The Langevin equation in the case of a Fokker-Planck Equation has the following form

$$\frac{\partial x_i}{\partial t} = DF(\mathbf{r})_i + \eta, \tag{3.23}$$

where η is a so-called *noise term* from stochastic processes. Solving this equation using the same method as for the isotropic case yields the following sampling rules

$$x_{i+1} = x_i + \xi + DF(\mathbf{r})_i \delta t, \tag{3.24}$$

where ξ is the same as for the isotropic case. Observe that when the drift term goes to zero, the Fokker-Planck - and isotropic solutions are equal, just as required. For more details regarding the Fokker-Planck Equation and Langevin equations, see Refs. [22–24].

3.2.3 Connecting Anisotropic - and Isotropic Diffusion Models

To this point, it might seem far-fetched that switching the diffusion model to a Fokker-Planck diffusion does not violate the original equation, i.e. the complex time Schrödinger equation (the projection operator). Introducing the distribution function $f(\mathbf{r},t) = \Phi(\mathbf{r},t)\Psi_T(\mathbf{r})$, restating the imaginary time Schrödinger equation in terms of $f(\mathbf{r},t)$ yields

$$\begin{aligned} -\frac{\partial}{\partial t} f(\mathbf{r},t) \;=\; \Psi_T(\mathbf{r})\Big[-\frac{\partial}{\partial t}\Phi(\mathbf{r},t)\Big] &= \Psi_T(\mathbf{r})\left(\widehat{\mathbf{H}} - E_T\right)\Phi(\mathbf{r},t) \\ &= \Psi_T(\mathbf{r})\left(\widehat{\mathbf{H}} - E_T\right)\Psi_T(\mathbf{r})^{-1} f(\mathbf{r},t) \\ &= -\frac{1}{2}\Psi_T(\mathbf{r})\nabla^2\left(\Psi_T(\mathbf{r})^{-1} f(\mathbf{r},t)\right) + \widehat{\mathbf{V}} f(\mathbf{r},t) - E_T f(\mathbf{r},t). \end{aligned} \tag{3.25}$$

Expanding the Laplacian term further reveals

$$\begin{aligned} K(\mathbf{r},t) &\equiv -\frac{1}{2}\Psi_T(\mathbf{r})\nabla^2\left(\Psi_T(\mathbf{r})^{-1} f(\mathbf{r},t)\right) \\ &= -\frac{1}{2}\Psi_T(\mathbf{r})\nabla\cdot\left(\nabla\left[\Psi_T(\mathbf{r})^{-1} f(\mathbf{r},t)\right]\right), \qquad (3.26) \\ \nabla\left[\Psi_T(\mathbf{r})^{-1} f(\mathbf{r},t)\right] &= -\Psi_T(\mathbf{r})^{-2}\nabla\Psi_T(\mathbf{r}) f(\mathbf{r},t) + \Psi_T(\mathbf{r})^{-1}\nabla f(\mathbf{r},t). \qquad (3.27) \end{aligned}$$

Combining these equations and applying the product rule numerous times yield

$$\begin{aligned} K(\mathbf{r},t) &= -\frac{1}{2}\Psi_T(\mathbf{r})\Big[\left(2\Psi_T(\mathbf{r})^{-3}\left|\nabla\Psi_T(\mathbf{r})\right|^2 f(\mathbf{r},t)\right. \\ &\quad -\Psi_T(\mathbf{r})^{-2}\nabla^2\Psi_T(\mathbf{r}) f(\mathbf{r},t) \\ &\quad \left.-\Psi_T(\mathbf{r})^{-2}\nabla\Psi_T(\mathbf{r})\cdot\nabla f(\mathbf{r},t)\right) \\ &\quad +\Psi_T(\mathbf{r})^{-1}\nabla^2 f(\mathbf{r},t) \\ &\quad -\Psi_T(\mathbf{r})^{-2}\nabla\Psi_T(\mathbf{r})\cdot\nabla f(\mathbf{r},t)\Big] \\ &= -\left|\Psi_T(\mathbf{r})^{-1}\nabla\Psi_T(\mathbf{r})\right|^2 f(\mathbf{r},t) \\ &\quad +\frac{1}{2}\Psi_T(\mathbf{r})^{-1}\nabla^2\Psi_T(\mathbf{r}) f(\mathbf{r},t) \\ &\quad +\Psi_T(\mathbf{r})^{-1}\nabla\Psi_T(\mathbf{r})\cdot\nabla f(\mathbf{r},t) \\ &\quad -\frac{1}{2}\nabla^2 f(\mathbf{r},t). \end{aligned}$$

Introducing the following identity helps clean up the messy calculations:

$$-\left|\Psi_T(\mathbf{r})^{-1}\nabla\Psi_T(\mathbf{r})\right|^2 = \nabla\cdot\left(\Psi_T(\mathbf{r})^{-1}\nabla\Psi_T(\mathbf{r})\right) - \Psi_T(\mathbf{r})^{-1}\nabla^2\Psi_T(\mathbf{r}),$$

which inserted into the expression for $K(\mathbf{r},t)$ reveals

$$\begin{aligned} K(\mathbf{r},t) &= \nabla\cdot\left(\Psi_T(\mathbf{r})^{-1}\nabla\Psi_T(\mathbf{r})\right) f(\mathbf{r},t) \\ &\quad + \left(\frac{1}{2}-1\right)\Psi_T(\mathbf{r})^{-1}\nabla^2\Psi_T(\mathbf{r}) f(\mathbf{r},t) \\ &\quad + \Psi_T(\mathbf{r})^{-1}\nabla\Psi_T(\mathbf{r})\cdot\nabla f(\mathbf{r},t) \\ &\quad - \frac{1}{2}\nabla^2 f(\mathbf{r},t). \end{aligned}$$

Inserting the expression for the quantum force $\mathbf{F}(\mathbf{r}) = 2\Psi_T(\mathbf{r})^{-1}\nabla\Psi_T(\mathbf{r})$ and the local kinetic energy $K_L(\mathbf{r}) = -\frac{1}{2}\Psi_T(\mathbf{r})^{-1}\nabla^2\Psi_T(\mathbf{r})$ simplifies the expression dramatically

$$\begin{aligned} K(\mathbf{r},t) &= -\frac{1}{2}\nabla^2 f(\mathbf{r},t) + \frac{1}{2}\underbrace{\left[\mathbf{F}(\mathbf{r})\cdot\nabla f(\mathbf{r},t) + f(\mathbf{r},t)\nabla\cdot\mathbf{F}(\mathbf{r})\right]}_{\nabla\cdot[\mathbf{F}f(\mathbf{r},t)]} + K_L(\mathbf{r})f(\mathbf{r},t) \\ &= \frac{1}{2}\nabla\cdot\left[\left(\nabla-\mathbf{F}(\mathbf{r})\right)f(\mathbf{r},t)\right] + K_L(\mathbf{r})f(\mathbf{r},t). \end{aligned}$$

Inserting everything back into Eq. (3.25) yields

$$\begin{aligned} -\frac{\partial}{\partial t}f(\mathbf{r},t) &= -\frac{1}{2}\nabla\cdot\left[\left(\nabla-\mathbf{F}(\mathbf{r})\right)f(\mathbf{r},t)\right] + K_L(\mathbf{r})f(\mathbf{r},t) + \widehat{\mathbf{V}}f(\mathbf{r},t) - E_T f(\mathbf{r},t) \\ \frac{\partial}{\partial t}f(\mathbf{r},t) &= \frac{1}{2}\nabla\cdot\left[\left(\nabla-\mathbf{F}(\mathbf{r})\right)f(\mathbf{r},t)\right] - \left(E_L(\mathbf{r})-E_T\right)f(\mathbf{r},t), \end{aligned} \tag{3.28}$$

which is the Fokker-Planck diffusion equation from Eq. (3.20) with a constant shift representing the branching Green's function in the case Fokker-Planck diffusion.

Just as in traditional importance sampled Monte-Carlo integrals, optimized sampling is obtained in QMC by switching distributions into one which exploits known information about the problem at hand. In the case of standard Monte-Carlo integration, the sampling distribution is substituted with one which are similar to the original integrand, resulting in a smoother sampled function, whereas in QMC, a distribution is constructed with the sole purpose of imitating the exact ground state in order to suggest moves more efficiently. It is therefore reasonable to call the use of Fokker-Planck diffusion *importance sampled* QMC.

The energy estimated using the new distribution $f(\mathbf{r},t)$ will still equal the exact energy in the limit of convergence. This is demonstrated in the following equations:

$$\begin{aligned} E_{\text{QMC}} &= \frac{1}{N}\int f(\mathbf{r},\tau)\frac{1}{\Psi_T(\mathbf{r})}\widehat{\mathbf{H}}\Psi_T(\mathbf{r})\mathrm{d}\mathbf{r} \\ &= \frac{1}{N}\int \Phi(\mathbf{r},\tau)\widehat{\mathbf{H}}\Psi_T(\mathbf{r})\mathrm{d}\mathbf{r} \\ &= \frac{1}{N}\left\langle\Phi(\tau)\right|\widehat{\mathbf{H}}\left|\Psi_T\right\rangle, \end{aligned}$$

where

$$\begin{aligned} N &= \int f(\mathbf{r},\tau)\mathrm{d}\mathbf{r} \\ &= \int \Phi(\mathbf{r},\tau)\Psi_T(\mathbf{r})\mathrm{d}\mathbf{r} \\ &= \left\langle\Phi(\tau)|\Psi_T\right\rangle, \end{aligned}$$

which results in the following expression for the energy:

$$E_{\text{QMC}} = \frac{\left\langle\Phi(\tau)\right|\widehat{\mathbf{H}}\left|\Psi_T\right\rangle}{\left\langle\Phi(\tau)|\Psi_T\right\rangle}.$$

Assuming that the walkers have converged to the exact ground state, i.e. $|\Phi(\tau)\rangle = |\Phi_0\rangle$, letting the Hamiltonian work to the left yields

$$\begin{aligned} E_{\text{QMC}} &= E_0\frac{\langle\Phi_0|\Psi_T\rangle}{\langle\Phi_0|\Psi_T\rangle} \\ &= E_0. \end{aligned}$$

Estimating the energy in QMC will be discussed in detail in Sections 3.6.4 and 3.9.

3.3 Diffusive Equilibrium Constraints

Upon convergence of a Markov process, the ensemble of walkers will on average span the system's most likely state. This is exactly the behavior of a system of diffusing particles described by statistical mechanics: It will *thermalize*, that is, reach equilibrium.

Once thermalization is reached, expectation values may be sampled. However, simply spawning a Markov process and waiting for thermalization is an inefficient and unpractical scenario. This may take forever, or it may not; either way it is not optimal. Introducing rules of acceptance and rejection on top of the suggested transitions given by the Langevin equation in Eq. (3.19 or Eq. (3.24) will result in an optimized sampling. Special care must be taken not to violate necessary properties of the Markov process. If any of the conditions discussed in this section break, there is no guarantee that the system will thermalize properly.

3.3.1 Detailed Balance

For Markov processes, detailed balance is achieved by demanding a *reversible* Markov process. This boils down to a statistical requirement stating that

$$P_i W(i \to j) = P_j W(j \to i), \tag{3.29}$$

where P_i is the probability density in configuration i, and $W(i \to j)$ is the transition probability between states i and j.

3.3.2 Ergodicity

Another requirement is that the sampling must be *ergodic* [25], that is, the random walkers need to be able to reach any configuration in the space spanned by the distribution function. It is tempting to define a brute force acceptance rule where only steps resulting in a higher overall probability is accepted, however, this limits the path of the walker, and will thus break the requirement of ergodicity.

3.4 The Metropolis Algorithm

The Metropolis Algorithm is a simple set of acceptance/rejection rules used in order to make the thermalization more efficient. For a given probability distribution function P, the Metropolis algorithm will force sampled points to follow this distribution.

Starting from the criteria of detailed balance given in Eq. (3.29, and further introducing a model for the transition probability $W(i \to j)$ as consisting of two parts: The probability of selecting configuration j given configuration i, $g(i \to j)$, times a probability of accepting the selected move, $A(i \to j)$, yields

$$\begin{aligned} P_i W(i \to j) &= P_j W(j \to i), \\ P_i g(i \to j) A(i \to j) &= P_j g(j \to i) A(j \to i). \end{aligned} \tag{3.30}$$

Inserting the probability distribution as the wave function squared and the selection probability as the Green's function, the expression becomes

$$\begin{aligned} |\psi_i|^2 G(i \to j) A(i \to j) &= |\psi_j|^2 G(j \to i) A(j \to i), \\ \frac{A(j \to i)}{A(i \to j)} &= \frac{G(i \to j)}{G(j \to i)} \frac{|\psi_i|^2}{|\psi_j|^2} \equiv R_G(j \to i) R_\psi(j \to i)^2, \end{aligned} \tag{3.31}$$

where the defined ratios correspond to the Green's function - and wave function ratio, respectively.

Assume now that configuration i has a higher overall probability than configuration j. The essence of the Metropolis algorithm is that the step is automatically accepted, that is, $A(i \to j) = 1$. In other words, a more efficient thermalization is obtained by accepting all these moves. What saves Metropolis from breaking the criteria of ergodicity, is the fact that suggested moves to lower probability states are not automatically rejected. This is demonstrated by solving Eq. (3.31) for the case where $A(i \to j) = 1$, that is, the case where $P_i < P_j$. This yields

$$A(j \to i) = R_G(j \to i) R_\psi(j \to i)^2.$$

Combining both scenarios into one expression yield the following acceptance/rejection rules:

$$A(i \to j) = \begin{cases} R_G(i \to j) R_\psi(i \to j)^2 & R_G(i \to j) R_\psi(i \to j)^2 < 1 \\ 1 & \text{else} \end{cases} . \tag{3.32}$$

This equation can be simplified to

$$A(i \to j) = \min\{R_G(i \to j) R_\psi(i \to j)^2, 1\}. \tag{3.33}$$

In the isotropic diffusion case, the Green's function ratio cancels due to symmetry, i.e. $R_G(i \to j) = 1$, resulting in the standard Metropolis algorithm:

$$A(i \to j) = \min\{R_\psi(i \to j)^2, 1\}. \tag{3.34}$$

On the other hand, for Fokker-Planck diffusion, there will be no cancellation of the Green's functions. Inserting Eq. (3.22) into Eq. (3.32) results in the *Metropolis Hastings algorithm* [25]. The ratio of the Green's functions can be evaluated efficiently by simply subtracting the exponents of the exponentials. This is best demonstrated by calculating the logarithm

$$\begin{aligned} \log R_G^{\mathrm{FP}}(i \to j) &= \log\left(G_{\mathrm{Diff}}^{\mathrm{FP}}(j \to i)/G_{\mathrm{Diff}}^{\mathrm{FP}}(i \to j)\right) \\ &= \frac{1}{2}\big(F(x_j) + F(x_i)\big)\big(\frac{1}{2} D\delta t(F(x_j) - F(x_i)) + x_i - x_i\big), \end{aligned} \tag{3.35}$$

$$A(i \to j) = \min\{\exp\left(\log R_G^{\mathrm{FP}}(i \to j)\right) R_\psi(i \to j)^2, 1\}. \tag{3.36}$$

Derived from detailed balance, the Metroplis Algorithm is an essential part of any Markov Chain Monte-Carlo algorithm. Besides QMC, methods for solving problems such as the *Ising Model* greatly benefit from these rules [26].

In practice, without the Metropolis sampling, the ensemble of walkers will not span that of the trial wave function. This is due to the fact that the time-step used in the simulations is finite, and the trial positions of the walkers are random. A chart flow describing the implementation of the Metropolis algorithm and the diffusion process is given in Figure 3.1.

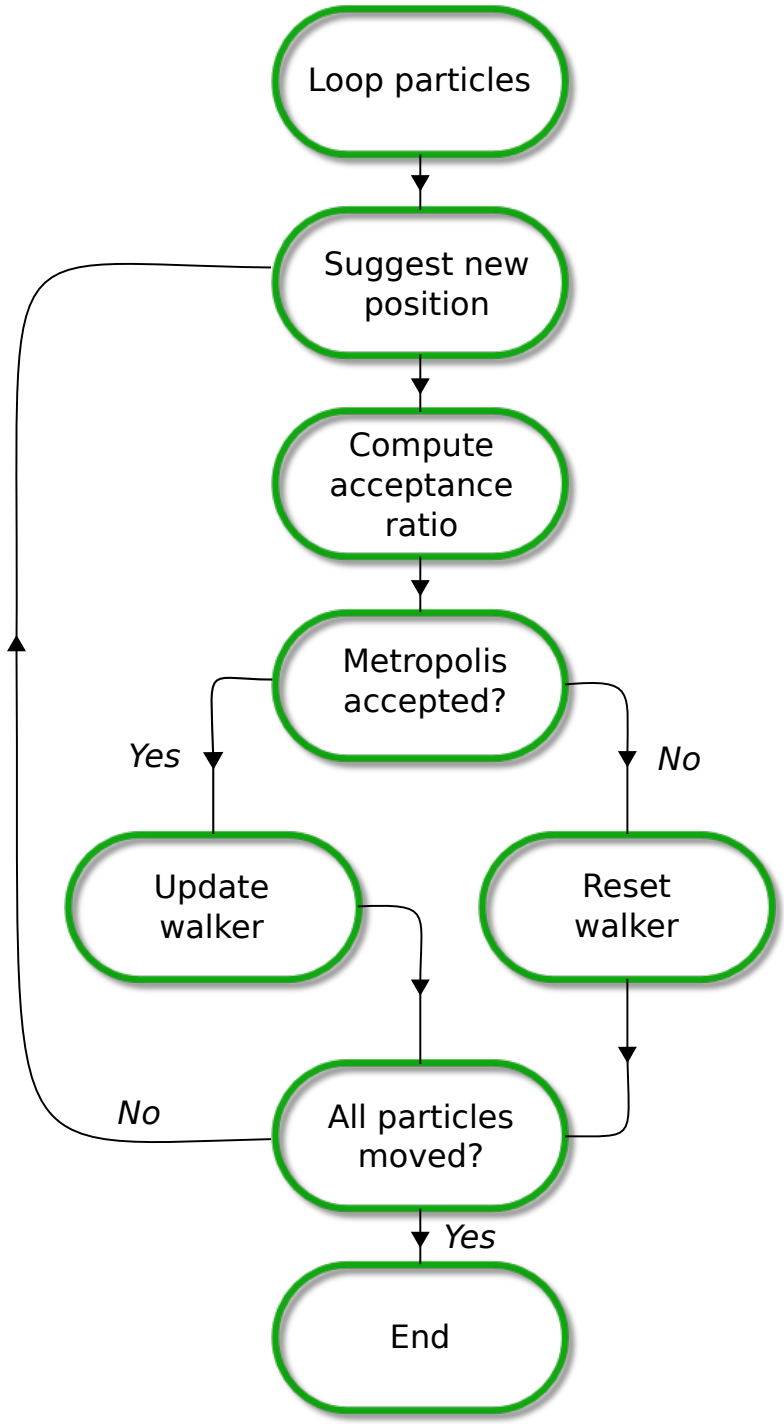


Figure 3.1: Flow chart describing the process of iterating a walker through a single time-step, that is, simulation the application of the Green's function from Eq. (3.15) using the Metropolis algorithm. New positions are suggested according to the chosen diffusion model.

3.5 The Process of Branching

In the previous section it became clear that the Metropolis test will guide the walkers to span a distribution representing the trial wave function. This implies that without further action, no changes to the distribution can be made, and the point of modelling the projection operator from Eq. (3.5) is rendered useless. The important fact to include is that the branching Green's function from Eq. (3.38) and Eq. (3.37) distribute weights to the walkers, effectively altering the spanned distribution.

The process of branching in QMC is simulated by the creation and destruction of walkers with probability equal to that of the branching Green's function [21]. The explicit shapes in case of isotropic - (ISO) and anisotropic diffusion (FP) are

$$G_B^{\mathrm{ISO}}(i \to j) = e^{-\left(\frac{1}{2}[V(x_i)+V(x_j)]-E_T\right)\delta\tau}, \tag{3.37}$$

$$G_B^{\mathrm{FP}}(i \to j) = e^{-\left(\frac{1}{2}[E_L(x_i)+E_L(x_j)]-E_T\right)\delta\tau}, \tag{3.38}$$

where $E_L(x_i)$ is the energy evaluated in configuration x_i (see Section 3.6.4 for details). The three different scenarios which arise is

- $G_B = 1$: No branching.
- $G_B = 0$: The current walker is to be removed from the current ensemble.
- $G_B > 1$: On average $G_B - 1$ replicas of the current walker are made.

Defining the following quantity allows for an efficient simulation of this behavior

$$\overline{G}_B = \mathrm{floor}\left(G_B + a\right), \tag{3.39}$$

where a is a uniformly distributed number on $[0, 1)$. The probability that $\overline{G}_B = G_B + 1$ is then equal to $G_B - \mathrm{floor}(G_B)$. As an example, assume $G_B = 3.3$. The value of $\overline{G}_B$ is then either three or four, depending on whether $a < 0.7$ or not. The probability that $a < 0.7$ is obviously 70%, implying that there is a 30% chance that $\overline{G}_B$ is equal to four, and 70% chance that is is equal to three.

There are some programming challenges due to the fact that the number of walkers is not conserved, such as cleaning up inactive walkers and stabilizing the population across different computational nodes. For details regarding this, see the code documentation at Ref. [9]. Isotropic diffusion is in practice never used with branching due to the singularities in the Coulomb interaction (see Eq. (3.37)). This singularity may cause large fluctuations in the walker population, which is far from an optimal behavior.

The process of branching is demonstrated in Figure 3.2.

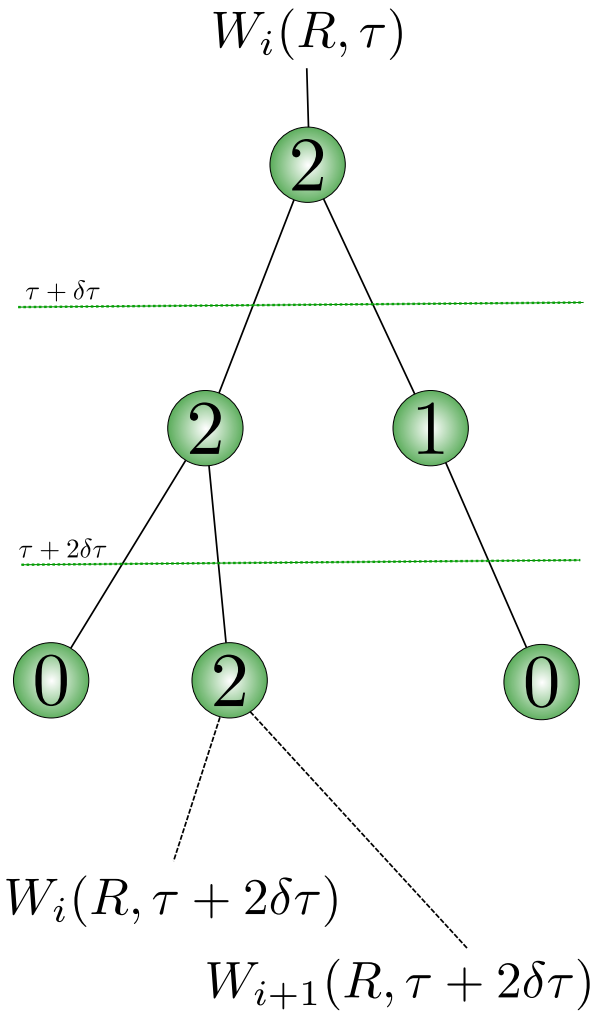


Figure 3.2: The process of branching illustrated. The initial walker $W_i(R,\tau)$ is branched according to the rules of Section 3.5. The numerical value inside the nodes represents the value of $\overline{G}_B$ from Eq. (3.39). Each horizontal dashed line represent a diffusion step, i.e. a transition in time. Two lines exiting the same node represent identical walkers. After moving through the diffusion process, no two walkers should ever be equal, given that not all of the steps was rejected by the Metropolis test.

3.6 The Trial Wave Function

The initial condition of the QMC calculations, that is, the trial wave function $\Psi_T(\mathbf{r})$, can in principle be chosen to be any normalizable wave function whose overlap with the exact ground state wave function, $\Psi_0(\mathbf{r})$, is non-zero.

If the overlap is zero, that is, if $C_0 = 0$ in Eq. (3.7), the formalism breaks down, and no final state of convergence can be reached. On the other hand, the opposite scenario implies the opposite behavior; the closer C_0 is to unity, the more rapidly the exact ground state, $\Psi_0(\mathbf{r})$, will become the dominant contribution to the distribution.

In other words, the trial wave function should be chosen in such a way that the overlap is optimized, i.e. close to unity. Since the exact ground state is unknown, this overlap has to be optimized based on educated guesses and by forcing known properties of the exact ground state into the approximation. This will be the focus in this section.

Before getting into specifics, a few notes on many-body theory is needed. From this point on, all particles are assumed to be identical. For more information regarding Quantum Mechanical concepts and many-body theory, see for example Refs. [17, 18, 27].

3.6.1 Many-body Wave Functions

Many-body theory arise from the existence of *many-body interactions*, which in this thesis will be truncated at the level of the Coulomb interaction, that is, the two-body interaction. Nature operates using N-body interactions, however, it is overall safe to assume that the contributions beyond Coulomb decrease as the order of the interactions increase. If only one-body interactions were present, as is the case for non-interacting particles, the full system would decouple into N single-particle systems, rendering many-body theory redundant.

Finding the ground state is, not surprisingly, equivalent to solving the time-independent Schrödinger Equation from Eq. (3.2) for the lowest energy eigenvalue, that is

$$\widehat{\mathbf{H}}\Psi_0(\mathbf{r}) = E_0\Psi_0(\mathbf{r}), \tag{3.40}$$

where $\mathbf{r} \equiv \{\mathbf{r}_1, \mathbf{r}_2, ..., \mathbf{r}_N\}$ represents the position of every particle. Exact solutions to realistic many-body systems rarely exist, however, like in Section 3.1.1, expanding the solution in a known basis $\Phi_k(\mathbf{r})$ is always legal, which reduces the problem into that of a *coefficient hunt*

$$\Psi_0(\mathbf{r}) = \sum_{k=0}^{\infty} C'_k\Phi_k(\mathbf{r}), \tag{3.41}$$

where the primed coefficients should not to be confused with the previous coefficients expanding an arbitrary state in the $\Psi_i(\mathbf{r})$ basis (see Eq. (3.6)). Different many-body methods give rise to different ways of estimating these coefficients, however, certain concepts are necessarily common, for instance truncating the basis at some level, K:

$$\Psi_0(\mathbf{r}) = \sum_{k=0}^{K} \tilde{C}'_k\Phi_k(\mathbf{r}), \tag{3.42}$$

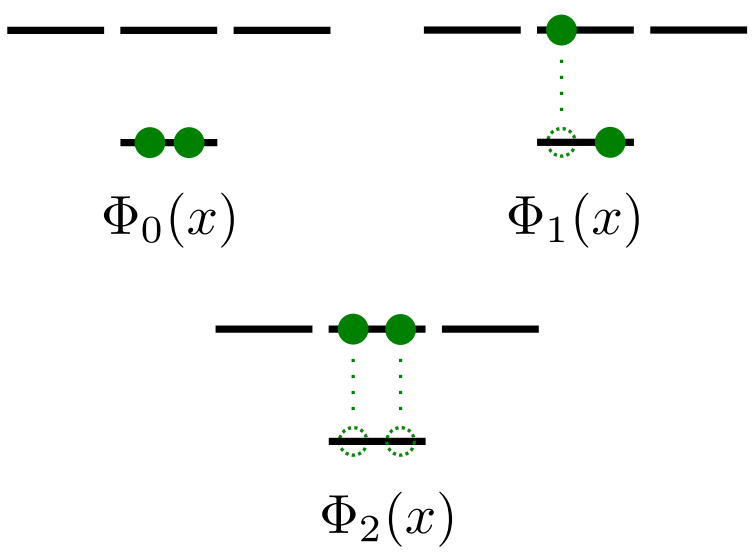


Figure 3.3: Three different electron configurations in an shell structure making up three different $\Phi_k(\mathbf{r})$, i.e. constituents of the many-body basis described in Eq. (3.42). An electron (solid dot) is represented by e.g. the orbital $\phi_{1s}(\mathbf{r}_1)$.

where $\tilde{C}'_k \neq C'_k$ unless K tends to infinity, however, it is overall safe to assume that the value of the coefficients decrease as k increase.

The many-body basis elements $\Phi_k(\mathbf{r})$ are constructed using N elements from a basis of single-particle wave functions, or *orbitals* for short, where N denotes the total number of particles in the system. In other words, these orbitals, labeled $\phi_n(\mathbf{r}_i)$, combined in different ways give rise to the different many-body wave functions making up the total wave function. The process of calculating basis elements often boils down to an exercise in combinatorics involving combinations of orbitals.

To bring some clarity to the relation between the different wave functions, consider the following example: Imagine electrons surrounding a nucleus, i.e an atom. A single electron occupying a state with quantum number n at a position $\mathbf{r}_i$ is then described by the orbital $\phi_n(\mathbf{r}_i)$. Each unique[3] configuration of electrons (in terms of n) will give rise to one unique $\Phi_k(\mathbf{r})$. In other words, the complete basis of $\Phi_k(\mathbf{r})$ is described by the collection of all possible excited states and the ground state. $\Phi_0(\mathbf{r})$ is the ground state of the atom, $\Phi_1(\mathbf{r})$ has one electron exited to a higher shell, $\Phi_2(x)$ has another, and so on. See Figure 3.3 for a demonstration of this. The ordering of the terms in Eq. (3.42) are thus chosen to represent higher and higher excitations, i.e. the states has higher and higher energy eigenvalues.

To summarize, constructing an approximation to an unknown many-body ground state wave function involves three steps:

[3]Two wave functions are considered equal if they differ by nothing but a phase factor.

Step one	Choose a basis of orbitals $\phi_n(\mathbf{r}_i)$, e.g. hydrogen states.
Step two	Construct $\Phi_k(\mathbf{r})$ from $N\times$ $\phi_n(\mathbf{r}_i)$.
Step three	Construct $\Psi_0(\mathbf{r})$ from $K\times$ $\phi_k(\mathbf{r})$.

The last step is well described by Eq. (3.42), but is seldom necessary to perform explicitly; expressions involving the approximated ground state wave function is given in terms of the constituent $\Phi_k(\mathbf{r})$ elements and their coefficients.

Step one in detail

The Hamiltonian of an N-particle system is

$$\widehat{\mathbf{H}} = \widehat{\mathbf{H}}_0 + \widehat{\mathbf{H}}_{\mathrm{I}}, \tag{3.43}$$

where $\widehat{\mathbf{H}}_0$ and $\widehat{\mathbf{H}}_{\mathrm{I}}$ are the one-body - and the many-body Hamiltonian, respectively. As mentioned in the introduction, the many-body interactions are truncated at the level of the two-body Coulomb interaction. The one-body term consist of the external potential $\widehat{\mathbf{u}}_{\mathrm{ext}}(\mathbf{r}_i)$ and the kinetic term $\widehat{\mathbf{t}}(\mathbf{r}_i)$ for all particles. In order words, the two Hamiltonians are written

$$\begin{aligned} \widehat{\mathbf{H}}_0 &= \sum_{i=1}^{N} \widehat{\mathbf{h}}_0(\mathbf{r}_i) \\ &= \sum_{i=1}^{N} \widehat{\mathbf{t}}(\mathbf{r}_i) + \widehat{\mathbf{u}}_{\mathrm{ext}}(\mathbf{r}_i), \end{aligned} \tag{3.44}$$

and

$$\begin{aligned} \widehat{\mathbf{H}}_{\mathrm{I}} &\simeq \sum_{i<j=1}^{N} \widehat{\mathbf{v}}(r_{ij}) \\ &= \sum_{i<j=1}^{N} \frac{1}{r_{ij}}, \end{aligned} \tag{3.45}$$

where $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ is the distance between two particles.

In order to optimize the overlap C_0 with the exact wave function, the single-particle orbitals are commonly chosen to be the eigenfunctions of the non-interacting single-particle Hamiltonian, that is

$$\widehat{\mathbf{h}}_0(\mathbf{r}_i)\phi_n(\mathbf{r}_i) = \epsilon_n\phi_n(\mathbf{r}_i). \tag{3.46}$$

If no such choice can be made, choosing, free-particle solutions, Laguerre polynomials, or similar, is the general strategy. However, for these bases, the expansion truncation K from Eq. (3.42) needs to be higher in order to achieve a satisfying overlap.

Step two in detail

In the case of *fermions*, that is, half-integer spin particles like electrons, protons, etc., $\Phi_k(\mathbf{r})$ is an anti-symmetric function[4] on the form of a determinant: The so-called *Slater determinant*. The shape of the

[4]Interchanging two particles in an anti-symmetric wave function will reproduce the state changing only the sign.

determinant is given in Eq. (3.47). The anti-symmetry is a direct consequence of the *Pauli Exclusion Principle*: At any given time, two fermions cannot occupy the same state.

Bosons, on the other hand, have symmetric wave functions, which in many ways are easier to deal with because of the lack of an exclusion principle. The bosonic many-body wave function is given in Eq. (3.48). In order to keep the terminology less abstract and confusing, the focus will be on systems of fermions from here on.

$$\begin{aligned}\Phi_0^{\mathrm{AS}}(\mathbf{r}_1,\mathbf{r}_2,...,\mathbf{r}_N) &\propto \sum_{\mathrm{P}}(-)^{\mathrm{P}}\widehat{\mathbf{P}}\phi_1(\mathbf{r}_1)\phi_2(\mathbf{r}_2)...\phi_N(\mathbf{r}_N)\\ &= \begin{vmatrix}\phi_1(\mathbf{r}_1) & \phi_2(\mathbf{r}_1) & \cdots & \phi_N(\mathbf{r}_1)\\ \phi_1(\mathbf{r}_2) & \phi_2(\mathbf{r}_2) & \cdots & \phi_N(\mathbf{r}_2)\\ \vdots & \vdots & \ddots & \vdots\\ \phi_1(\mathbf{r}_N) & \phi_2(\mathbf{r}_N) & \cdots & \phi_N(\mathbf{r}_N)\end{vmatrix}, \qquad (3.47)\end{aligned}$$

$$\Phi_0^{\mathrm{S}}(\mathbf{r}_1,\mathbf{r}_2,...,\mathbf{r}_N) \propto \sum_{\mathrm{P}}\widehat{\mathbf{P}}\phi_1(\mathbf{r}_1)\phi_2(\mathbf{r}_2)...\phi_N(\mathbf{r}_N). \qquad (3.48)$$

The permutation operator $\widehat{\mathbf{P}}$ is simply a way of writing *in any combination of particles and states*, hence the combinatoric exercise mentioned previously. Any combination of N orbital elements $\phi_n(\mathbf{r}_i)$ can be used to produce different $\Phi_k(\mathbf{r})$. For illustrative purposes, and for the purpose of this thesis in general where a single determinant ansatz is used, only the ground state has been presented.

Dealing with correlations

The contributions to the ground state on the right-hand side in Eq. (3.41) for $k > 0$ are referred to as *correlation* terms. Given that the single-particle wave functions are chosen by Eq. (3.46), the existence of the correlation terms, i.e. $C_k' \neq 0$ for $k > 0$, follows as a direct consequence of the electron-electron interaction, hence the name.

As an example, imagine performing an energy calculation with two particles being infinitely close; the Coulomb singularity will cause the energy to blow up. However, if the calculations are performed using the exact wave function, the diverging terms will cancel out; the energy eigenvalue is independent of the position of the state.

In other words, a necessary property of the exact wave function is that the singularities in the many-body interactions are canceled. The basic idea is thus to make sure the trial wave function also has this property. By doing so, it is brought closer to the exact wave function.

These criteria are called *cusp conditions* [26], and serve as powerful guides when it comes to selecting an optimal trial wave function.

3.6.2 Choosing the Trial Wave Function

To recap, choosing the trial wave function boils down to optimizing the overlap $C_0 = \langle\Psi_0|\Psi_T\rangle$ using a priori knowledge about the system at hand. As discussed previously, the optimal choice of single-particle basis is the eigenfunctions of the non-interacting case (given that they exist). Starting from Eq. (3.42), from here on referred to as the *spatial wave function*, the first step is to make sure the cusp conditions are obeyed.

Introducing the correlation functions $f(r_{ij})$, where r_{ij} is the relative distance between particle i and j, the general ansatz for the trial wave function becomes

$$\Psi_T(\mathbf{r}_1, ..., \mathbf{r}_N) = \Big[\sum_{k=0}^{K} C_k \Phi_k(\mathbf{r}_1, ..., \mathbf{r}_N)\Big] \prod_{i<j}^{N} f(r_{ij}). \tag{3.49}$$

The idea is now to choose $f(r_{ij})$ in such a way that the cusp conditions are obeyed. This should, in light of previous discussions, reduce the amount of terms needed in the spatial wave function to obtain a satisfying overlap.

Explicit shapes

Several models for the correlation function exist, however, some are less practical than others. An example given in ref. [21] demonstrates this nicely: Hylleraas presented the following correlation function

$$f(r_{ij})_{\text{Hylleraas}} = e^{-\frac{1}{2}(r_i + r_j)} \sum_k d_k (r_{ij})^{a_k} (r_i + r_j)^{b_k} (r_i - r_j)^{e_k}, \tag{3.50}$$

where all k-subscripted parameters are free. Calculating the helium ground state energy using this correlation function with nine terms yields a four decimal precision. Eight digit precision is achieved by including 1078 terms. For the purpose of QMC, including such vast amounts of parameters is out of the question. The strength of QMC is that very good results can be obtained using a very simple ansatz to the wave function.

A commonly used correlation function in studies involving few variational parameters is the *Padé Jastrow* function

$$\begin{aligned} \prod_{i<j}^{N} f(r_{ij}) &= \exp(U), \\ U &= \sum_{i<j}^{N} \left(\frac{\sum_k a_k r_{ij}^k}{1 + \sum_k \beta_k r_{ij}^k} \right) + \sum_{i}^{N} \left(\frac{\sum_k a'_k r_i^k}{1 + \sum_k \alpha_k r_i^k} \right). \end{aligned}$$

For systems where the correlations are relatively well behaved, it is custom to drop the second sum all together, and keep only the $k = 1$ term from the first. The resulting function reads

$$f(r_{ij}; \beta) = \exp\left(\frac{a_{ij} r_{ij}}{1 + \beta r_{ij}} \right), \tag{3.51}$$

where β is a variational parameter, and $a_{k=1} \equiv a_{ij}$ is a constant depending on the relative spin-orientation of particles i and j tuned in such a way that the cusp conditions are obeyed. For three dimensions, $a_{ij} = 1/4$ or $a_{ij} = 1/2$ depending on whether or not the spins of i and j are parallell or anti parallell, respectively [21]. For two dimensions, the values are $a_{ij} = 1/3$ (parallell) or $a_{ij} = 1$ (anti-parallel) [28]. This is the correlation function used for all systems in this thesis.

Shifting the focus back to the spatial wave function, in the case of a fermionic system, the evaluation of an $N \times N$ Slater determinant severely limits the efficiency of many-particle simulations. However, assuming the Hamiltonian to be spin-independent, the eigenstates for different spin eigenvalues will be identical.

This fact results in the spatial wave function being split in two: One part for each spin eigenvalue. A detailed derivation of this is given in the appendix of Ref. [29]. The resulting wave function thus becomes

$$\Psi_T(\mathbf{r};\beta) = \Big[\sum_{k=0}^{K} C'_k \tilde{\Phi}_k(\mathbf{r}_1,...,\mathbf{r}_{\frac{N}{2}})\tilde{\Phi}_k(\mathbf{r}_{\frac{N}{2}-1},...,\mathbf{r}_N)\Big]\prod_{i<j}^{N} f(r_{ij};\beta). \tag{3.52}$$

Due to the identical nature of the particles, they may be arbitrarily ordered. For simplicity, the first half represents spin up, and the second half spin down. The spin up determinant will from here on be labeled $|\mathbf{S}^{\uparrow}|$, and the spin down one $|\mathbf{S}^{\downarrow}|$, where the $\mathbf{S}$ matrix will be referred to as the *Slater matrix*. Stitching everything together, the explicit shape of the trial wave function becomes

$$\Psi_T(\mathbf{r}_1,...,\mathbf{r}_N;\beta) = \sum_{k=0}^{K} C'_k |\mathbf{S}^{\uparrow}|_k |\mathbf{S}^{\downarrow}|_k \prod_{i<j}^{N} f(r_{ij};\beta). \tag{3.53}$$

This shape is referred to as a *multi-determinant* trial wave function unless $K = 1$, in which it will be referred to as a *single-determinant* trial wave function.

Limitations

Depending on the complexity of the system at hand, more complicated trial wave functions might be needed to obtain reasonable convergence. However, it is important to distinguish between simply integrating a trial wave function, and performing the full diffusion calculation. As a reminder: Simple integration will not be able to alter the distribution; what you have is what you get. Solving the diffusion problem, on the other hand, will alter the distribution from that of the trial wave function ($\tau = 0$) into a distribution closer to the exact wave function by Eq. (3.6).

Because of this fact, limitations due to the trial wave function in full[5] QMC is far less than what is the case of standard Monte-Carlo integration. A more complex trial wave function might convergence faster, but at the expense of being more CPU-intensive. This implies that CPU time per walker can be traded for convergence time. For systems of many particles, the CPU time per walker needs to be as low as possible in order to get the computation done in a reasonable amount of time. In other words, the choice of trial wave function needs to be done in light of the system at hand, and the specific aim of the computation.

On the other hand, when the number of particles in the system increase, it is safe to assume that the quality of the trial wave function will decrease. This is demonstrated in Ref. [30], where calculations for F_2 (18 particles) need an increase in the number of determinants to achieve a result with the same precision as calculations for O_2 (16 particles).

Single-determinant trial wave functions

In the case of well-behaving systems, a single determinant with a simple Jastrow factor serves as a reasonable trial wave function. This simplicity opens up the possibility of simulating large systems efficiently. More details regarding this will be given in Section 4.3.

In order to further optimize the overlap with the exact wave function, a second variational parameter α is introduced in the spatial wave function

[5] "Full" in the sense that all Green's functions are applied. As will be revealed later, VMC corresponds to a standard importance sampled Monte-Carlo integration by omitting the branching process.

$$\Psi_T(\mathbf{r}_1, ..., \mathbf{r}_N; \alpha, \beta) = D^{\uparrow}(\alpha) D^{\downarrow}(\alpha) \prod_{i<j}^{N} f(r_{ij}; \beta). \tag{3.54}$$

Determining the optimal values of the variational parameters will be the topic of the next section. If the introduction of the variational parameter was redundant, optimizations would simply yield $\alpha = 1$.

3.6.3 Selecting Optimal Variational Parameters

All practical ways of determining the optimal values of the variational parameters originate from the same powerful principle: *The Variational Principle*. The easiest way of demonstrating the principle is to evaluate the expectation value of the energy, using an approach similar to what used in Eq. (3.6). Consider the following relations

$$\begin{aligned} E_k &= \langle \Psi_k | \, \widehat{\mathbf{H}} \, | \Psi_k \rangle \, , \\ E &= \langle \Psi_T(\alpha, \beta) | \, \widehat{\mathbf{H}} \, | \Psi_T(\alpha, \beta) \rangle \\ &= \sum_{kl} C_k^* C_l \underbrace{\langle \Psi_k | \, \widehat{\mathbf{H}} \, | \Psi_l \rangle}_{E_k \delta_{kl}} \\ &= \sum_k |C_k|^2 E_k. \end{aligned}$$

Just as with the projection operator, introducing $E_k = E_0 + \delta E_k$ where $\delta E_k \geq 0$ will simplify the arguments

$$\begin{aligned} E &= \sum_k |C_k^2| (E_0 + \delta E_k) \\ &= E_0 \underbrace{\sum_k |C_k^2|}_{1} + \underbrace{\sum_k |C_k|^2 \delta E_k}_{\geq 0} \\ &\geq E_0. \end{aligned}$$

The conclusion is remarkable: No matter which trial wave function is used, the resulting energy will always be greater or equal to the exact ground state energy. This implies that the problem of choosing variational parameters comes down to a minimization problem in the parameters space

$$\frac{\partial \langle E \rangle}{\partial \alpha_i} = \frac{\partial}{\partial \alpha_i} \langle \Psi_T(\alpha_i) | \, \widehat{\mathbf{H}} \, | \Psi_T(\alpha_i) \rangle = 0 \tag{3.55}$$

In order to work with Eq. (3.55) in practice, it needs to be rewritten in terms of known values. Since the wave function depends on the variational parameter, the normalization factor needs to be included in the expression of the expectation value. Applying the product rule numerous times yields

$$\begin{aligned}
\frac{\partial\langle E\rangle}{\partial\alpha_i} &= \frac{\partial}{\partial\alpha_i}\frac{\langle\Psi_T(\alpha_i)|\,\widehat{\mathbf{H}}\,|\Psi_T(\alpha_i)\rangle}{\langle\Psi_T(\alpha_i)|\Psi_T(\alpha_i)\rangle} \\
&= \frac{\left(\langle\Psi_T(\alpha_i)|\,\frac{\partial}{\partial\alpha_i}\widehat{\mathbf{H}}\,|\Psi_T(\alpha_i)\rangle + \langle\Psi_T(\alpha_i)|\,\widehat{\mathbf{H}}\frac{\partial}{\partial\alpha_i}\,|\Psi_T(\alpha_i)\rangle\right)}{\langle\Psi_T(\alpha_i)|\Psi_T(\alpha_i)\rangle^2}\langle\Psi_T(\alpha_i)|\Psi_T(\alpha_i)\rangle \\
&- \langle\Psi_T(\alpha_i)|\,\widehat{\mathbf{H}}\,|\Psi_T(\alpha_i)\rangle\frac{\left(\langle\Psi_T(\alpha_i)|\,\frac{\partial}{\partial\alpha_i}\right)|\Psi_T(\alpha_i)\rangle + \langle\Psi_T(\alpha_i)|\left(\frac{\partial}{\partial\alpha_i}\,|\Psi_T(\alpha_i)\rangle\right)}{\langle\Psi_T(\alpha_i)|\Psi_T(\alpha_i)\rangle^2}.
\end{aligned}$$

The Hamiltonian does not depend on the variational parameters, hence both terms in the first expansion is equal. Cleaning up the expression yields

$$\begin{aligned}
\frac{\partial\langle E\rangle}{\partial\alpha_i} &= 2\left(\frac{\langle\Psi_T(\alpha_i)|\,\widehat{\mathbf{H}}\frac{\partial}{\partial\alpha_i}\,|\Psi_T(\alpha_i)\rangle}{\langle\Psi_T(\alpha_i)|\Psi_T(\alpha_i)\rangle} - \langle E\rangle\frac{\langle\Psi_T(\alpha_i)|\,\frac{\partial}{\partial\alpha_i}\,|\Psi_T(\alpha_i)\rangle}{\langle\Psi_T(\alpha_i)|\Psi_T(\alpha_i)\rangle}\right) \\
&= 2\left(\left\langle E\frac{\partial\Psi_T}{\partial\alpha_i}\right\rangle - \langle E\rangle\left\langle\frac{\partial\Psi_T}{\partial\alpha_i}\right\rangle\right). \qquad (3.56)
\end{aligned}$$

In the case of $\Psi_T(\mathbf{r};\alpha_i)$ being represented by a Slater determinant, the relationship between the variational derivative of the determinant and the variational derivative of the single-particle orbitals $\phi_n(\mathbf{r}_i;\alpha_i)$ is

$$\frac{\partial\Psi_T(\mathbf{r};\alpha_i)}{\partial\alpha_i} = \sum_{p=1}^{N}\sum_{q=0}^{N/2}\phi_q(\mathbf{r}_p;\alpha_i)\left[\frac{\partial\phi_q(\mathbf{r}_p;\alpha_i)}{\partial\alpha_i}\right]\mathbf{S}^{-1}_{qp}, \qquad (3.57)$$

where $\mathbf{S}^{-1}_{qi}$ is the inverse of the Slater matrix, which will be discussed in more detail in Section 4.4.1.

Using these expressions for the *variational energy gradient*, the derivatives can be calculated exactly the same way the energy. The gradient can then be used to move in the direction of the variational minimum in Eq. (3.55).

This strategy gives rise to numerous ways of finding the optimal parameters, such as using the well known Newton's method, conjugate gradient methods [31], steepest descent (similar to Newton's method), and many more. The method implemented for this thesis is called *Adaptive Stochastic Gradient Descent*, and is an efficient iterative algorithm for seeking the variational minimum. The gradient descent methods will be covered in Section 3.7.

3.6.4 Calculating Expectation Values

The expectation value of an operator $\widehat{\mathbf{O}}$ is obtained by sampling *local* values, $O_L(x)$

$$\begin{aligned}
\langle \Psi_T | \widehat{\mathbf{O}} | \Psi_T \rangle &= \int \Psi_T(x)^* \widehat{\mathbf{O}} \Psi_T(x) \mathrm{d}x \\
&= \int |\Psi_T|^2 \left(\frac{1}{\Psi_T(x)} \widehat{\mathbf{O}} \Psi_T(x) \right) \mathrm{d}x \\
&= \int |\Psi_T|^2 O_L(x) \mathrm{d}x. \qquad (3.58) \\
O_L(x) &= \frac{1}{\Psi_T(x)} \widehat{\mathbf{O}} \Psi_T(x). \qquad (3.59)
\end{aligned}$$

Discretizing the integral yields

$$\langle \Psi_T | \widehat{\mathbf{O}} | \Psi_T \rangle \equiv \langle O \rangle \simeq \frac{1}{n} \sum_{i=1}^{n} O_L(x_i) \equiv \overline{O}, \qquad (3.60)$$

where x_i is a random variable taken from distribution of the trial wave function. The *ensemble average*, $\langle O \rangle$ will, given ergodicity, equal the estimated average $\overline{O}$ in the limit $n \to \infty$, that is

$$\langle O \rangle = \lim_{n \to \infty} \overline{O} = \lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} O_L(x_i). \qquad (3.61)$$

In the case of the energy estimation, this implies that once the walkers reach equilibrium, local values can be sampled based on their configurations $\mathbf{r}_i$ (remember that Metropolis ensures that the walkers follow $|\Psi_T(\mathbf{r})|^2$). In the case of energies, the explicit expression becomes

$$\langle E \rangle \simeq \frac{1}{n} \sum_{i=1}^{n} \left(\frac{1}{\Psi_T(\mathbf{r}_i)} \left(-\frac{1}{2} \nabla^2 \right) \Psi_T(\mathbf{r}_i) + V(\mathbf{r}_i) \right). \qquad (3.62)$$

Incorporating the branching Green's function G_B into the above equation is covered in the DMC section.

3.6.5 Normalization

Every explicit calculation using the trial wave function in QMC involves taking ratios. Calculating ratios implies a cancellation in the normalization factors. Eq. (3.32) from the Metropolis section, the quantum force in the Fokker-Planck equation, and the sampling of local values described in the previous section demonstrate exactly this; everything involves ratios.

Not having to normalize the wave functions does not only save a lot of CPU time, but it also removes the need of including the normalization factors of the single-particle wave functions; any constants multiplying $\phi_n(x_i)$ in Eq. (3.47) and Eq. (3.48) can be taken outside the sum over permutations, and will thus cancel when the ratio between two wave functions constituting of the same single-particle orbitals are computed.

Note, however, that this argument is valid for single determinant wave functions only.

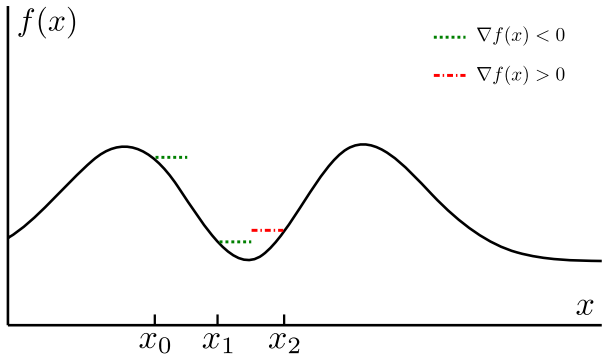


Figure 3.4: Two steps of a one dimensional Gradient Descent process. Steps are taken in the direction of the negative gradient (indicated by dotted lines).

3.7 Gradient Descent Methods

The direction of a gradient serves as a guide to extremal values. Gradient descent, also called steepest descent[6], is a family of minimization methods using this property of gradients in order to backtrace a local minimum in the vicinity of an initial guess.

3.7.1 General Gradient Descent

Seeking maxima or minima is simply a question of whether the positive or the negative direction of the gradient is followed. Imagine a function $f(x)$, with a minimum residing at $x = x_m$. The information at hand is then

$$\nabla f(x_m) = 0 \tag{3.63}$$
$$\nabla f(x_m - \mathrm{d}x) < 0 \tag{3.64}$$
$$\nabla f(x_m + \mathrm{d}x) > 0 \tag{3.65}$$

where $\mathrm{d}x$ is a infinite decimal displacement.

As an example, imagine starting from an initial guess x_0. The direction of the gradient is then calculated and followed a number of steps. From Figure 3.4 and the previous equations, it is clear that crossing the true minimum induces a sign change in the gradient. The brute force way of minimizing is to simply end the calculation at this point, however, this would require an extreme amount of very small steps in order to achieve good precision.

The difference equation describing the steps from the previous paragraph is

$$x_{i+1} = x_i - \delta \frac{\nabla f(x_i)}{|\nabla f(x_i)|}. \tag{3.66}$$

[6]In literature, steepest - and gradient descent are sometimes referred to as being different. However, for simplicity, these will not be differentiated.

An improved algorithm would be to continue iterating even though the minimum is crossed, however, this would cause the constant step-length algorithms to oscillate between two points, e.g. x_1 and x_2 in Figure 3.4. To counter this, a changing step-length δ_i is introduced

$$x_{i+1} = x_i - \delta_i \nabla f(x_i). \tag{3.67}$$

All gradient/steepest descent methods are in principle described by Eq. (3.67)[7]. Some examples are

- Brute Force I $\quad \delta_i = \delta \frac{1}{|\nabla f(x_i)|}$
- Brute Force II $\quad \delta_i = \delta$
- Monotone Decreasing $\quad \delta_i = \delta / i^N$
- Newton's Method $\quad \delta_i = \frac{1}{\nabla^2 f(x_i)}$

Iterative gradient methods will only reveal one local extrema, depending on the choice of x_0 and δ. In order to find several extrema, multiple unique processes can be run sequentially or in parallel with different initial guesses.

3.7.2 Stochastic Gradient Descent

Minimizing stochastic quantities such as the variance and expectation values adds another layer of complications on top of the methods introduced in the previous section. Assuming a closed form expression for the stochastic quantity is unobtainable, the gradient needs to be calculated by using e.g. Monte-Carlo sampling. Eq. (3.56) is an example of such a process.

A precise sampling of the stochastic quantities is expensive and unpractical. Stochastic gradient methods use different techniques in order to make the sampling more effective, such as multiple walkers, thermalization, and more.

Using a finite difference scheme with stochastic quantities is dangerous, as uncertainties in the values will cause the gradient to become unstable when the variations are low close to the minimum. This is illustrated in Figure 3.5.

3.7.3 Adaptive Stochastic Gradient Descent

Adaptive Stochastic Gradient Descent (ASGD) has its roots in the mathematics of automated control theory [32]. The automated process is that of choosing an optimal step-length δ_i for the current transition $x_i \to x_{i+1}$. This process is based on the inner product of the old and the new gradient though a variable X_i

$$X_i \equiv -\nabla_i \cdot \nabla_{i-1}. \tag{3.68}$$

The step-length from Eq. (3.67) is modelled in the following manner in ASGD:

[7] This fact sets the perfect scene for an object oriented implementation of gradient descent methods.

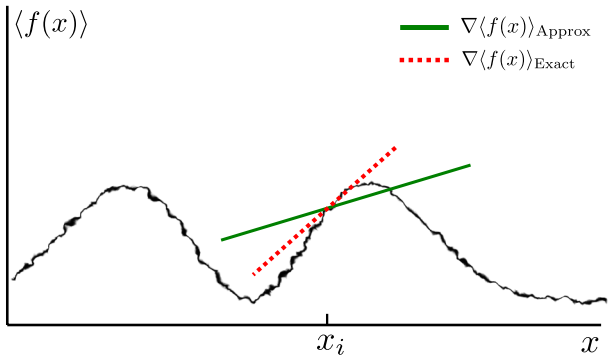


Figure 3.5: A one dimensional plot of an expectation value function. Smeared lines are representing uncertainties due to rough sampling. The direction of the local gradient (solid green line) at a point x_i is not necessarily a good estimate of the actual analytic gradient (dashed red line).

$$\delta_i = \gamma(t_i) \tag{3.69}$$
$$\gamma(t) = a/(t+A) \tag{3.70}$$
$$t_{i+1} = \max(t_i + f(X_i), 0) \tag{3.71}$$
$$f(x) = f_{\min} + \frac{f_{\max} - f_{\min}}{1 - (f_{\max}/f_{\min})e^{-x/\omega}} \tag{3.72}$$

with $f_{\max} > 0$, $f_{\min} < 0$, and $\omega > 0$. The free parameters are a, A and t_0, however, Ref. [33] suggests $A = 20$ and $t_0 = t_1 = A$ for universal usage.

Notice that the step-length increase if t_i decrease and vice-versa. A smaller step-length is sought for regions close to the minimum. The function $f(x)$ is responsible of altering the step-length by changing the trend of t. Close to the minimum, a smaller step-length is sought, and hence t must increase. Being close to the minimum implies that the gradient changes sign frequently. Crossing the minimum with ASGD has the following consequence

- Eq. (3.68): The value of X_i will be positive.
- Eq. (3.72): $f(X_i)$ will return a value in $[0, f_{\max}]$ depending on the magnitude of X_i.
- Eq. (3.71): The value of t will increase, i.e. $t_{i+1} > t_i$.
- Eq. (3.70): The step-length will decrease.

The second step regarding $f(X_i)$ can be visualized in Figure 3.6.

Assumptions

These assumptions are selected direct citations from Ref. [33]. They are listed in order to give an impression that the shapes of the functions used in ASGD are not selected at random, but carefully chosen to work optimally in a stochastic space with averages estimated using very few samples.

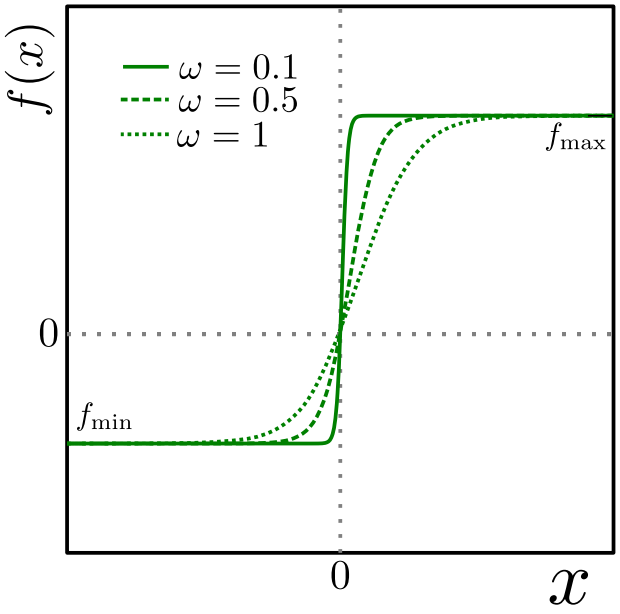


Figure 3.6: Examples of $f(X_i)$ as published in Ref. [33]. As $\omega \to 0$, $f(x)$ approaches a step function.

- The statistical error in the sampled gradients are distributed with zero mean.

This is shown in Ref. [33]; they are normally distributed. The implication is that upon combining gradient estimates for N different processes, the accumulative error will tend to zero quickly.

- The step-length $\gamma(t)$ is a positive monotone decreasing function defined on $[0, \infty)$ with maximum at $t = 0$.

With $\gamma(t)$ being as in Eq (3.70), this is easily shown.

- The function $f(x)$ is continuous and monotone increasing with $f_{\min} = \lim_{x \to \infty} f(x)$ and $f_{\max} = \lim_{x \to -\infty} f(x)$.

This is exactly the behavior displayed in Figure 3.6.

Implementation

A flow chart of the implementation is given in Figure 3.8. For specific details regarding the implementation, see the documentation of the code in Ref. [9]. An example of minimization using ASGD is given in Figure 3.7.

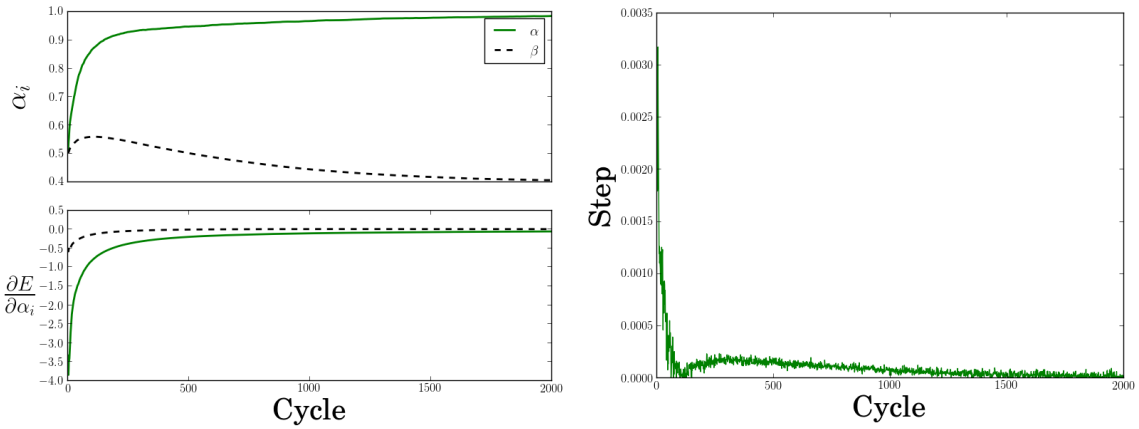


Figure 3.7: Results of Adaptive Stochastic Gradient Descent used on a two-particle quantum dot with unit oscillator frequency using 400 cycles pr. gradient sampling and 40 independent walkers. The right figure shows the evolution of the time-step. The left figure shows the evolution of the variational parameters α and β introduced in Section 3.6 on top, and the evolution of the gradients on the bottom. The gradients are averaged to reveal the pattern underlying the noise. Dispite this averaging, it is apparent that they tend to zero, β somewhat before α. The step rushes to zero with a small rebound as it attempts to cross to negative values.

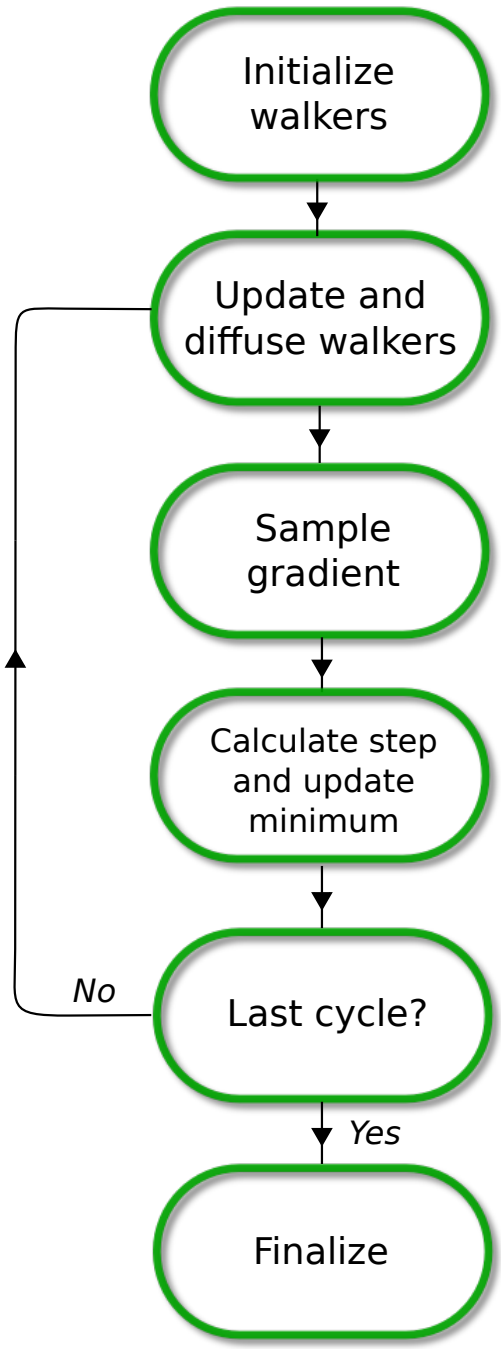


Figure 3.8: Chart flow of ASGD algorithm. Diffusing a walker is done as described in Figure 3.1. Updating the walkers involves recalculating any values affected by updating the minimum. The step is calculated by Eq. (3.70). In the case of QMC, the gradient is sampled by Eq. (3.56).

3.8 Variational Monte-Carlo

As briefly mentioned in the Section 3.1.1, neglecting the branching term, i.e. setting $G_{\mathrm{B}} = 1$, corresponds to a method called Variational Monte-Carlo (VMC). The name comes from the fact that the method is is variational, that is, it supplies an upper bound to the exact ground state energy. The better the trial wave function is, the closer the VMC energy is to the exact ground state energy.

The converged state of the Markov chain in VMC is controlled by the Metropolis test, and will thus equal the trial wave function squared. From the flow chart of the VMC algorithm in Figure 3.9, it is clear that VMC corresponds to a standard Monte-Carlo integration of the local energy sampled using a distribution equal to the trial wave function squared.

3.8.1 Motivating the use of Diffusion Theory

An important question to answer is why e.g. the Langevin equation is so important if the result is simply an expectation value. Statistics states that *any* distribution may be used when calculating an expectation value. Why bother with a trial wave function, thermalization, and so on?

The reason is simple, yet not obvious. The quantity of interest, the local energy, *depends on the trial wave function*. This is demonstrated in the following expression:

$$\begin{aligned} E_{\mathrm{VMC}} &= \int P(\mathbf{r}) \frac{1}{\Psi_T(\mathbf{r})} \widehat{\mathbf{H}} \Psi_T(\mathbf{r}) \mathrm{d}\mathbf{r} \\ &\simeq \frac{1}{n} \sum_{i=1}^{n} \frac{1}{\Psi_T(\mathbf{r}_i)} \widehat{\mathbf{H}} \Psi_T(\mathbf{r}_i), \end{aligned} \tag{3.73}$$

where the points $\mathbf{r}_i$ are drawn from the distribution $P(\mathbf{r})$.

Equation (3.73) implies that the evaluation of the local energy in an arbitrary distribution $P(\mathbf{r})$ is *undefined* at the zeros of $\Psi_T(\mathbf{r})$. In other words, it is not guaranteed that $P(\mathbf{r})$ does not generate a point $\mathbf{r}_m$ in such a way that $\Psi_T(\mathbf{r}_m) = 0$.

However, if the distribution is chosen as $P(\mathbf{r}) = |\Psi_T(\mathbf{r})|^2$, the probability of sampling a point where the local energy is undefined equals zero. This comes as a consequence of the following relation

$$\Psi_T(\mathbf{r}_m) = 0 \quad \Longrightarrow \quad P(\mathbf{r}_m) = 0 \tag{3.74}$$

In other words, the more undefined the energy is at a point, the less probable the point is. This hidden detail is what Quantum Monte-Carlo safely takes care of that standard Monte-Carlo does not.

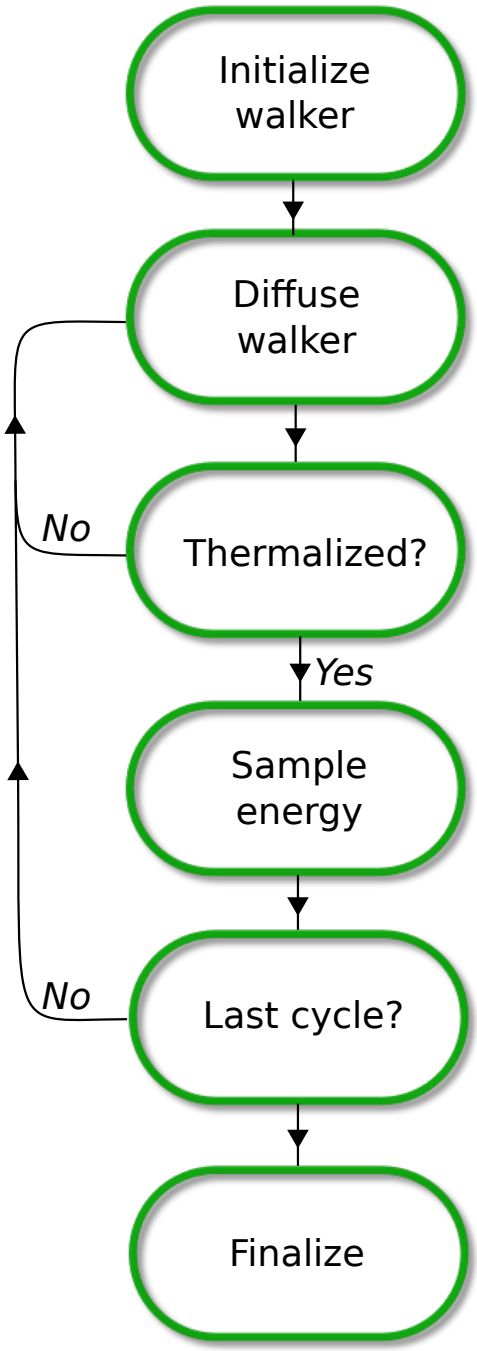


Figure 3.9: Chart flow of the Variational Monte-Carlo algorithm. The second step, *Diffuse Walker*, is the process described in Figure 3.1. Energies are sampled as described in Section 3.6.4. The thermalization is set to a fixed number of cycles.

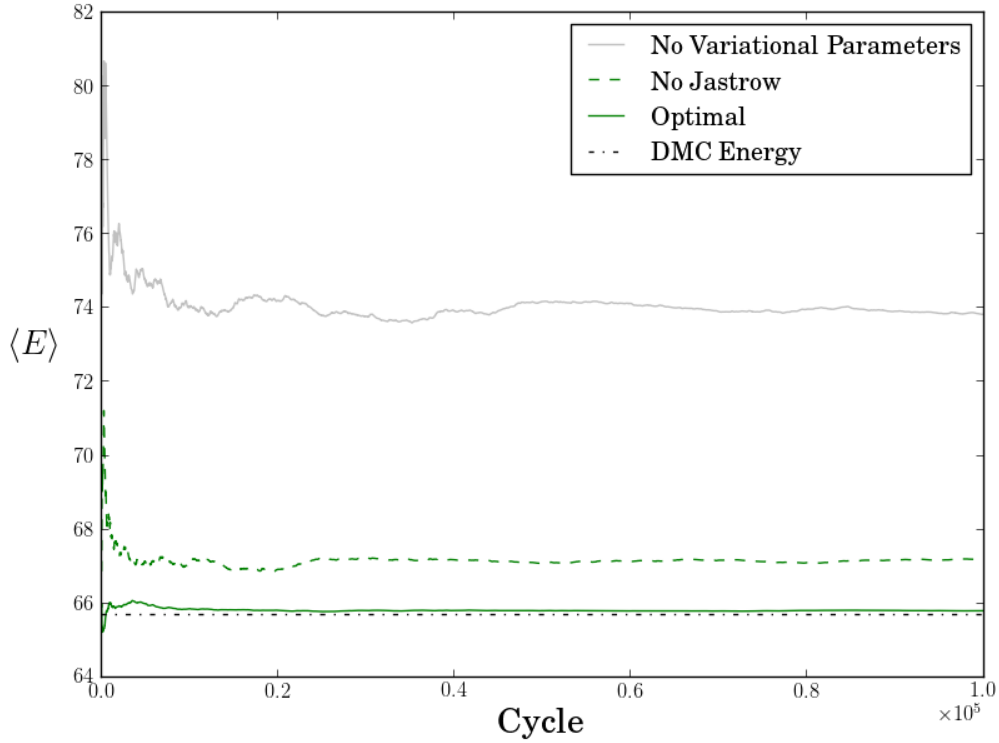


Figure 3.10: Comparison of the VMC energy using different trial wave functions. The DMC energy is believed to be very close to the exact ground state. It is apparent that adding a variational parameter to the trial wave function lowers the energy substantially, however, when adding the Jastrow factor (denoted *Optimal*), described in Section 3.6.2, the VMC energy gets very close to the "exact" answer. A lower energy means a better energy when dealing with variational methods. In this example, a 12-particle quantum dot with unit frequency is used.

3.8.2 Implementation

Beyond a given point, VMC does not benefit much from an increase of samples. It is much more important that the system is properly thermalized, than using several walkers or many cycles. It is therefore sufficient to use a single walker per VMC simulation.

The single walker is initialized in a normal distributed manner, and released to diffuse according to the process in Figure 3.1. After accumulating energies, the final energy is calculated as the mean of these. The process is described in Figure 3.9.

For more details regarding the specific implementation, see the code in Ref. [9].

3.8.3 Limitations

The only limitation in VMC is the choice of the trial wave function. This makes VMC extremely robust; it will *always* produce a result. As the overlap C_0 from Eq. (3.6) approach unity, the VMC energy approaches the exact ground state energy as a monotone decreasing function. Figure 3.10 demonstrates this effect. As more optimizations are added to the trial wave function, the lower the VMC energy gets.

3.9 Diffusion Monte-Carlo

Applying the full diffusion framework introduced in Sections 3.1.1 and 3.2 results in a method known as Diffusion Monte-Carlo (DMC)[8]. Diffusion Monte-Carlo results are often referred to as *exact*, in the sense that DMC is overall one of the most precise many-body methods available. However, just as any many-body method, DMC also has its limitations. These will be discussed in Section 3.9.3.

Where other many-body methods run into the *curse of dimensionality*, that is, the run time scales very bad with increasing number of particles, DMC with its position basis Quantum Monte-Carlo formalism does not. With DMC, it is simply a matter of evaluating a more complicated trial wave function, or simulating for a longer period of cycles, in order to reach convergence to the believed "exact" ground state in a satisfactory way.

3.9.1 Implementation

Branching is a major part of the DMC algorithm. Diffusion Monte-Carlo uses a large ensemble of walkers to generate enormous amounts of statistics. These walkers are initialized using a VMC calculation, i.e. the walkers are initially distributed according to the trial wave function squared.

There are three layers of loops in the DMC method implemented in this thesis, two of which are obvious: The time-step - and walker loops. However, introducing a third *block loop* within the walker loop boosts the convergence dramatically. For each walker, this loop continues until the walker is either dead ($G_B = 0$), or has diffused n_b times. Using this method, "good" walkers will have multiple offspring pr cycle, while "bad" walkers will rarely survive the block loop. Perfect walkers will supply a ton of statistics as they surf through all the loops without branching ($G_B \sim 1$).

A flow chart of the DMC algorithm is given in Figure 3.11. Comparing it the the corresponding figure for VMC, it is apparent that DMC is by far more complicated.

3.9.2 Sampling the Energy

As mentioned in Section 3.1.1, the role of the branching Green's function is to distribute the correct weights to each walker. Each walker's contribution to the cumulative energy sampling should thus be weighed according to the value of the branching Green's function. Let E_k denote the cumulative energy for time-step $\tau = k\delta\tau$, n_w be the number of walkers in the system at time-step k, $\tilde{n}_{b,i}$ be the number of blocks walker i survives, and let $W_i(\mathbf{r}, \tau)$ represent walker i. The relation is then

$$E_k = \frac{1}{n_w} \sum_{i=1}^{n_w} \frac{1}{\tilde{n}_{b,i}} \sum_{l=1}^{\tilde{n}_{b,i}} G_{\mathrm{B}}\Big(W_i(\mathbf{r}, \tau_k + l\delta\tau)\Big) E_L\Big(W_i(\mathbf{r}, \tau_k + l\delta\tau)\Big). \tag{3.75}$$

As required, setting $G_B = n_w = n_b = 1$ reproduces the VMC expression.

The new trial energy from Eq. (3.13) is set equal to the previous cycle's average energy, that is

$$E_T = E_k. \tag{3.76}$$

The DMC energy is updated each cycle to be the trailing average of the trial energies:

[8] In literature, DMC is also known as *Projection Monte-Carlo*, for reasons described in Section 3.1.1.

$$E_{\text{DMC}} = \overline{E_T} = \frac{1}{n}\sum_{k=1}^{n} E_k. \tag{3.77}$$

3.9.3 Limitations

By introducing the branching term, DMC is a far less robust method compared to VMC. Action must be taken in order to stabilize the iterations through tuning of parameters such as population size, time-step, block size, etc. This is the disadvantage of DMC compared to other many-body methods such as Coupled Cluster [2], which is far more automatic.

Time-step errors

The error introduced by the short time approximation in Eq. (3.14) goes as $\mathcal{O}(\delta\tau^2)$. In addition, there is a second error related to the time-step, arising from the fact that not all steps are accepted by the Metropolis algorithm. This introduces a effective reduction in the time-step, and is countered by scaling the time-step with the acceptance ratio upon calculating G_B [21]. However, DMC is rarely used without importance sampling, which, due to the quantum force, has an acceptance ratio very close to one. It is therefore common to ignore this problem, as its effect on the final result is minimal.

Selecting the time-step

Studying the branching Green's function in equation 3.38 in more detail reveals that it's magnitude increases exponentially with the spread of the energies ΔE, that is

$$G_B \propto \exp\left(\Delta E \delta\tau\right). \tag{3.78}$$

As will be shown in Section 3.11.1, the spread in energy samples are higher the worse of an approximation to the ground state the trial wave function is. In addition, the magnitude of the spread scales with the magnitude of the energy.

Setting an upper bound to the branching function might seem like a good idea, however, each time a walker is denied its replicas, an uncontrollable error is introduced in the distribution.

The solution is to balance out the increase in ΔE by lowering the time-step accordingly. That is

$$\delta\tau \propto \frac{1}{\Delta E}. \tag{3.79}$$

However, having too low a time-step will hinder DMC from evolving walkers efficiently, especially if the positional span of the distribution is large. In other words, a balance must be found. An indication of whether the time-step was chosen too low or not is obtained by looking at the resulting DMC density. If the density is spiky and disorganized, the time-step was too low.

Another source of error is due to the *fixed node approximation*. This approximation will be covered in the next section.

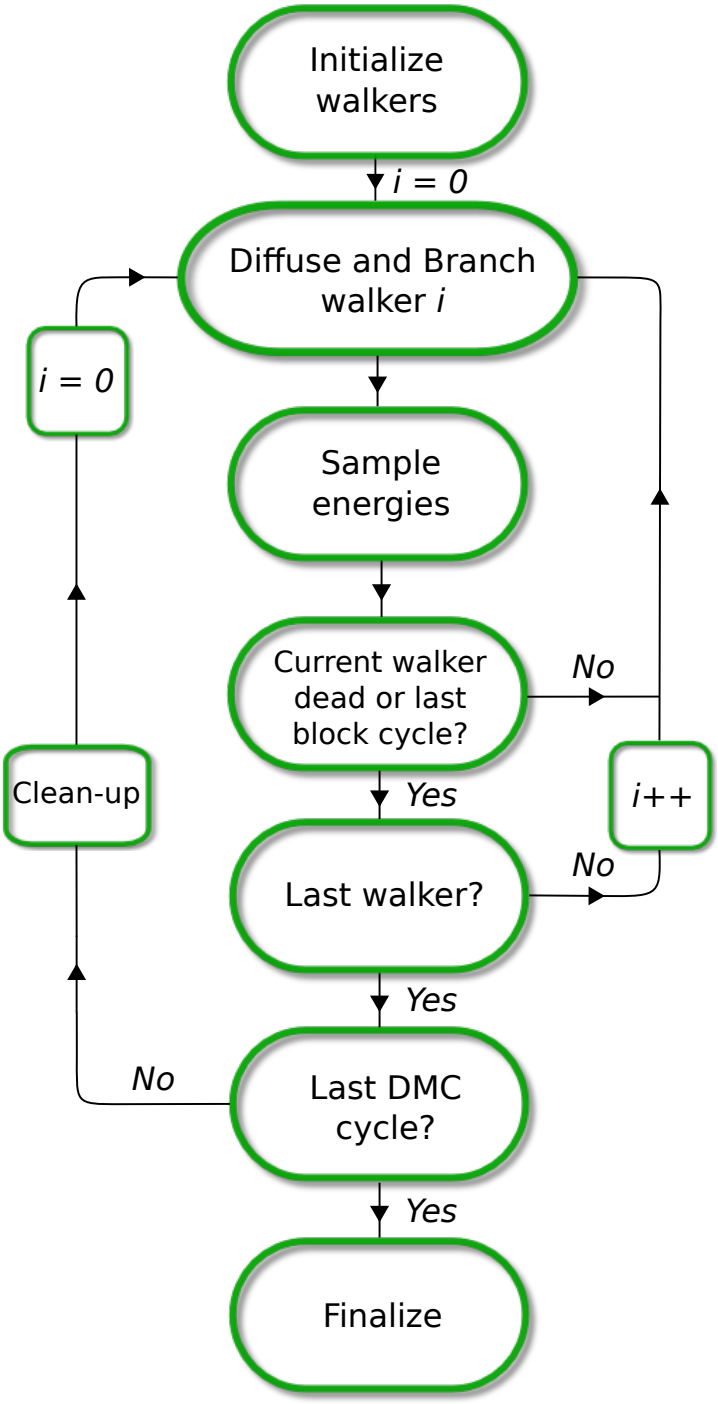


Figure 3.11: Chart flow of the Diffusion Monte-Carlo algorithm. The variable i represents the currently moved walker. The second step, *Diffuse and Branch Walker*, is the process described in Figure 3.1 in combination with the branching from Figure 3.2. Energies are sampled as in Eq. (3.75). Thermalization is set to a fixed number of cycles.

3.9.4 Fixed node approximation

Looking at Eq. (3.48), it is apparent that by choosing positive phases for the single-particle wave functions, the bosonic many-body wave function is exclusively positive. For fermions however, the sign change upon interchanging two particles introduces the possibility that the wave function will have both negative and positive regions, independent of the choice of phases in the single-particle wave functions.

As DMC iterates, the density of walkers at a given time, $P(\mathbf{r}, \tau)$, represents the projected wave function from Eq. (3.6) multiplied by the trial wave function.

$$P(\mathbf{r}, \tau) = \Phi(\mathbf{r}, \tau)\Psi_T(\mathbf{r}). \tag{3.80}$$

This relationship is described in high detail in Section 3.2.2.

Applying the projector operator to the distribution yields

$$\lim_{\tau\to\infty} P(\mathbf{r}, \tau) = \langle\Phi_0|\Phi_T\rangle\, \Phi_0(\mathbf{r})\Psi_T(\mathbf{r}), \tag{3.81}$$

which, if interpreted as a density, should always be greater than zero. In the case of fermions, this is not guaranteed, as the node structure of the exact ground state and the trial wave function will generally be different.

To avoid this anomaly in the density, $\Phi(\mathbf{r}, \tau)$ and $\Psi_T(\mathbf{r})$ have to change sign simultaneously[9]. The brute force way of solving this problem is to *fix* the nodes by rejecting a walker's step if the trial wave function changes sign:

$$\frac{\Psi_T(\mathbf{r}_i)}{\Psi_T(\mathbf{r}_j)} < 0 \quad \Longrightarrow \quad A(i \to j) = 0, \tag{3.82}$$

where $A(i \to j)$ is the probability of accepting the move, as described in Section 3.4. An illustrative example is given in Figure 3.12.

3.10 Estimating One-body Densities

The one-body density is defined as

$$\rho(\mathbf{r}_1) = \iint_{\mathbf{r}_2\mathbf{r}_3} \ldots \int_{\mathbf{r}_N} |\Phi(\mathbf{r}_1\mathbf{r}_2 \ldots \mathbf{r}_N)|^2 \, \mathrm{d}\mathbf{r}_2 \ldots \mathrm{d}\mathbf{r}_N. \tag{3.83}$$

Unlike the distribution $|\Phi(\mathbf{r})|^2$, which describes the distribution of any of the particles in the system, the one-body density $\rho(\mathbf{r}_1)$ describes the simultaneous distribution of every particle in the system, that is, $\rho(\mathbf{r}_1)\mathrm{d}\mathbf{r}_1$ represents the probability of finding *any* of the system's N particles within the volume element $\mathrm{d}\mathbf{r}_1$. Due to the indistinguishable nature of the particles, the fact that the first coordinate is chosen is purely conventional; any of the N coordinates contain information about all the particles. For the same reason, the one-body density should be normalized to the number of particles, and not to unity.

[9] It should be mentioned that more sophisticated methods exist for dealing with the sign problem, some of which splits the distribution of walkers into a negative and a positive regions, however, due to the position space being infinite, this requires an enormous amount of walkers to succeed.

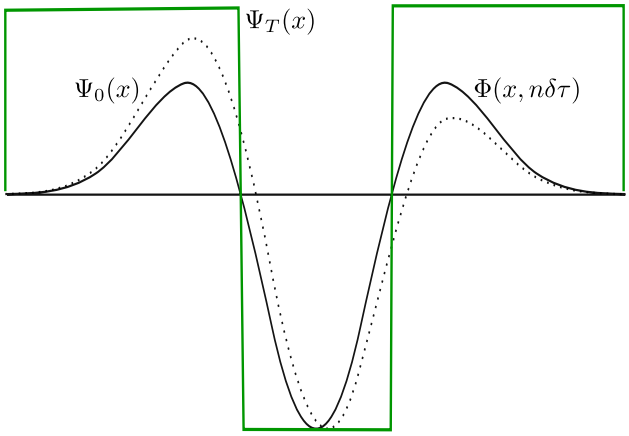


Figure 3.12: A one-dimensional illustration of the fixed node approximation. The dotted line represents the exact ground state $\Psi_0(x)$. The distribution of walkers after n Monte-Carlo cycles is represented by $\Phi(x, n\delta\tau)$. The trial wave function $\Psi_T(x)$ shares nodes with $\Phi(x, n\delta\tau)$, making it impossible for $\Phi(x, n\delta\tau)$ to match $\Psi_0(x)$.

In a Monte-Carlo simulation, estimating the one-body density is done by collecting snapshots of the walkers' positions. These snapshots serve as samples to a histogram where each set of Cartesian coordinates give rise to one histogram-count.

3.10.1 Estimating the Exact Ground State Density

It was mentioned in the section on VMC that the final state of convergence was to have to walkers span the trial wave function. This implies that the one-body density of the trial wave function, called the *variational density*, can be calculated using positional data generated from VMC.

The challenge lies in estimating the one-body density of the exact wave function given a set of DMC data. As described in Section 3.2.2, the distribution of walkers span the *mixed density* $f(\mathbf{r}, \tau) = \Phi(\mathbf{r}, \tau)\Psi_T(\mathbf{r})$, which does not correspond to the ground state distribution unless the trial wave function is indeed the exact ground state. Hence a histogram of the DMC data does not suffice when it comes to presenting the exact wave function.

This implies the need of a method for transforming the one-body density of $f(\mathbf{r}, \tau)$ into the *pure density* $|\Phi(\mathbf{r}, \tau)|^2$. To achieve this, the following relation is needed [21]

$$\begin{aligned}\langle A\rangle_0 = \frac{\langle\Phi_0|\,\widehat{\mathbf{A}}\,|\Phi_0\rangle}{\langle\Phi_0|\Phi_0\rangle} &\simeq 2\frac{\langle\Phi_0|\,\widehat{\mathbf{A}}\,|\Psi_T\rangle}{\langle\Phi_0|\Psi_T\rangle} - \frac{\langle\Psi_T|\,\widehat{\mathbf{A}}\,|\Psi_T\rangle}{\langle\Psi_T|\Psi_T\rangle} + \mathcal{O}(\Delta^2) \\ &= 2\langle A\rangle_{\mathrm{DMC}} - \langle A\rangle_{\mathrm{VMC}} + \mathcal{O}(\Delta^2),\end{aligned} \tag{3.84}$$

where $\Delta \equiv \Psi_T(\mathbf{r}) - \Phi(\mathbf{r}, \tau)$.

Expressed in terms of *density operators*, the expectation values for the different methods are [34]

$$\begin{aligned}\langle A\rangle_0 &= \mathrm{tr}(\widehat{\rho}_0\widehat{\mathbf{A}})\\ \langle A\rangle_{\mathrm{VMC}} &= \mathrm{tr}(\widehat{\rho}_{\mathrm{VMC}}\widehat{\mathbf{A}})\\ \langle A\rangle_{\mathrm{DMC}} &= \mathrm{tr}(\widehat{\rho}_{\mathrm{DMC}}\widehat{\mathbf{A}})\end{aligned}$$

where tr denotes the *trace*, i.e. the sum of all eigenvalues. Inserting these equations into Eq. (3.84) yield

$$\begin{aligned}\mathrm{tr}(\widehat{\rho}_0\widehat{\mathbf{A}}) &\simeq 2\mathrm{tr}(\widehat{\rho}_{\mathrm{DMC}}\widehat{\mathbf{A}}) - \mathrm{tr}(\widehat{\rho}_{\mathrm{VMC}}\widehat{\mathbf{A}}) + \mathcal{O}(\Delta^2)\\ &\simeq \mathrm{tr}\left((2\widehat{\rho}_{\mathrm{DMC}} - \widehat{\rho}_{\mathrm{VMC}})\,\widehat{\mathbf{A}}\right) + \mathcal{O}(\Delta^2),\end{aligned} \tag{3.85}$$

which leads to the conclusion that the mixed density can be transformed in the following manner:

$$\widehat{\rho}_0 \simeq 2\widehat{\rho}_{\mathrm{DMC}} - \widehat{\rho}_{\mathrm{VMC}}. \tag{3.86}$$

It is clear that if this relation holds for regular densities, it will hold for one-body densities as well. In other words, the resulting densities from DMC can be combined with the corresponding one from VMC to produce the pure density.

3.10.2 Radial Densities

Integrating out every degree of freedom except one radial coordinate from the density results in the radial one-body density times the radius (squared for three dimensions). In other words

$$\begin{aligned}I_{3\mathrm{D}} &= \iint \rho(\mathbf{r}_1,\theta_1,\phi_1) r_1^2 \sin\theta_1 \mathrm{d}\theta_1 \mathrm{d}\phi_1\\ &\propto r_1^2\rho(\mathbf{r}_1),\end{aligned} \tag{3.87}$$

$$\begin{aligned}I_{2\mathrm{D}} &= \int \rho(\mathbf{r}_1,\phi_1) r_1 \mathrm{d}\phi_1\\ &\propto r_1\rho(\mathbf{r}_1).\end{aligned} \tag{3.88}$$

In practice, these integrals are calculated by creating histograms $H(r)$ of all sampled radii. Transforming the histograms into radial one-body densities $\rho(r_1)$ are according to Eq. (3.87) and Eq. (3.88) done in the following manner

$$\rho(\mathbf{r}_1) = \frac{H(\mathbf{r}_1)}{r_1^{(d-1)}}, \tag{3.89}$$

where d denotes the number of dimensions. An example presenting two radial one-body densities is given in Figure 3.13. Notice, however, that the radial density for certain systems such as atoms needs to be scaled with r or r^2 in order to reveal the interesting shapes.

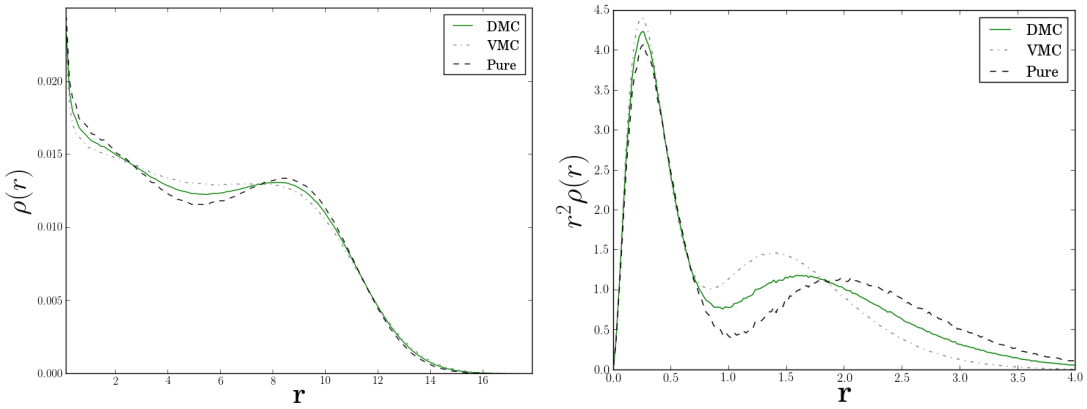


Figure 3.13: Two examples of radial one-body densities for VMC, DMC, and the pure density from Eq. (3.86). On the left: A 12-particle two-dimensional quantum dot with frequency $\omega = 0.1$. The density diverges close to zero due to a "$\frac{0}{0}$" expression (see Eq. (3.89)). On the right: Unscaled radial density for the beryllium atom, i.e. $r_1^2\rho(\mathbf{r}_1)$. The densities will be discussed in the result section.

3.11 Estimating the Statistical Error

As with any statistical result, the statistical errors need to be supplied in order for the result to be considered useful. Systematic errors, that is, errors introduced due to limitations in the model, is discussed in each method's respective section, and is not be related to the statistical error, that is, the exact solution does not necessarily have to be within the error. This is only true if there are no systematic errors.

Statistical errors can be estimated using several methods, some of which are *naive* in the sense that they assume the dataset to be completely *uncorrelated*, i.e. independent of each other.

3.11.1 The Variance and Standard Deviates

Given a set of samples, e.g. local energies, their *variance* is a measure of their spread from the true mean value. The definition of variance reads

$$\begin{aligned} \mathrm{Var}(E) &\equiv \left\langle (E - \langle E \rangle)^2 \right\rangle \\ &= \left\langle E^2 \right\rangle - 2\underbrace{\langle E \langle E \rangle \rangle}_{\langle E \rangle \langle E \rangle} + \langle E \rangle^2 \\ &= \left\langle E^2 \right\rangle - \langle E \rangle^2 . \end{aligned} \tag{3.90}$$

(3.91)

A problem with this definition is that the exact mean $\langle E \rangle$ needs to be known. In a Monte-Carlo simulation, the resulting average $\overline{E}$ is an approximation to the exact mean. Thus the following approximation has to be done:

$$\mathrm{Var}(E) \simeq \overline{E^2} - \overline{E}^2. \tag{3.92}$$

In the case of having the exact wave function, i.e $|\Psi_T\rangle = |\Psi_0\rangle$, the variance becomes zero:

$$\begin{aligned} \mathrm{Var}(E)_{\mathrm{Exact}} &= \langle\Psi_0|\,\widehat{\mathbf{H}}^2\,|\Psi_0\rangle - \langle\Psi_0|\,\widehat{\mathbf{H}}\,|\Psi_0\rangle^2 \\ &= E_0^2 - (E_0)^2 \\ &= 0 \end{aligned}$$

The variance is in other words an excellent measure of how close to the exact wave function the trial wave function is, hence the variance minimum is sometimes used to obtain optimal variational parameters discussed in Section 3.6.3.

A common misconception is that the numerical value of the variance can be used to compare properties of *different* systems. For instance, if system A has variance equal to half of system B's, one could easily conclude that system A has the best fitting trial wave function. However, this is not generally true. The variance has unit energy squared (in the case of local energies), and will thus scale with the magnitude of the energy. One can only safely use the variance as a direct measure locally in each specific system, e.g. in simulations of the beryllium atom.

Another misconception is that the variance is a direct numerical measure of the error. This can in no way be true given that the units mismatch. The *standard deviation*, σ, is the square root of the variance, that is

$$\sigma^2(x) = \mathrm{Var}(x), \tag{3.93}$$

and has a unit equal to that of the measured value. It is therefore related to the *spread* in the sampled value; zero deviation implies perfect samples, while increasing deviation means increasing spread and statistical uncertainty. The standard deviation is in other words a useful quantity when it comes to calculating the error, i.e. the expected deviation from the exact mean $\langle E\rangle$.

3.11.2 The Covariance and correlated samples

It was briefly mentioned in the introduction that certain error estimation techniques are too naive in the sense that they do not account for samples being correlated. Two samples, x and y, are said to be correlated if their *covariance*, $\mathrm{Cov}(x, y)$, is non-zero. The covariance is defined the following way:

$$\begin{aligned} \mathrm{Cov}(x,y) &\equiv \langle (x - \langle x\rangle)(y - \langle y\rangle)\rangle \\ &= \langle xy - x\langle y\rangle - \langle x\rangle y + \langle x\rangle\langle y\rangle\rangle \\ &= \langle xy\rangle - \langle x\langle y\rangle\rangle \underbrace{- \langle y\langle x\rangle\rangle + \langle\langle x\rangle\langle y\rangle\rangle}_{0} \\ &= \langle xy\rangle - \langle x\rangle\langle y\rangle\,. \end{aligned} \tag{3.94}$$

Using this definition, whether or not the samples are correlated boils down to whether or not $\langle xy\rangle = \langle x\rangle\langle y\rangle$. Notice that the variance is the diagonal elements of the covariance matrix, i.e $\mathrm{Cov}(x, x) = \mathrm{Var}(x)$.

The consequence of ignoring the correlations is an error estimate which is generally smaller than the true error; correlated samplings are more clustered, i.e. less spread, due to previous samples' influence on the value of the current sample[10]. Denoting the true standard deviation as σ_c, the above discussion can be distilled to

[10] Samples in QMC are obviously correlated due to the nature of the Langevin equation (difference equation).

$$\sigma_c(x) \geq \sigma(x), \tag{3.95}$$

where $\sigma(x)$ is the deviation from Eq. (3.93).

3.11.3 The Deviate from the Exact Mean

There is an important difference between the deviate from the exact mean, and the deviate of a single sample from its combined mean. In other words:

$$\sigma(\overline{x}) \neq \sigma(x). \tag{3.96}$$

Imagine doing a number of simulations, each resulting in a unique $\overline{x}$. The quantity of interest is not the error of single samples, but the error in the calculated mean. Let m denote the outcome of a single Monte-Carlo calculation. That is

$$m = \frac{1}{n}\sum_{i=1}^{n} x_i. \tag{3.97}$$

The error in the mean is obtained by calculating the standard deviation in m. That is, calculating

$$\sigma^2(m) \quad = \quad \left\langle m^2 \right\rangle - \left\langle m \right\rangle^2. \tag{3.98}$$

Combining the two above equations yield

$$\begin{aligned}
\sigma^2(m) &= \left\langle \frac{1}{n^2}\left[\sum_{i=1}^{n} x_i\right]^2 \right\rangle - \left\langle \frac{1}{n}\sum_{i=1}^{n} x_i \right\rangle^2 \\
&= \frac{1}{n^2}\left(\left\langle \sum_{i=1}^{n} x_i \sum_{j=1}^{n} x_j \right\rangle - \left\langle \sum_{i=1}^{n} x_i \right\rangle \left\langle \sum_{j=1}^{n} x_j \right\rangle \right) \\
&= \frac{1}{n^2}\sum_{i,j=1}^{n} \left\langle x_i x_j \right\rangle - \left\langle x_i \right\rangle \left\langle x_j \right\rangle \\
&= \frac{1}{n^2}\sum_{i,j=1}^{n} \mathrm{Cov}(x_i, x_j).
\end{aligned} \tag{3.99}$$

This result is important; the true error is given in terms of the covariance, and is, as discussed previously, only equal to the sample variance if the samples are uncorrelated. Going back to the definition of covariance in Eq. (3.94), it is apparent that in order to calculate the covariance as in Eq. (3.99), the true mean $\langle x_i \rangle$ needs to be known. Using m as an approximation to the exact mean yields

$$\begin{aligned}
\mathrm{Cov}(x_i, x_j) &= \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \\
&\simeq \langle (x_i - m)(x_j - m) \rangle \\
&\simeq \frac{1}{n^2} \sum_{k,l=1}^{n} (x_k - m)(x_l - m) \qquad (3.100) \\
&\equiv \frac{1}{n} \mathrm{Cov}(x). \qquad (3.101)
\end{aligned}$$

Inserting this relation into Eq. (3.99) yields

$$\begin{aligned}
\sigma^2(m) &= \frac{1}{n^2} \sum_{i,j=1}^{n} \mathrm{Cov}(x_i, x_j) \\
&\simeq \frac{1}{n^2} \sum_{i,j=1}^{n} \frac{1}{n} \mathrm{Cov}(x) \\
&= \frac{1}{n^3} \mathrm{Cov}(x) \underbrace{\sum_{i,j=1}^{n}}_{n^2} \\
&= \frac{1}{n} \mathrm{Cov}(x), \qquad (3.102)
\end{aligned}$$

which serves as an estimate of the full error including correlations.

Explicitly computing the covariance is rarely done in Monte-Carlo simulations; if the sample size is large, it is extremely expensive. A variety of alternative methods to counter the correlations are available, the simplest of which is to define a *correlation length*[11], τ, which defines an interval at which points from the sampling sets are used for actual averaging. In other words, only the points $x_0, x_\tau, ..., x_{n\tau}$ are used in the calculation of the mean. In other words

$$m = \frac{1}{n} \sum_{k=0}^{n} x_{k \cdot \tau}. \qquad (3.103)$$

This implies that $n\tau$ samples are needed in order to get the same amount of samples to the average as in Eq. (3.97); the *effective sample size* becomes $n_{\mathrm{eff}} = n_{\mathrm{tot}}/\tau$. In the cases where $\tau = 1$, the sample set is uncorrelated. For details regarding the derivations of τ based on the covariance, see Refs. [35] and [26].

3.11.4 Blocking

Introducing correlation lengths in the system solver are not an efficient option. Neither is calculating the covariance of billions of data points. However, the error is not a value vital to the simulation process, i.e. there is no need to know the error at any stage during the sampling. This means that the error estimation can be done post process (given that the sample set is stored).

An efficient algorithm for calculating the error of correlated data is *blocking*. This method is described in high detail in Ref. [35], however, details aside, the idea itself is quite intuitive: Given a set of N samples from a single Monte-Carlo simulation, imagine dividing the dataset into *blocks* of n samples, that is, into

[11] In literature, this parameter is often referred to as the *auto-correlation time*.

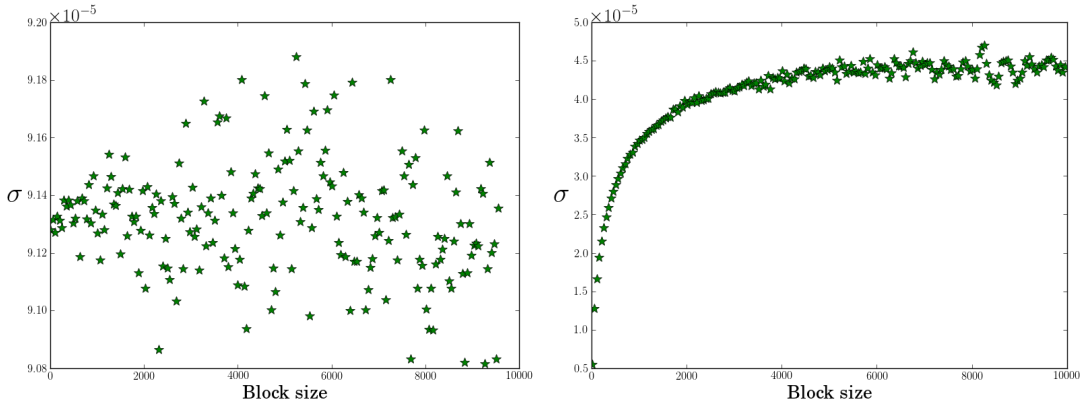


Figure 3.14: Left hand side: Blocking result of (approximately) uncorrelated data generated from a uniform Monte-Carlo integration of $\int_1^2 2x\mathrm{d}x$ resulting in 3.00003 (exact is 3.0). This is in excellent agreement with the magnitude of the error $\sim 9 \cdot 10^{-5}$. There is no sign of a plateau, which implies fairly uncorrelated data (the span of the spread is small and apparently random). Right hand side: Blocking result of a DMC simulation of a 6-particle two-dimensional quantum dot with frequency $\omega = 0.1$. The plateau is strongly present, implying correlated data. The resulting total error is $\sim 4.5 \cdot 10^{-5}$.

blocks of size $n_b = N/n$. The error in each block σ_n calculated using Eq. (3.102) will naturally increase as n decrease, that is

$$\sigma_n \propto \frac{1}{\sqrt{n}}. \tag{3.104}$$

However, treating each block as an individual simulation, n_b averages m_n can be used to calculate the total error from Eq. (3.98), that is, estimate the covariance. This is demonstrated in the following expression

$$\overline{m_n^r} \equiv \frac{1}{n_b}\sum_{k=1}^{n_b} m_k^r, \tag{3.105}$$

$$\begin{aligned}\sigma^2(m) &= \left\langle m^2 \right\rangle - \langle m \rangle^2 \\ &\simeq \overline{m_n^2} - (\overline{m_n})^2.\end{aligned} \tag{3.106}$$

The approximation in Eq. (3.106) should hold for a range of different block sizes, however, just as there is no a priori way of telling the correlation length, there is no a priori way of telling how many blocks is needed. However, what is known, is that if the system is correlated, there should be a range of different block sizes which fulfills Eq. (3.106) to reasonable precision.

The result of a blocking analysis is therefore a series of $(n, \sigma(m_n))$ pairs which can be plotted. The plot should in light of previous arguments result in a increasing curve which stabilizes over a certain span of block sizes. This plateau will then serve as a reasonable approximation to the covariance, that is, the true error. See Figure 3.14 for a demonstration of blocking plots.

3.11.5 Variance Estimators

The standard intuitive variance estimator

$$\sigma^2(x) \simeq \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2 = \left(\frac{1}{n}\sum_{i=1}^{n} x_i^2\right) - \overline{x}^2, \tag{3.107}$$

is just an example of a variance estimator. A more precise estimator is

$$\sigma^2(x) \simeq \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2 = \left(\frac{1}{n-1}\sum_{i=1}^{n} x_i^2\right) - \frac{n}{n-1}\overline{x}^2, \tag{3.108}$$

which is only noticeably different from Eq. (3.107) when the sample size gets small, as it does in blocking analysis. It is therefore standard to use Eq. (3.108) for blocking errors.

4

Generalization and Optimization

There is a big difference in strategy between writing code for a specific problem, and creating a general solver. A general Quantum Monte-Carlo (QMC) solver involves several layers of complexity, such as support for different potentials, single-particle bases, sampling models, etc., which may easily lead to a *combinatorical explosion* if the planning is not done right.

This chapter begins by introducing a list of underlying assumptions regarding the modelled systems. Whether or not a system can be solved is then a question of whether the listed assumptions are valid for the system or not. The next part will cover generalization, that is, the flexibility of the code and the strategies used to obtain this flexibility. The result of a generalized code is that different systems and algorithms can be implemented by making simple changes to the code. Finally, optimizations will be covered. Optimizations are crucial in order to maintain efficiency for a high number of particles.

4.1 Underlying Assumptions and Goals

In large computational projects it is custom to plan every single part of the program before the actual process of coding begins. Coding massive frameworks without planning almost exclusively result in unforeseen consequences, rendering the code difficult to expand, disorganized, and inefficient. The code used in this thesis has been completely restructured four times. This section will cover the assumptions regarding the modelled systems and the goals regarding generalization and optimization made in the planning stages preliminary to the coding process.

4.1.1 Assumptions

The code structure was designed based on the following assumptions

(i) The particles of the simulated systems are either all fermions or all bosons.

(ii) The Hamiltonian is spin - and time independent.

(iii) The trial wave function of a fermionic system is a single determinant.

(iv) A bosonic system is modelled by all particles being in the same assumed single-particle ground state.

As discussed in Section 3.6, the second assumption implies that the Slater determinant can be split into parts corresponding to different spin eigenvalues. The time-independence is a requirement on the QMC solver explained in Section 3.1.1. The assumptions listed are considered true for any system which is implemented in the code, and will thus be applied in all of the following sections.

4.1.2 Generalization Goals

The implementation should be general for:

(i) Fermions and bosons.

(ii) Anisotropic- and isotropic diffusion, i.e. Brute Force - or Importance sampling.

(iii) Different gradient descent algorithms.

(iv) Any Jastrow factor.

(v) Any error estimation algorithm.

(vi) Any single-particle basis, including expanded single-particle bases.

(vii) Any combination of any potentials.

In addition, the following constraint is set on the solvers:

(viii) Full numerical support for all values involving derivatives.

The challenge is, despite the increase in the number of different combinations, to preserve simplicity and structure as layers of complexity are added. Achieving generalization by the use of conditional if tests inside the solvers is considered inefficient, and should only be used if no other solution is apparent or exists.

4.1.3 Optimization Goals

Modern computers have a vast amount of physical memory available, which makes run time optimizations favored over memory optimizations. The following list may appear short, but every step brings immense amounts of complexity to the implementation

(i) Identical values should never be re-calculated.

(ii) Generalization should not be achieved through conditional if tests in repeated function calls, but rather through polymorphism (see Section 2.2.4).

(iii) Linear scaling of run time vs. the number of processors (CPUs) for large simulations.

Goal (ii) has been the ground pillar of the code design.

4.2 Specifics Regarding Generalization

This section will introduce how object orientation is used to achieve the goals regarding generalization of the code. Several examples are discussed, however, for more details regarding the implementation of methods, see the code in Ref. [9].

4.2.1 Generalization Goals (i)-(vii)

As discussed in Section 3.6.1, the mathematical difference between fermions and bosons (of importance to QMC) is how the many-body wave functions are constructed from the single-particle bases. In the case of fermions, the expression is given in terms of a Slater determinant which, due to the fact that the Hamiltonian is spin-independent, can be split into two parts corresponding to spin up and spin down. On the other hand, for bosons, it is simply the product of all states due to the fact that they are all assumed to occupy the same orbital. This is demonstrated in the following code example, where the wave functions of fermionic and bosonic systems are evaluated:

```
1  double Fermions::get_spatial_wf(const Walker* walker) {
2      using namespace arma;
3
4      //Spin up times Spin down (determinants)
5      return det(walker->phi(span(0, n2 - 1), span())) * det(walker->phi(span(n2, n_p - 1),
           span()));
6  }
7
8  double Bosons::get_spatial_wf(const Walker* walker) {
9
10     double wf = 1;
11
12     //Using the phi matrix as a vector in the case of bosons.
13     //Assuming all particles to occupy the same single-particle state (neglecting
           permutations).
14     for (int i = 0; i < n_p; i++){
15       wf *= walker->phi(i);
16     }
17
18     return wf;
19 }
```

Listing 4.1: The implementation of the evaluation of fermionic and bosonic wave functions. Line 5: The fermion class accesses the walker's single-particle state matrix and returns the determinant of the first half (spin up) times the determinant of the second half (spin down). Line 14-16: The boson class simply calculates the product of all the single-particle states.

Overloaded pure virtual methods for fermions and bosons exist for all of the methods which involves evaluating the many-body wave function, for example the spatial ratio and the sum of Laplacians. When the `QMC` solver asks the `System*` object for a spatial ratio, depending on whether fermions or bosons are loaded run-time, the fermion or boson spatial ratio is evaluated.

It is apparent that this way of implementing the system class takes care of optimization goal (ii) in Section 4.1.3 regarding no use of conditional if tests to determine the nature of the system.

A similar polymorphic splitting is introduced in the following classes:

- `Orbitals` The hydrogen-like or the harmonic oscillator orbitals.
- `BasisFunctions` Stand-alone single-particle wave functions initialized by `Orbitals`.
- `Sampling` Brute force - or importance sampling.
- `Diffusion` Isotropic or Fokker-Planck diffusion. Automatically selected by `Sampling`.
- `ErrorEstimator` Simple or Blocking.
- `Jastrow` Padé Jastrow - or no Jastrow factor.
- `QMC` Variational - (VMC) or Diffusion Monte-Carlo (DMC).
- `Minimizer` Adaptive Stochastic Gradient Descent (ASGD).

Implementing for example a new Jastrow Factor is done by simply creating a new subclass of `Jastrow`. The QMC solver does not need to change to adapt to the new implementation. For more details, see Section 2.2. The splitting done in `QMC` is done to avoid rewriting a lot of general QMC code, such as diffusing walkers.

A detailed description of the generalization of potentials, i.e. generalization goal (vii), is given in Section 2.2.4.

4.2.2 Generalization Goal (vi) and Expanded bases

An expanded single-particle basis is implemented as a subclass of the `Orbitals` superclass. It wraps around an `Orbitals` implementation, e.g. the harmonic oscillator orbitals, containing basis elements $\phi_\alpha(\mathbf{r}_j)$. In addition to these elements, the expanded basis class has a set of expansion coefficients $\mathbf{C}_{\gamma\alpha}$ from which the new basis elements are constructed in the following manner:

$$\psi_\gamma^{\text{Exp.}}(\mathbf{r}_j) = \sum_{\alpha=0}^{B-1} \mathbf{C}_{\gamma\alpha}\phi_\alpha(\mathbf{r}_j), \tag{4.1}$$

where B is the size of the expanded basis. The following code snippet presents the vital members of the expanded basis class:

```
1  class ExpandedBasis : public Orbitals {
2
3  ...
4
5  protected:
6
7      int basis_size;
8      arma::mat coeffs;
9      Orbitals* basis;
10
11     void calculate_coefficients();
12
13 };
```

Listing 4.2: The declaration of the expanded basis class. The vital members are the size of the basis, the expansion coefficients and another basis in which the new are expanded. A method for calculating the coefficients is present, but the actual implementation has not been a focus of this thesis.

The implementation of Eq. (4.1) into the expanded basis class is achieved by overloading the original `Orbitals::phi` virtual member function as shown in the following example

```
double ExpandedBasis::phi(const Walker* walker, int particle, int q_num) {

    double value = 0;

    //Dividing basis_size by half assuming a two-level system.
    for (int m = 0; m < basis_size/2; m++) {
        value += coeffs(q_num, m) * basis->phi(walker, particle, m);
    }

    return value;

}
```

Listing 4.3: The explicit implementation of the expanded basis single-particle wave function. The wave function is evaluated by expanding a given basis in a set of expansion coefficients (see the previous code example).

Expanded bases has not been a focus for the thesis, thus explicit algorithms for calculating the coefficients will not be covered. The reason the implementation has been presented, is to lay the foundation in case future Master students are to expand upon the code.

4.2.3 Generalization Goal (viii)

Support for evaluating derivatives numerically is important for two reasons; the first being debugging, the second being the cases where no closed-form expressions for the derivatives can be obtained or become too expensive to evaluate.

As an example, the orbital class implementation responsible for the gradient of the single-particle states, `Orbitals::del_phi`, is virtual. This implies that it can be overloaded to call the numerical derivative implementation `Orbitals::num_diff`. The same goes for the Laplacian, the Jastrow factor derivatives, and the variational derivatives in the minimizer. An alternative to numerically evaluating the derivatives of the single-particle wave functions would be to perform the derivative on the full many-body wave function, however, this would not fit naturally into the code design.

The implemented numerical derivatives are finite difference schemes with an error proportional to the square of the chosen step length.

4.3 Optimizations due to a Single two-level Determinant

Assuming the trial wave function to consist of a single term unlocks several optimizations involving the Slater determinant. Similar optimizations for bosons are considered trivial in comparison and will not be covered in detail. See the code in [9] for details regarding bosons.

Writing the Slater determinant as the determinant of the *Slater matrix* $\mathbf{S}$, the expression for the trial wave function in Eq. (3.54) becomes

$$\Psi_T = |\mathbf{S}^{\uparrow}||\mathbf{S}^{\downarrow}|J, \tag{4.2}$$

where the splitting of the Slater determinant into parts corresponding to spin up and spin down from Eq. (3.52) has been applied. The Jastrow-factor, J, is described in Section 3.6.2. Function arguments are skipped to clean up the expressions.

Several quantities involve evaluating the trial wave function, one of which is the quantum force from Section 3.2.2. The expression for the quantum force of particle i is

$$\begin{aligned}\mathbf{F}_i &= 2\frac{\nabla_i\left(|\mathbf{S}^{\uparrow}||\mathbf{S}^{\downarrow}|J\right)}{|\mathbf{S}^{\uparrow}||\mathbf{S}^{\downarrow}|J} \\ &= 2\left(\frac{\nabla_i|\mathbf{S}^{\uparrow}|}{|\mathbf{S}^{\uparrow}|}+\frac{\nabla_i|\mathbf{S}^{\downarrow}|}{|\mathbf{S}^{\downarrow}|}+\frac{\nabla_i J}{J}\right).\end{aligned} \tag{4.3}$$

The important part to realize now, is that particle i *either* has spin up or spin down. This implies that one of the derivatives in the last expression is zero due to the fact that the spins are opposite. Denoting the spin of particle i as α and the opposite spin as $\overline{\alpha}$, the expressions for the quantum force reads

$$\mathbf{F}_i = 2\left(\frac{\nabla_i|\mathbf{S}^{\alpha}|}{|\mathbf{S}^{\alpha}|}+\frac{\nabla_i|\mathbf{S}^{\overline{\alpha}}|}{|\mathbf{S}^{\overline{\alpha}}|}+\frac{\nabla_i J}{J}\right), \tag{4.4}$$

where

$$\nabla_i|\mathbf{S}^{\overline{\alpha}}| = 0, \tag{4.5}$$

due to the fact that there is no trace of particle i in $|\mathbf{S}^{\overline{\alpha}}|$. The resulting expression involves evaluating the Slater matrix for a single spin configuration only

$$\mathbf{F}_i = 2\left(\frac{\nabla_i|\mathbf{S}^{\alpha}|}{|\mathbf{S}^{\alpha}|}+\frac{\nabla_i J}{J}\right).$$

The expression for the local energy from Section 3.6.4 can be simplified in a similar manner. Starting from the original expression

$$E_L = -\frac{1}{2}\sum_i \frac{1}{\Psi_T}\nabla_i^2\Psi_T + \sum_i V_i, \tag{4.6}$$

the Laplacian can be expanded in the same way as was done for the quantum force

$$\begin{aligned}\frac{1}{\Psi_T}\nabla_i^2\Psi_T &= \frac{1}{|\mathbf{S}^{\alpha}||\mathbf{S}^{\overline{\alpha}}|J}\nabla_i^2|\mathbf{S}^{\alpha}||\mathbf{S}^{\overline{\alpha}}|J \\ &= \frac{\nabla_i^2|\mathbf{S}^{\alpha}|}{|\mathbf{S}^{\alpha}|}+\frac{\nabla_i^2|\mathbf{S}^{\overline{\alpha}}|}{|\mathbf{S}^{\overline{\alpha}}|}+\frac{\nabla_i^2 J}{J} \\ &\quad + 2\frac{\left(\nabla_i|\mathbf{S}^{\alpha}|\right)\left(\nabla_i|\mathbf{S}^{\overline{\alpha}}|\right)}{|\mathbf{S}^{\alpha}||\mathbf{S}^{\overline{\alpha}}|}+2\frac{\left(\nabla_i|\mathbf{S}^{\alpha}|\right)\left(\nabla_i J\right)}{|\mathbf{S}^{\alpha}|J}+2\frac{\left(\nabla_i|\mathbf{S}^{\overline{\alpha}}|\right)\left(\nabla_i J\right)}{|\mathbf{S}^{\overline{\alpha}}|J} \\ &= \frac{\nabla_i^2|\mathbf{S}^{\alpha}|}{|\mathbf{S}^{\alpha}|}+\frac{\nabla_i^2 J}{J}+2\frac{\nabla_i|\mathbf{S}^{\alpha}|}{|\mathbf{S}^{\alpha}|}\frac{\nabla_i J}{J},\end{aligned} \tag{4.7}$$

where half of the terms vanish due to Eq. (4.5).

The last expression involving the Slater matrix is the spatial ratio, R_ψ, used in the Metropolis algorithm from Section 3.4. The expression reads

$$R_\psi = \frac{\Psi_T^{\text{new}}}{\Psi_T^{\text{old}}} \tag{4.8}$$

$$= \frac{|\mathbf{S}^\uparrow|^{\text{new}}|\mathbf{S}^\downarrow|^{\text{new}}J^{\text{new}}}{|\mathbf{S}^\uparrow|^{\text{old}}|\mathbf{S}^\downarrow|^{\text{old}}J^{\text{old}}}, \tag{4.9}$$

where the old and new superscript denotes the wave function prior to and after moving one particle, respectively. Let again i be the currently moved particle with spin α. As discussed previously, the part of the trial wave function representing the opposite spin of α, $|\mathbf{S}^{\overline{\alpha}}|$, is independent of particle i. This implies that moving particle i does not change the value of $|\mathbf{S}^{\overline{\alpha}}|$, that is

$$|\mathbf{S}^{\overline{\alpha}}|^{\text{new}} = |\mathbf{S}^{\overline{\alpha}}|^{\text{old}}. \tag{4.10}$$

Inserting this into Eq. (4.8) in addition to the spin parameters α and $\overline{\alpha}$ gives

$$R_\psi = \underbrace{\frac{|\mathbf{S}^{\overline{\alpha}}|^{\text{new}}}{|\mathbf{S}^{\overline{\alpha}}|^{\text{old}}}}_{1} \frac{|\mathbf{S}^{\alpha}|^{\text{new}}}{|\mathbf{S}^{\alpha}|^{\text{old}}} \frac{J^{\text{new}}}{J^{\text{old}}} = \frac{|\mathbf{S}^{\alpha}|^{\text{new}}}{|\mathbf{S}^{\alpha}|^{\text{old}}} \frac{J^{\text{new}}}{J^{\text{old}}}. \tag{4.11}$$

From the expressions deduced in this section it is clear that the dimensionality of the calculations is halved by splitting the Slater determinant into two parts. Calculating the determinant of an $N \times N$ matrix costs $\mathcal{O}(N^2)$ floating point operations (flops), which yields a speedup of four times when estimating the determinants.

4.4 Optimizations due to Single-particle Moves

Moving one particle at the time implies that only a single row in the Slater determinant from Eq. (3.47) will be changed between the calculations. Changing a single row implies that many *co-factors* remain unchanged. Since all of the expressions deduced in the previous section contain ratios of the spatial wave functions, expressing these determinants in terms of their co-factors should reveal a cancellation of terms.

Expressing the cancellation mathematically presents the possibility to optimize the calculations by implementing only the parts which do not cancel.

In the previous section it became apparent that only the determinant whose spin level matches that of the moved particle needs to be calculated explicitly. In the following sections, the spin indication on the Slater matrix $|\mathbf{S}^\alpha|$ will be skipped in order to clean up the equations.

4.4.1 Optimizing the Slater determinant ratio

The inverse of the Slater matrix introduced in the previous section is given in terms of its *adjugate* by the following relation [36]

$$\mathbf{S}^{-1} = \frac{1}{|\mathbf{S}|}\text{adj}\mathbf{S}.$$

The adjugate of a matrix is the transpose of the cofactor matrix $\mathbf{C}$, that is

$$\mathbf{S}^{-1} = \frac{\mathbf{C}^T}{|\mathbf{S}|}, \tag{4.12}$$

$$\mathbf{S}^{-1}_{ij} = \frac{\mathbf{C}_{ji}}{|\mathbf{S}|}. \tag{4.13}$$

Moreover, the determinant can be expressed as a *cofactor expansion* around row j (Kramer's rule) [36]

$$|\mathbf{S}| = \sum_i \mathbf{S}_{ji}\mathbf{C}_{ji}, \tag{4.14}$$

where

$$\mathbf{S}_{ji} = \phi_i(\mathbf{r}_j). \tag{4.15}$$

The spatial part of the R_ψ ratio is obtained by inserting Eq. (4.14) into Eq. (4.11)

$$R_S = \frac{\sum_i \mathbf{S}^{\text{new}}_{ji}\mathbf{C}^{\text{new}}_{ji}}{\sum_i \mathbf{S}^{\text{old}}_{ji}\mathbf{C}^{\text{old}}_{ji}}. \tag{4.16}$$

Let j represent the moved particle. The j'th column of the cofactor matrix is unchanged when the particle moves (column j depends on every column but its own). In other words

$$\mathbf{C}^{\text{new}}_{ji} = \mathbf{C}^{\text{old}}_{ji} = (\mathbf{S}^{\text{old}}_{ij})^{-1}|\mathbf{S}^{\text{old}}|, \tag{4.17}$$

where the inverse relation of Eq. (4.13) has been used. Inserting this into Eq. (4.16) yields

$$\begin{aligned} R_S &= \frac{|\mathbf{S}^{\text{old}}| \sum_i \mathbf{S}^{\text{new}}_{ji}(\mathbf{S}^{\text{old}}_{ij})^{-1}}{|\mathbf{S}^{\text{old}}| \sum_i \mathbf{S}^{\text{old}}_{ji}(\mathbf{S}^{\text{old}}_{ij})^{-1}} \\ &= \frac{\sum_i \mathbf{S}^{\text{new}}_{ji}(\mathbf{S}^{\text{old}}_{ij})^{-1}}{\mathbf{I}_{jj}}. \end{aligned}$$

The diagonal element of the identity matrix is by definition unity. Inserting this fact combined with the relation from Eq. (4.15), an optimized expression for the ratio is obtained:

$$R_S = \sum_i \phi_i(\mathbf{r}^{\text{new}}_j)(\mathbf{S}^{\text{old}}_{ij})^{-1}, \tag{4.18}$$

where j is the currently moved particle. The sum over i spans the Slater matrix whose spin value matches that of particle j, i.e either $\mathbf{S}^\uparrow$ or $\mathbf{S}^\downarrow$, i.e. $\mathbf{S}^\alpha$ introduced in the previous section.

Similar reductions can be applied to all the Slater ratio expressions from the previous section [21, 26]:

$$\frac{\nabla_i |\mathbf{S}|}{|\mathbf{S}|} = \sum_k \nabla_i \phi_k(\mathbf{r}_i^{\text{new}})(\mathbf{S}_{ki}^{\text{new}})^{-1}, \tag{4.19}$$

$$\frac{\nabla_i^2 |\mathbf{S}|}{|\mathbf{S}|} = \sum_k \nabla_i^2 \phi_k(\mathbf{r}_i^{\text{new}})(\mathbf{S}_{ki}^{\text{new}})^{-1}, \tag{4.20}$$

where the sum k spans the Slater matrix whose spin values match that of the moved particle.

Closed form expressions for the derivatives and Laplacians of the single-particle wave functions can be implemented in order to avoid expensive numerical calculations. See Appendices D, E and F for a tabulation of closed form expressions used in this Thesis. Appendix C presents an efficient strategy for obtaining these expressions.

4.4.2 Optimizing the inverse Slater matrix

One might question the efficiency of calculating inverse matrices compared to brute force estimation of the determinants. The efficiency of the inverse becomes apparent, as with the ratio in Eq. (4.18), by co-factor expanding the expression; an updating algorithm which dramatically decreases the cost of calculating the inverse of the new Slater matrix can be implemented.

Let i denote the currently moved particle. The new inverse is given in terms of the previous by the following expression [21, 26]

$$\tilde{\mathbf{I}}_{ij} = \sum_l \mathbf{S}_{il}^{\text{new}} (\mathbf{S}_{lj}^{\text{old}})^{-1}, \tag{4.21}$$

$$(\mathbf{S}_{kj}^{\text{new}})^{-1} = (\mathbf{S}_{kj}^{\text{old}})^{-1} - \frac{1}{R_S}(\mathbf{S}_{ji}^{\text{old}})^{-1}\tilde{\mathbf{I}}_{ij} \qquad j \neq i, \tag{4.22}$$

$$(\mathbf{S}_{ki}^{\text{new}})^{-1} = \frac{1}{R_S}(\mathbf{S}_{ki}^{\text{old}})^{-1} \qquad \text{else.} \tag{4.23}$$

This reduces the cost of calculating the inverse by an order of magnitude down to $\mathcal{O}(N^2)$ flops.

Further optimization can be achieved by calculating the $\tilde{\mathbf{I}}$ vector for particle i prior to performing the loop over k and j. Again, this loop should only update the inverse Slater matrix whose spin value correspond to that of the moved particle.

4.4.3 Optimizing the Padé Jastrow factor Ratio

Such as was done with the Green's function ratio in Eq. (3.36), the ratio between two Jastrow factors are best expressed in terms of the logarithm

$$\log \frac{J^{\text{new}}}{J^{\text{old}}} = \sum_{k<j=1}^{N} \left[\frac{a_{kj} r_{kj}^{\text{new}}}{1 + \beta r_{kj}^{\text{new}}} - \frac{a_{kj} r_{kj}^{\text{old}}}{1 + \beta r_{kj}^{\text{old}}} \right] \tag{4.24}$$

$$\equiv \sum_{k<j=1}^{N} \left[g_{kj}^{\text{new}} - g_{kj}^{\text{old}} \right]. \tag{4.25}$$

The relative distances r_{kj} behave much like the cofactors in Section 4.4.1: Changing r_i changes only r_{ij}, that is

$$r_{kj}^{\text{new}} = r_{kj}^{\text{old}} \qquad k \neq i, \tag{4.26}$$

which inserted into Eq. (4.25) yields

$$\begin{aligned} \log \frac{J^{\text{new}}}{J^{\text{old}}} &= \sum_{k<j\neq i} \underbrace{\left[g_{kj}^{\text{old}} - g_{kj}^{\text{old}}\right]}_{0} + \sum_{j=1}^{N} \left[g_{ij}^{\text{new}} - g_{ij}^{\text{old}}\right] \\ &= \sum_{j=1}^{N} a_{ij} \left(\frac{r_{ij}^{\text{new}}}{1+\beta r_{ij}^{\text{new}}} - \frac{r_{ij}^{\text{old}}}{1+\beta r_{ij}^{\text{old}}} \right). \end{aligned} \tag{4.27}$$

Exponentiating both sides reveals the final optimized ratio

$$\frac{J^{\text{new}}}{J^{\text{old}}} = \exp\left[\sum_{j=1}^{N} a_{ij} \left(\frac{r_{ij}^{\text{new}}}{1+\beta r_{ij}^{\text{new}}} - \frac{r_{ij}^{\text{old}}}{1+\beta r_{ij}^{\text{old}}} \right) \right], \tag{4.28}$$

where i denotes the currently moved particle.

4.5 Optimizing the Padé Jastrow Derivative Ratios

The shape of the Padé Jastrow factor is general in the sense that it is independent of the system at hand. Calculating closed form expressions for the derivatives can then be done once and for all.

Closed form expressions are not only very efficient compared to a numerical evaluation, but also exact to machine precision. These facts render closed form expressions of high interest to any Monte-Carlo implementation.

4.5.1 The Gradient

Using the notation of Eq. (4.25), the x-component of the Padé Jastrow gradient ratio for particle i is

$$\frac{1}{J}\frac{\partial J}{\partial x_i} = \frac{1}{\prod_{k<l} \exp g_{kl}} \frac{\partial}{\partial x_i} \prod_{k<l} \exp g_{kl}. \tag{4.29}$$

Using the product rule, the above product will be transformed into a sum, where only the terms which has k or l equal to i survive the differentiation. In addition, the terms independent of i will cancel the corresponding terms in the denominator. Performing this calculation yields

$$\begin{aligned}\frac{1}{J}\frac{\partial J}{\partial x_i} &= \sum_{k\neq i}\frac{1}{\exp g_{ik}}\frac{\partial}{\partial x_i}\exp g_{ik}\\ &= \sum_{k\neq i}\frac{1}{\exp g_{ik}}\frac{\partial g_{ik}}{\partial x_i}\exp g_{ik}\\ &= \sum_{k\neq i}\frac{\partial g_{ik}}{\partial x_i}\\ &= \sum_{k\neq i}\frac{\partial g_{ik}}{\partial r_{ik}}\frac{\partial r_{ik}}{\partial x_i},\end{aligned} \tag{4.30}$$

where

$$\begin{aligned}\frac{\partial g_{ik}}{\partial r_{ik}} &= \frac{\partial}{\partial r_{ik}}\left(\frac{a_{ik}r_{ik}}{1+\beta r_{ik}}\right)\\ &= \frac{a_{ik}}{1+\beta r_{ik}} - \frac{a_{ik}r_{ik}}{(1+\beta r_{ik})^2}\beta\\ &= \frac{a_{ik}(1+\beta r_{ik}) - a_{ik}\beta r_{ik}}{(1+\beta r_{ik})^2}\\ &= \frac{a_{ik}}{(1+\beta r_{ik})^2},\end{aligned} \tag{4.31}$$

and

$$\begin{aligned}\frac{\partial r_{ik}}{\partial x_i} &= \frac{\partial}{\partial x_i}\sqrt{(x_i-x_k)^2+(y_i-y_k)^2+(z_i-z_k)^2}\\ &= \frac{1}{2}2(x_i-x_k)/\sqrt{(x_i-x_k)^2+(y_i-y_k)^2+(z_i-z_k)^2}\\ &= \frac{x_i-x_k}{r_{ik}}.\end{aligned} \tag{4.32}$$

Combining these expressions yield

$$\frac{1}{J}\frac{\partial J}{\partial x_i} = \sum_{k\neq i}\frac{a_{ik}}{r_{ik}}\frac{x_i-x_k}{(1+\beta r_{ik})^2}. \tag{4.33}$$

Changing the Cartesian variable in the differentiation changes only the numerator of Eq. (4.33). In other words, generalizing to the full gradient is done by substituting the Cartesian difference with the position vector difference. The expression for the gradient becomes

$$\frac{\nabla_i J}{J} = \sum_{k\neq i=1}^{N}\frac{a_{ik}}{r_{ik}}\frac{\mathbf{r}_i-\mathbf{r}_k}{(1+\beta r_{ik})^2}. \tag{4.34}$$

4.5.2 The Laplacian

The same strategy used to obtain the closed form expression for the gradient in the previous section can be applied to the Laplacian. The full calculation is done in Ref. [26]. The expression becomes

$$\frac{\nabla_i^2 J}{J} = \left|\frac{\nabla_i J}{J}\right|^2 + \sum_{k\neq i=1}^{N}\left(\frac{d-1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}} + \frac{\partial^2 g_{ik}}{\partial r_{ik}^2}\right), \tag{4.35}$$

where d is the number of dimensions arising due to the fact that the Laplacian, unlike the gradient, is a summation over the contributions from all dimensions. A simple differentiation of Eq. (4.31) with respect to r_{ik} yields

$$\frac{\partial^2 g_{ik}}{\partial r_{ik}^2} = -\frac{2a_{ik}\beta}{(1+\beta r_{ik})^3} \tag{4.36}$$

Inserting Eq. (4.31) and Eq. (4.36) into Eq. (4.35) further reveal

$$\begin{aligned}\frac{\nabla_i^2 J}{J} &= \left|\frac{\nabla_i J}{J}\right|^2 + \sum_{k\neq i=1}^{N}\left(\frac{d-1}{r_{ik}}\frac{a_{ik}}{(1+\beta r_{ik})^2} - \frac{2a_{ik}\beta}{(1+\beta r_{ik})^3}\right)\\ &= \left|\frac{\nabla_i J}{J}\right|^2 + \sum_{k\neq i=1}^{N} a_{ik}\frac{(d-1)(1+\beta r_{ik}) - 2\beta r_{ik}}{r_{ik}(1+\beta r_{ik})^3},\end{aligned}$$

which when cleaned up results in

$$\frac{\nabla_i^2 J}{J} = \left|\frac{\nabla_i J}{J}\right|^2 + \sum_{k\neq i} a_{ik}\frac{(d-3)(\beta r_{ik}+1)+2}{r_{ik}(1+\beta r_{ik})^3}. \tag{4.37}$$

The local energy calculation needs the sum of the Laplacians for all particles (see Eq. (4.6)). In other words, the quantity of interest becomes

$$\sum_i \frac{\nabla_i^2 J}{J} = \sum_i \left[\left|\frac{\nabla_i J}{J}\right|^2 + \sum_{k\neq i}^{N} a_{ik}\frac{(d-3)(\beta r_{ik}+1)+2}{r_{ik}(1+\beta r_{ik})^3}\right]. \tag{4.38}$$

Due to the symmetry of r_{ik}, the second term count equal values twice. Further optimization can thus be achieved by calculating only the terms where $k > i$, and multiply the sum by two. Bringing it all together yields

$$\sum_i \frac{\nabla_i^2 J}{J} = \sum_i \left|\frac{\nabla_i J}{J}\right|^2 + 2\sum_{k>i} a_{ik}\frac{(d-3)(\beta r_{ik}+1)+2}{r_{ik}(1+\beta r_{ik})^3}. \tag{4.39}$$

4.6 Tabulating Recalculated Data

The expressions introduced so far in this chapter contain terms which are identical. Explicitly calculating these terms every time they are countered in the code would waste a lot of CPU time in contrast to the optimal scenario where they are calculated only once.

Tabulating data is crucial in order for the code to remain efficient for an increased number of particles.

4.6.1 The relative distance matrix

In the discussions regarding the optimization of the Jastrow ratio in Section 4.4.3, it became clear that moving one particle only changed N of the relative distances. Tabulating these distances in a matrix $\mathbf{r}_{\text{rel}}$, which then can be updated when a particle is moved, will ensure that the relative distances are calculated once and for all. The matrix is set up in the following manner:

$$\mathbf{r}_{\text{rel}} = \mathbf{r}_{\text{rel}}^T = \begin{pmatrix} 0 & r_{12} & r_{13} & \cdots & r_{1N} \\ r_{21} & 0 & r_{23} & \cdots & r_{2N} \\ & & \ddots & \ddots & \vdots \\ & \cdots & & 0 & r_{(N-1)N} \\ & & & & 0 \end{pmatrix}, \tag{4.40}$$

where the first equality expresses that the matrix is symmetric, that is, equal to its own transpose. The following code presents the updating of the matrix when a particle is moved

```
1  void Sampling::update_pos(const Walker* walker_pre, Walker* walker_post, int particle)
       const {
2
3      ...
4
5      //Updating the part of the r_rel matrix which is changed by moving the [particle]
6      for (int j = 0; j < n_p; j++) {
7          if (j != particle) {
8              walker_post->r_rel(particle, j) = walker_post->r_rel(j, particle)
9                      = walker_post->calc_r_rel(particle, j);
10         }
11     }
12
13     ...
14
15 }
```

Listing 4.4: The code used to update the relative distance matrix of a walker when a particle is moved. In line 8, the symmetry of the matrix is exploited to further decrease the number of calls to the relative distance function in line 9.

Functions such as `Coulomb::get_potential_energy` and all of the Jastrow functions can then simply access these matrix elements without having to perform any explicit calculations.

A similar updating algorithm has been implemented for the squared distance vector and the absolute value of the distance vector.

4.6.2 The Slater related matrices

Apart from the inverse, whose optimization was covered in Section 4.4.2, calculating the single-particle wave functions and their gradients are the most expensive operations in the QMC algorithm.

Storing these function values in matrices representing the Slater matrices $\mathbf{S}^{\uparrow}$ and $\mathbf{S}^{\downarrow}$ introduced in Section 4.3, ensures that these values never become recalculated. The following expression describes how the Slater matrix is represented in the code

$$\mathbf{S} \equiv \text{join}\left(\mathbf{S}^{\uparrow},\ \mathbf{S}^{\downarrow}\right) = \begin{bmatrix} \phi_1(\mathbf{r}_0) & \phi_1(\mathbf{r}_1) & \cdots & \phi_1(\mathbf{r}_N) \\ \phi_2(\mathbf{r}_0) & \phi_2(\mathbf{r}_1) & \cdots & \phi_2(\mathbf{r}_N) \\ \vdots & \vdots & & \vdots \\ \phi_{N/2}(\mathbf{r}_0) & \phi_{N/2}(\mathbf{r}_1) & \cdots & \phi_{N/2}(\mathbf{r}_N) \end{bmatrix}, \tag{4.41}$$

where $\text{join}(\mathbf{A}, \mathbf{B})$ denotes joining the columns of the matrices $\mathbf{A}$ and $\mathbf{B}$, i.e. *concatenating* the matrices [36]. Similarly for the gradient terms, the following matrix is responsible for storing the gradient of all the elements in Eq. (4.41)

$$\mathbf{dS} \equiv \begin{bmatrix} \nabla\phi_1(\mathbf{r}_0) & \nabla\phi_1(\mathbf{r}_1) & \cdots & \nabla\phi_1(\mathbf{r}_N) \\ \nabla\phi_2(\mathbf{r}_0) & \nabla\phi_2(\mathbf{r}_1) & \cdots & \nabla\phi_2(\mathbf{r}_N) \\ \vdots & \vdots & & \vdots \\ \nabla\phi_{N/2}(\mathbf{r}_0) & \nabla\phi_{N/2}(\mathbf{r}_1) & \cdots & \nabla\phi_{N/2}(\mathbf{r}_N) \end{bmatrix}. \tag{4.42}$$

The inverse Slater matrices are implemented this way as well:

$$\mathbf{S}^{-1} \equiv \text{join}\left((\mathbf{S}^{\uparrow})^{-1}, (\mathbf{S}^{\downarrow})^{-1}\right). \tag{4.43}$$

In the code, these matrices are stored as `walker.phi`, `walker.del_phi` and `walker.inv`. When one particle is moved, only a single row in the two first matrices needs to be recalculated, and only half of the concatenated inverse needs to be updated.

This concatenation of matrices ensures that no conditional tests are needed in order to access the correct matrix for a given particle.

4.6.3 The Padé Jastrow gradient

Just as for the Jastrow Laplacian, there are symmetries in the expression for the gradient in Eq. (4.34), which implies the existence of an optimized way of calculating it. However, unlike the Laplacian, the gradient is split into components, which makes the exploitation of symmetries a little less straight-forward.

Defining

$$\mathrm{d}\mathbf{J}_{ik} \equiv \frac{a_{ik}}{r_{ik}} \frac{\mathbf{r}_i - \mathbf{r}_k}{(1 + \beta r_{ik})^2}, \tag{4.44}$$

it is apparent that

$$\mathrm{d}\mathbf{J}_{ik} = -\mathrm{d}\mathbf{J}_{ki}. \tag{4.45}$$

Since the gradient can be written in terms of this quantity, that is,

$$\frac{\nabla_i J}{J} = \sum_{k \neq i = 1}^{N} \mathrm{d}\mathbf{J}_{ik}, \tag{4.46}$$

the code can be optimized by exploiting this antisymmetry. Consider the following matrix

$$\mathrm{d}\mathbf{J} \equiv \begin{pmatrix} 0 & \mathrm{d}\mathbf{J}_{12} & \mathrm{d}\mathbf{J}_{13} & \cdots & \mathrm{d}\mathbf{J}_{1N} \\ -\mathrm{d}\mathbf{J}_{12} & 0 & \mathrm{d}\mathbf{J}_{23} & \cdots & \mathrm{d}\mathbf{J}_{2N} \\ & & \ddots & \ddots & \vdots \\ & (-) & & 0 & \mathrm{d}\mathbf{J}_{(N-1)N} \\ & & & & 0 \end{pmatrix}. \tag{4.47}$$

In the same way that the relative distance matrix was equal to its transpose, the matrix above is equal to the negative of its transpose. In other words:

$$\mathrm{d}\mathbf{J} = -\mathrm{d}\mathbf{J}^T. \tag{4.48}$$

Additionally, just as for the relative distances, moving a single-particle only changes one row and one column in the matrix. This implies that a similar updating algorithm as the one discussed in Section 4.6.1 can be implemented. This is demonstrated in the following code snippet:

```
1  void Pade_Jastrow::get_dJ_matrix(Walker* walker, int moved_particle) const {
2
3      int i = moved_particle;
4      for (int j = 0; j < n_p; j++) {
5          if (j == i) continue;
6
7          b_ij = 1.0 + beta * walker->r_rel(i, j);
8          factor = a(i, j) / (walker->r_rel(i, j) * b_ij * b_ij);
9          for (int k = 0; k < dim; k++) {
10             walker->dJ(i, j, k) = (walker->r(i, k) - walker->r(j, k)) * factor;
11             walker->dJ(j, i, k) = -walker->dJ(i, j, k);
12         }
13     }
14 }
```

Listing 4.5: The updating algorithm for the three-dimensional matrix used in the Padé Jastrow gradient. In line 11, the symmetry property is exploited by setting the transposed term equal to the negative of the already calculated term.

Calculating the new gradient is now only a matter of summing over the rows of the matrix in Eq. (4.47):

$$\frac{\nabla_i J^{\mathrm{new}}}{J^{\mathrm{new}}} = \sum_{k \neq i=1}^{N} \mathrm{d}\mathbf{J}^{\mathrm{new}}_{ik}. \tag{4.49}$$

Further optimization can be achieved by realizing that the function which calculates the new gradient also has access to the gradient of the previous iteration

$$\frac{\nabla_i J^{\mathrm{old}}}{J^{\mathrm{old}}} = \sum_{k \neq i=1}^{N} \mathrm{d}\mathbf{J}^{\mathrm{old}}_{ik}. \tag{4.50}$$

As mentioned previously, moving a particle, p, only a single row and column in $\mathrm{d}\mathbf{J}$. For all other particles $i \neq p$, only a single term from the new matrix is required to update the old gradient, that is

$$\frac{\nabla_{i \neq p} J^{\mathrm{new}}}{J^{\mathrm{new}}} = \sum_{k=1}^{N} \mathrm{d}\mathbf{J}^{\mathrm{new}}_{ik} = \left[\sum_{k \neq p} \mathrm{d}\mathbf{J}^{\mathrm{old}}_{ik} \right] + \mathrm{d}\mathbf{J}^{\mathrm{new}}_{ip}. \tag{4.51}$$

Adding and subtracting the term missing from the sum to make it equal to the old gradient from Eq. (4.50) gives

$$\frac{\nabla_{i\neq p} J^{\text{new}}}{J^{\text{new}}} = \left[\sum_{k\neq p} \mathrm{d}\mathbf{J}^{\text{old}}_{ik} + \mathrm{d}\mathbf{J}^{\text{old}}_{ip}\right] - \mathrm{d}\mathbf{J}^{\text{old}}_{ip} + \mathrm{d}\mathbf{J}^{\text{new}}_{ip} \tag{4.52}$$

$$= \frac{\nabla_i J^{\text{old}}}{J^{\text{old}}} - \mathrm{d}\mathbf{J}^{\text{old}}_{ip} + \mathrm{d}\mathbf{J}^{\text{new}}_{ip}, \tag{4.53}$$

which effectively reduces the calculation to two flops. For the case with $i = p$, the entire sum must be calculated as in Eq. (4.49). This process is demonstrated in the following code

```
1  void Pade_Jastrow::get_grad(const Walker* walker_pre, Walker* walker_post, int p) const {
2      double sum;
3
4      for (int i = 0; i < n_p; i++) {
5          if (i == p) {
6
7              //for i == p the entire sum needs to be calculated
8              for (int k = 0; k < dim; k++) {
9
10                 sum = 0;
11                 for (int j = 0; j < n_p; j++) {
12                     sum += walker_post->dJ(p, j, k);
13                 }
14
15                 walker_post->jast_grad(p, k) = sum;
16             }
17
18         } else {
19
20             //for i != p only one term differ from the old and the new matrix
21             for (int k = 0; k < dim; k++) {
22                 walker_post->jast_grad(i, k) = walker_pre->jast_grad(i, k)
23                         + walker_post->dJ(i, p, k) - walker_pre->dJ(i, p, k);
24             }
25         }
26     }
27 }
```

Listing 4.6: The implementation of the Padé Jastrow gradient using the matrix from Eq. (4.47). Lines 18-25 describe the case for gradients not equal to the moved particle, i.e. Eq. (4.53). Lines 5-18 describe the case for the gradient of the moved particle, where the full sum is calculated as in Eq. (4.49).

The dimensions of $\mathrm{d}\mathbf{J}$ are $N \times N \times d$, where N is the number of particles and d is the dimension. This implies that the optimizations in the Jastrow gradient discussed in this section scale very well with N. See Section 6.1 for a demonstration of the speedup in a $N = 30$, $d = 2$ case.

4.6.4 The single-particle Wave Functions

For systems of many particles, the function call `Orbitals::phi(walker, i, qnum)` needs to figure out which expression is related to which quantum number. The brute force implementation is to simply perform a test on the quantum number, and use this to return the corresponding expression. This is demonstrated in the following code:

```
double AlphaHarmonicOscillator::phi(const Walker* walker, int particle, int q_num) {

    //Ground state of the harmonic oscillator
    if (q_num == 0){
      return exp(-0.5*w*walker->get_r_i2(i));
    }

    ...

}
```

For a low number of particles this is quite efficient, however, this is not the case for a large number of particles.

A more efficient implementation is to represent the single-particle wave functions as `BasisFunctions` objects. These objects hold only one pure virtual member function `BasisFunctions::eval()` which takes on input the particle number i and the walker and returns the evaluated single-particle wave function. The object itself is defined by a quantum number q.

The following is an example of a two-dimensional harmonic oscillator single-particle wave function for quantum number $q = 1$

```
double HarmonicOscillator_1::eval(const Walker* walker, int i) {

    y = walker->r(i, 1);

    //y*exp(-k^2*r^2/2)

    H = y;
    return H*exp(-0.5*w*walker->get_r_i2(i));

}
```

These objects representing single-particle wave functions can be loaded into an array in such a way that element q corresponds to the `BasisFunctions` object representing this quantum number, e.g. `basis_functions[1] = new HarmonicOscillator_1()`. The new `Orbitals::phi` implementation then simply becomes a call to an array. This is demonstrated in the following code

```
double Orbitals::phi(const Walker* walker, int particle, int q_num) {
    return basis_functions[q_num]->eval(walker, particle);
}
```

Listing 4.7: The implementation of the single-particle basis used in the code. It is simply a call to an array holding all the different single-particle wave functions. The quantum number is used as an index, and the corresponding evaluation function is called with the supplied walker for the given particle.

All discussed optimizations thus far are general in the sense that they are independent of the explicit shape of the single-particle wave functions. There should, however, be room for optimizations within the basis functions themselves, as long as these are applied locally within each class where the explicit shape of the orbitals are absolute.

As discussed previously, only a single column in the Slater related matrices from Section 4.6.2 needs to be updated when a particle is moved. This implies that the terms which are independent of the quantum numbers can be calculated once for every particle instead of once for every quantum number and every particle.

These terms often come in the shape of exponential factors, which are conserved in derivatives, implying that these terms appear in both the gradients and the Laplacians as well as in the single-particle wave functions.

Looking at the harmonic oscillator - and the hydrogen-like wave functions listed in Appendix D - F, the exponential factors are indeed independent of the quantum numbers in all of the terms. Referring to the quantum number independent terms as $\overline{Q}_i$, the expressions are

$$\overline{Q}_i^{\text{H.O.}} = e^{-\frac{1}{2}\alpha\omega r_i^2} \tag{4.54}$$
$$\overline{Q}_i^{\text{Hyd.}} = e^{-\frac{1}{n}\alpha Z r_i} \tag{4.55}$$

The hydrogen eigenstates have a dependence on the principal quantum number n in the exponential, however, several expressions share this exponential factor. Calculating for example the $n = 2$ exponentials beforehand saves 19 exponential calls per particle every cycle, resulting in a dramatic speedup nevertheless.

The implementation is very simple; the virtual function `Orbitals::set_qnum_indie_terms` is called whenever a particle is moved, which updates the value of the `exp_factor` pointer shared by all the loaded `BasisFunctions` objects and the `Orbitals` class. The following code snippet presents implementations of the function in case of the harmonic oscillator - and the hydrogen-like basis

```
1  void AlphaHarmonicOscillator::set_qnum_indie_terms(const Walker * walker, int i) {
2
3     //k2 = alpha*omega
4     *exp_factor = exp(-0.5 * (*k2) * walker->get_r_i2(i));
5  }
6
7  void hydrogenicOrbitals::set_qnum_indie_terms(Walker* walker, int i) {
8
9      //waler::calc_r_i() calculates |r_i| such that walker::get_r_i() can be used
10     walker->calc_r_i(i);
11
12     //k = alpha*Z
13     double kr = -(*k) * walker->get_r_i(i);
14
15     //Calculates only the exponentials needed based on the number of particles
16     *exp_factor_n1 = exp(kr);
17     if (n_p > 2) *exp_factor_n2 = exp(kr / 2);
18     if (n_p > 10) *exp_factor_n3 = exp(kr / 3);
19     if (n_p > 28) *exp_factor_n4 = exp(kr / 4);
20
21 }
```

Listing 4.8: Implementation of the function handling the calculation of the quantum number independent terms. Lines 1-5 describe the harmonic oscillator case, where the exponential factor $\exp(-\alpha\omega r_i^2)$ is the independent factor. Lines 7-21 describe the hydrogen-like case, where the calculated exponential factor $\exp(-\alpha Z|r_i|/n)$ has a dependence on the principal quantum number n. One factor is thus calculated per n, however, if no particles occupy states with a given n, the corresponding factor is not calculated (see lines 17-19).

The `BasisFunctions` objects share the pointer to the correct exponential factor with the orbital class. These exponential factors can then simply be accessed instead of being recalculated numerous times. The following code demonstrates accessing the exponential factor calculated in the previous code example

```
1  double HarmonicOscillator_1::eval(const Walker* walker, int i) {
2
3      y = walker->r(i, 1);
4
5      //y*exp(-k^2*r^2/2)
6
7      H = y;
8      return H*(*exp_factor);
9
```

```
10 }
```

Listing 4.9: The implementation of a single-particle wave function. The pointer to the previously calculated exponential factor is simply accessed in line 8.

For two-dimensional quantum dots, 112 `BasisFunctions` objects are needed for a 56-particle simulation. Applying the currently discussed optimization reduces the number of exponential calls needed to calculate every wave function from 112 to 1, which for an average DMC calculation results in $6 \cdot 10^{11}$ saved exponential calls. Generally, the number of exponential calls are reduced by a factor $\frac{N}{2}(N+1)$, where N is the number of particles.

4.7 CPU Cache Optimization

The *CPU cache* is a limited amount of memory directly connected to the CPU, designed to reduce the average time to access memory. Simply speaking, standard memory is slower than the CPU cache, as bits have to travel through the motherboard before it can be fed to the CPU (a so called *bus*).

Which values are held in the CPU cache is controlled by the compiler, however, if programmed poorly, the compiler will not be able to handle the cache storage optimally. Optimization tools such as `O3` exist in order to work around this, however, keeping the cache in mind from the beginning of the coding process may result in a much faster code. In the case of the QMC code, the most optimal use of the cache would be to have all the active walkers in the cache at all times.

The memory is sent to the cache as arrays, which means that storing walker data sequentially in memory is the way to go in order to make take full use of the processor cache. If objects are declared as pointers, which is the case for matrices of general sizes, the memory layout is uncontrollable, that is, it is not given that matrices which are declared sequentially will end up sequentially in the memory. This fact renders a QMC solver for a general number of particles hard to optimize with respect to the cache.

5

Modelled Systems

The systems modelled in this thesis are exclusively systems which have closed form solutions when the electron-electron interaction is removed, i.e. in the non-interacting case. As discussed in Section 3.6, these closed form solutions are used to construct an optimal trial wave function in the form of a single Slater determinant. Without such an optimal basis, a single Slater determinant is not sufficient in Quantum Monte-Carlo (QMC) simulations. It is therefore not random that the focus of this thesis has been on various kinds of *quantum dots* and *atomic systems*, resembling the analytically solvable harmonic oscillator and the hydrogen atom, respectively.

In this chapter atomic units will be used, that is, $\hbar = e = m_e = 4\pi\epsilon_0 = 1$, where m_e and e_0 is the electron mass and vacuum permittivity, respectively.

5.1 Atomic Systems

Atomic systems are described as a number of electrons surrounding oppositely charged nuclei. As an approximation, the position of the nucleus is fixed. Due to the fact that the mass of the core is several orders of magnitude larger than the mass of the electrons, this serves as a very good approximation. In literature this is referred to as the *Born-Oppenheimer Approximation* [27].

5.1.1 The Single-particle Basis

The single-particle basis used to construct the trial wave functions for atomic systems originates from the closed form solutions for the hydrogen atom, i.e. one electron surrounding a single nucleus.

Given a nucleus with charge Z, the external potential between the electron and the core is

$$\widehat{\mathbf{v}}_{\text{ext}}(\mathbf{r}) = -\frac{Z}{r}, \tag{5.1}$$

which results in the following single-particle Hamiltonian:

$$\widehat{\mathbf{h}}_0(\mathbf{r}) = -\frac{1}{2}\nabla^2 - \frac{Z}{r}. \tag{5.2}$$

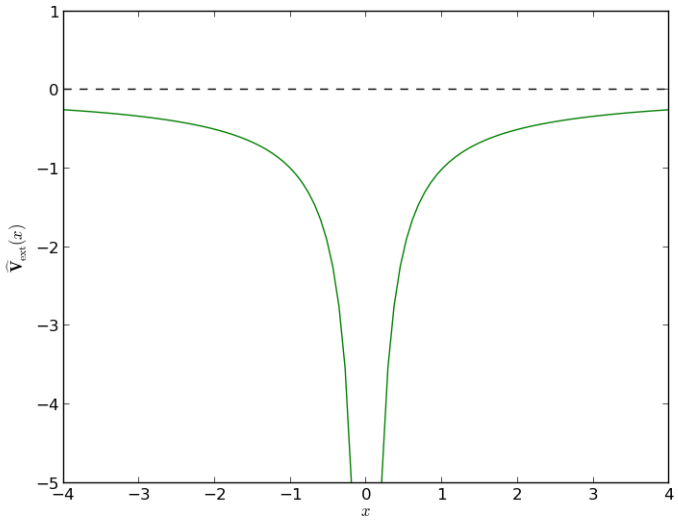


Figure 5.1: The one-dimensional version of the single-particle potential of hydrogen from Eq. (5.1). The potential is spherically symmetric for three dimensions and can be visualized by rotating the figure around all axes. The potential has a strong singularity at the origin, which originates from the fact that the nucleus (located at the origin) and the electrons have opposite charge, that is, they feel an attractive force.

The external potential is displayed in Figure 5.1. The strong singularity in the center is a result of the strong attraction between the electron and the oppositely charged proton. The eigenfunctions of the Hamiltonian are [17]

$$\phi_{nlm}(r,\theta,\phi;Z) \propto r^l e^{-Zr/n}\left[L_{n-l-1}^{2l+1}\left(\frac{2r}{n}Z\right)\right]Y_l^m(\theta,\phi), \tag{5.3}$$

where $L_{q-p}^{p}(x)$ are the *associated Laguerre polynomials* and $Y_l^m(\theta,\phi)$ are the *spherical harmonics*. The spherical harmonics are related to the *associated Legendre functions* P_l^m in the following manner:

$$Y_l^m(\theta,\phi) \propto P_l^m(\cos\theta)e^{im\phi}, \tag{5.4}$$

In the current model, the principal quantum number n together with Z control the energy level of the atom,

$$E(n;Z) = -\frac{Z^2}{2n^2}, \tag{5.5}$$

which implies that the energy levels are degenerate for all combinations of l and m. For a given value of n, the allowed levels of the *azimuthal* quantum number l and the *magnetic* quantum number m are

$$\begin{aligned} n &= 1, 2, ... \\ l &= 0, 1, ..., n-1. \\ m &= -l, -l+1, ..., l-1, l. \end{aligned}$$

Different values for l and m for a given n define the shell structure of the hydrogen atom.

A problem with the single-particle basis of hydrogen is the complex terms in Eq. (5.4), i.e. the spherical harmonics. The introduction of the *solid harmonics* $S_l^m(r,\theta,\phi)$ allows for using real-valued eigenfunctions in the QMC simulations. The solid harmonics are related to the spherical harmonics through [37]

$$S_l^m(r,\theta,\phi) \propto r^l \left[Y_l^m(\theta,\phi) + (-1)^m Y_l^{-m}(\theta,\phi)\right] \tag{5.6}$$

$$\propto r^l P_l^{|m|}(\cos\theta) \begin{cases} \cos m\phi & m \geq 0 \\ \sin |m|\phi & m < 0 \end{cases}, \tag{5.7}$$

which results in the following real eigenfunctions

$$\phi_{nlm}(r,\theta,\phi;k) \propto e^{-kr/n} \left[L_{n-l-1}^{2l+1}\left(\frac{2r}{n}k\right)\right] S_l^m(r,\theta,\phi) \equiv \phi_{nlm}^{\mathrm{H}}(\mathbf{r}), \tag{5.8}$$

where $k = \alpha Z$ is a scaled charge with α as a variational parameter chosen by the methods described in Section 3.6.3.

One should also keep in mind that the hydrogen-like wave functions yield only bound states. For atoms where the least bound electrons are weakly bound, a coupling to states in the continuum may be necessary. In quantum chemistry calculations, so-called *Slater orbitals* are used routinely in order to account for this deficiency [38]. In this thesis however, the aim has been to use the given basis without tailoring it to a specific system. This opens up the possibility to test the reliability of the basis without any fine tuning.

A set of quantum numbers nlm is mapped to a single index i. A listing of all the single-particle wave functions and their closed form derivatives are given in Appendix F.

5.1.2 Atoms

An atom is described as N electrons surrounding a fixed nucleus of charge $Z = N$. The Hamiltonian consists of N single-particle hydrogen Hamiltonians in addition to the Coulomb interaction, which results in

$$\widehat{\mathbf{H}}_{\mathrm{Atoms}}(\mathbf{r}) = \sum_{i=1}^{N} \widehat{\mathbf{h}}_0(\mathbf{r}_i) + \sum_{i<j} \frac{1}{r_{ij}} \tag{5.9}$$

$$= \sum_{i=1}^{N} \left[-\frac{1}{2}\nabla_i^2 - \frac{Z}{r_i}\right] + \sum_{i<j} \frac{1}{r_{ij}}, \tag{5.10}$$

where $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$. Excluding the Coulomb term, the Hamiltonian can be decoupled into single-particle terms with a total ground state energy

$$E_0 = -\frac{Z^2}{2} \sum_{i=1}^{N} \frac{1}{n_i^2}. \tag{5.11}$$

The Slater determinant is set up to fill the N lowest lying states, that is, the N states with lowest n without breaking the Pauli principle, using the single-particle orbitals from Eq. (5.8). The degeneracy of level n in an atom is given by the following expression:

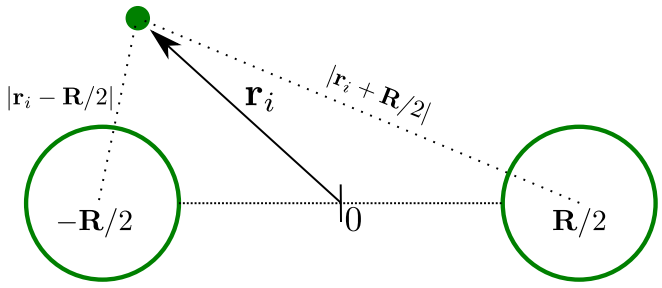


Figure 5.2: The model for the diatomic molecule used in this thesis. The two largest circles represent the atoms. An electron at position $\mathbf{r}_i$ gets a potential energy contribution from both the cores equal to $Z/|\mathbf{r}_i+\mathbf{R}/2|$ and $Z/|\mathbf{r}_i-\mathbf{R}/2|$, where Z is the charge of the nuclei (homonuclear). The diatomic molecular wave functions are set up as a superposition of two hydrogen-like wave functions, one at position $\mathbf{r}_i+\mathbf{R}/2$ and the second at position $\mathbf{r}_i - \mathbf{R}/2$.

$$g(n) = 2\sum_{l=0}^{n-1}\sum_{m=-l}^{l} 1 = 2n^2. \tag{5.12}$$

5.1.3 Homonuclear Diatomic Molecules

A homonuclear diatomic molecule consists of two atoms (diatomic molecule) of the same element (homonuclear) with charge $Z = N/2$, separated by a distance R. The origin is set between the atoms, which are fixed at positions $\pm\mathbf{R}/2$. An electron at position $\mathbf{r}_i$ gets a contribution from both the cores as displayed in Figure 5.2. In addition, there is a repulsive Coulomb potential between the two cores equal to Z^2/R, from here on referred to as the atomic nucleus potential. The resulting Hamiltonian becomes

$$\widehat{\mathbf{H}}_{\text{Mol.}}(\mathbf{r},\mathbf{R}) = \sum_{i=1}^{N}\left[-\frac{1}{2}\nabla_i^2 + \frac{Z}{|\mathbf{r}_i+\mathbf{R}/2|} + \frac{Z}{|\mathbf{r}_i-\mathbf{R}/2|}\right] + \frac{Z^2}{R} + \sum_{i<j}\frac{1}{r_{ij}}. \tag{5.13}$$

In order to transform the hydrogen eigenstates $\phi^{\text{H}}_{nlm}(\mathbf{r})$, which are symmetric around a single nucleus, into molecular single-particle states $\phi^{\pm}_{nlm}(\mathbf{r}_i)$, a superposition of the two mono-nucleus wave functions are used:

$$\phi^{+}_{nlm}(\mathbf{r}_i,\mathbf{R}) = \phi^{\text{H}}_{nlm}(\mathbf{r}_i+\mathbf{R}/2) + \phi^{\text{H}}_{nlm}(\mathbf{r}_i-\mathbf{R}/2), \tag{5.14}$$
$$\phi^{-}_{nlm}(\mathbf{r}_i,\mathbf{R}) = \phi^{\text{H}}_{nlm}(\mathbf{r}_i+\mathbf{R}/2) - \phi^{\text{H}}_{nlm}(\mathbf{r}_i-\mathbf{R}/2), \tag{5.15}$$

which reads "electron surrounding first nucleus combined with electron surrounding second nucleus". From Figure 5.2 it is easy to see that the two function arguments represent the electron position in the reference frames of the two nuclei.

As seen from the equations above, there are necessarily two ways of doing this superposition: Adding and subtracting the states. It is easy to show that

$$\left\langle \phi^{-}_{n'l'm'} | \phi^{+}_{nlm} \right\rangle = 0, \tag{5.16}$$

which implies that these states form an expanded complete set of single-particle states for the molecular system, resulting in a four-fold degeneracy in each set of quantum numbers nlm. It is necessary to use both the positive and negative states in order to fit e.g. four electrons into $n = 0$ for the case of the lithium molecule ($N = 6$). Using only the positive or only the negative states would result in a singular Slater determinant.

Using $\mathbf{R} = (R_x, R_y, R_z)$ as the vector separating the atoms, $\mathbf{j} = (0, 1, 0)$ as the unit vector in the y-direction, $\mathbf{r}_i = (x_i, y_i, z_i)$ as the electron position, and the chain rule of derivation, the gradient in the $\mathbf{j}$-direction becomes

$$\begin{aligned}
\mathbf{j} \cdot \nabla_i \phi^{\pm}_{nlm}(\mathbf{r}_i, \mathbf{R}) &= \underbrace{\frac{\partial (y_i + R_y/2)}{\partial y_i}}_{1} \frac{\partial \phi^{\mathrm{H}}_{nlm}(\mathbf{r}_i + \mathbf{R}/2)}{\partial (y_i + R_y/2)} \\
&\pm \underbrace{\frac{\partial (y_i - R_y/2)}{\partial y_i}}_{1} \frac{\partial \phi^{\mathrm{H}}_{nlm}(\mathbf{r}_i - \mathbf{R}/2)}{\partial (y_i - R_y/2)} \\
&= \frac{\partial \phi^{\mathrm{H}}_{nlm}(\mathbf{r}_i + \mathbf{R}/2)}{\partial (y_i + R_y/2)} \pm \frac{\partial \phi^{\mathrm{H}}_{nlm}(\mathbf{r}_i - \mathbf{R}/2)}{\partial (y_i - R_y/2)} \\
&= \frac{\partial \phi^{\mathrm{H}}_{nlm}(\tilde{\mathbf{R}}^{+}_{\mathbf{i}})}{\partial \tilde{Y}^{+}_{i}} \pm \frac{\partial \phi^{\mathrm{H}}_{nlm}(\tilde{\mathbf{R}}^{-}_{\mathbf{i}})}{\partial \tilde{Y}^{-}_{i}},
\end{aligned} \tag{5.17}$$

where $\tilde{\mathbf{R}}^{\pm}_{\mathbf{i}} = \mathbf{r}_i \pm \mathbf{R}/2 = (\tilde{X}^{\pm}_i, \tilde{Y}^{\pm}_i, \tilde{Z}^{\pm}_i)$ represents the electron position in the reference frame of the two nuclei. Equation (5.17) demonstrates that the closed form expressions used in simulations of single atoms can be reused in the case of diatomic molecules. In other words, the functions in Appendix F can simply be called with $\tilde{\mathbf{R}}^{\pm}_{\mathbf{i}}$ instead of $\mathbf{r}_i$ and then be either subtracted or added. This result holds for the Laplacian as well.

The non-interacting energy is equal to that of the regular atoms in the limit $R \to \infty$, however, now with a four-fold degeneracy and a charge equal to $N/2$. This four-foulding also implies that the degeneracy of level n becomes $g(n) = 4n^2$.

5.2 Quantum Dots

Quantum dots are electrons confined within a potential well. This potential well can be broadened and narrowed in such a way that the material properties of the quantum dot can be tuned to desired values. Such manufactured quantum dots have practical applications in for example solar cells [39], lasers [40], medical imaging [41], and quantum computing [42], however, the focus on quantum dots in this thesis has been purely academic.

Understanding the physics behind strongly and weakly confined electrons are of great importance when it comes to understanding many-body theory in general. The purpose of studying quantum dots from an academic point of view is to investigate the behavior of the system a function of the level of confinement and the number of electrons.

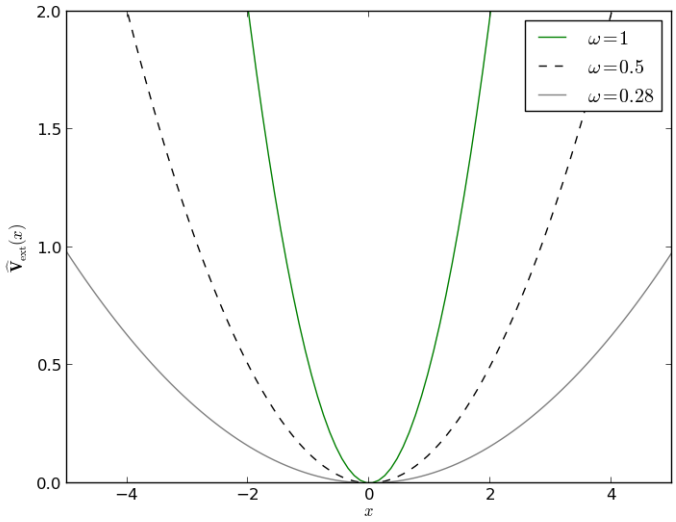


Figure 5.3: A one-dimensional version of the single-particle potential of quantum dots. In two - and three dimensions, the potential is rotationally/spherically symmetric and equal along all axes. Just as the hydrogen potential in Figure 5.1, the electrons are drawn to the center.

The model for the quantum dot used in this thesis is electrons trapped in a harmonic potential with frequency ω, which in the case of no electron-electron interaction can be solved analytically.

5.2.1 The Single Particle Basis

Just as the hydrogen potential was used to describe atoms, the *harmonic oscillator* potential is used to describe quantum dots. The potential is on the form

$$\widehat{\mathbf{v}}_{\text{ext}}(\mathbf{r}) = \frac{1}{2}\omega^2 r^2, \tag{5.18}$$

where ω is the oscillator frequency representing the strength of the confinement. The potential for different ω is presented in Figure 5.3. As for atoms, the potential has its minimum in the center, however, with no singularity. Note also that no matter how low the frequency becomes, the electrons will always be bound as long as the frequency is non-zero. The single-particle Hamiltonian is

$$\widehat{\mathbf{h}}_0(\mathbf{r}) = -\frac{1}{2}\nabla^2 + \frac{1}{2}\omega^2 r^2. \tag{5.19}$$

The eigenfunctions of the Hamiltonian for two and three dimensions are [27]

$$\phi_{n_x,n_y}(\mathbf{r}) = H_{n_x}(\sqrt{w}x)H_{n_y}(\sqrt{w}y)e^{-\frac{1}{2}wr^2} \tag{5.20}$$

$$\phi_{n_x,n_y,n_z}(\mathbf{r}) = H_{n_x}(\sqrt{w}x)H_{n_y}(\sqrt{w}y)H_{n_z}(\sqrt{w}z)e^{-\frac{1}{2}wr^2}, \tag{5.21}$$

where $H_n(x)$ is the n'th level Hermite polynomial. The shell structure of a quantum dot arises from different combinations of n_x, n_y, and for three dimensions n_z, which yield the same total n.

The variational parameter α is introduced by letting $\omega \to \alpha\omega$, just as $Z \to \alpha Z$ for atoms. Defining $k \equiv \sqrt{\alpha\omega}$, the eigenfunctions which are used as the single-particle orbitals for quantum dots in this thesis are

$$\phi_{n_x,n_y}(\mathbf{r}) = H_{n_x}(kx)H_{n_y}(ky)e^{-\frac{1}{2}k^2r^2}, \tag{5.22}$$

$$\phi_{n_x,n_y,n_z}(\mathbf{r}) = H_{n_x}(kx)H_{n_y}(ky)H_{n_z}(kz)e^{-\frac{1}{2}k^2r^2}. \tag{5.23}$$

As for the hydrogen states, a set of quantum numbers are mapped to an integer i. A listing of all the single-particle wave functions and their closed form derivatives are given in Appendix D for two dimensions and Appendix E for three dimensions.

5.2.2 Two - and Three-dimensional Quantum Dots

The quantum dot used in this thesis is described as N interacting electrons confined in an oscillator potential with frequency ω. The Hamiltonian is

$$\widehat{\mathbf{H}}_{\text{Q.D.}}(\mathbf{r}) = \sum_{i=1}^{N} \widehat{\mathbf{h}}_0(\mathbf{r}_i) + \sum_{i<j} \frac{1}{r_{ij}} \tag{5.24}$$

$$= \sum_{i=1}^{N} \left[-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2 r_i^2 \right] + \sum_{i<j} \frac{1}{r_{ij}}, \tag{5.25}$$

where $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$. Excluding the Coulomb term, the Hamiltonian can be decoupled into single-particle terms with a total ground state energy [17]

$$E_0 = \omega \sum_{i=1}^{N} \left(n_i + \frac{d}{2} \right), \tag{5.26}$$

where d is the number of dimensions, $n_i = n_x + n_y + n_z \geq 0$ for three dimensions and $n_i = n_x + n_y \geq 0$ for two dimensions. The degeneracy of level n in a quantum dot is $g(n) = 2n$ for two dimensions and $g(n) = (n+2)(n+1)$ for three dimensions.

The Slater determinant is set up to fill the N lowest lying states, that is, the N states with lowest n without breaking the Pauli principle, using the single-particle orbitals from Eq. (5.8).

5.2.3 Double-well Quantum Dots

The same strategy used to transform an atomic system into a homonuclear diatomic molecular system can be applied to two-dimensional quantum dots, resulting in a double-well quantum dot. Double-well quantum dots are used as a practical quantum dot system in experiments [43].

The model for the double-well potential used in this thesis is [44]

$$\widehat{\mathbf{v}}_{ext}(\mathbf{r}) = \frac{1}{2}m^*\omega_0^2 \left[r^2 + \frac{1}{4}R^2 - R|x| \right], \tag{5.27}$$

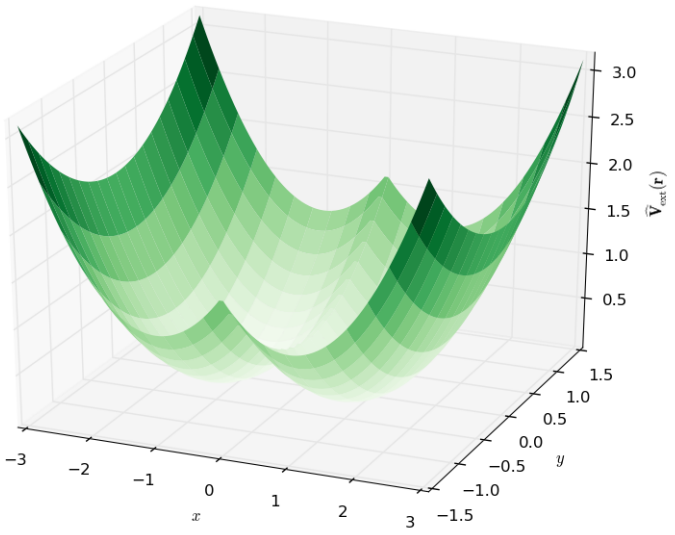


Figure 5.4: The external potential for a double-well quantum dot from Eq. (5.27) with $R = 2$ and $m^*\omega_0^2 = 1$, where R is the distance between the well centers and m^* and ω_0 are material constants.

where R is the distance between the wells, m^* is a material parameter and ω_0 is the confinement strength. For simplicity, the wells are separated in the x-direction. The potential is presented in Figure 5.4.

The full Hamiltonian becomes

$$\widehat{\mathbf{H}}_{\text{QDDW}}(\mathbf{r},\mathbf{R}) = \sum_{i=1}^{N}\left(-\frac{1}{2m^*}\nabla_i^2 + \frac{1}{2}m^*\omega_0^2\left[r^2 + \frac{1}{4}R^2 - R|x_i|\right]\right) + \sum_{i<j}\frac{1}{r_{ij}}, \tag{5.28}$$

which can be simplified to fit the standard form of the previous Hamiltonians by letting $r_i \to \sqrt{m^*}r_i$. Applying this transformation of coordinates yields

$$\widehat{\mathbf{H}}_{\text{QDDW}}(\mathbf{r},\mathbf{R}) \to \widehat{\mathbf{H}}_{\text{QDDW}}(\sqrt{m^*}\mathbf{r},\sqrt{m^*}\mathbf{R}) \tag{5.29}$$

$$= \sum_{i=1}^{N}\left(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega_0^2\left[r^2 + \frac{1}{4}R^2 - R|x_i|\right]\right) + \sqrt{m^*}\sum_{i<j}\frac{1}{r_{ij}}. \tag{5.30}$$

The eigenstates are, as for the homonuclear diatomic molecules in Eq. (5.14) and (5.15), given as positive and negative superpositions of the standard harmonic oscillator eigenstates. As shown for molecules in Eq. (5.17), the closed form expressions for the single-well quantum dot can be reused in the case of a double-well quantum dot.

The degeneracy of the n'th level is $g(n) = 4n$. The non-interacting single-particle energies are identical to those of the single-well in the limit $R \to \infty$, that is, in the case of two decoupled potential wells.

Part II

Results

6

Results

The results were produced using atomic units, i.e $\hbar = e = m_e = 4\pi\epsilon_0 = 1$, where m_e and e_0 is the electron mass and vacuum permittivity, respectively. This implies that all listed energies are given in Hartrees, i.e. scaled with $\hbar^2/m_e a_0^2$, and all lengths are given in Bohr radii, i.e. scaled with $a_0 = 4\pi\epsilon_0\hbar^2/m_e e^2$.

6.1 Optimization Results

The optimization results listed in this section are estimated using a 30-particle two-dimensional quantum dot as reference system.

Profiling the code revealed that $\sim 99\%$ of the run time was spent diffusing the particles, that is, spent in the function `QMC::diffuse_walker`. Variational Monte-Carlo (VMC), Diffusion Monte-Carlo (DMC), and Adaptive Stochastic Gradient Descent (ASGD) all rely heavily on the diffusion of particles, hence the parts of the code which do not involve diffusion walkers were neglected in the optimization process; it is a waste of time to optimize something which accounts for less than one percent of the run time.

The profiling tool of choice was *KCacheGrind*, which is available for free at the Ubuntu Software Center. KCacheGrind lists relative time spent in functions graphically in blocks whose sizes are proportional to the time spent inside the functions, much like standard hard drive listing software does with files and file sizes.

Optimizations discussed in Chapter 4 which are not mentioned in the following sections were considered standard implementations, and were thus implemented prior to the optimization process.

Storing the Slater matrix

This optimization is described in detail in Section 4.6.2. In addition to storing the Slater matrix, the calculation of $\tilde{\mathbf{I}}$ from the inverse updating algorithm in Eq. (4.22) was taken outside of the main loops.

The percentages listed in the following table represent the ratio between the total time spent inside the given function and the total run time.

`Orbitals::phi`	
Relative run time used prior to optimization	80.88%
Relative run time used after optimization	8.2%
Relative function speedup	9.86

The speedup comes not as a result of optimizations within the function itself, but rather as a result of far less calls to the function. If $\tilde{\mathbf{I}}$ was calculated outside of the main loops in the first place, the speedup would be far less significant.

Optimized Jastrow gradient

The optimization described in this section is discussed in detail in Section 4.6.3.

The percentages listed in the following table represent the ratio between the total time spent inside the given functions and the total run time.

`Jastrow::get_grad` & `Jastrow::calc_dJ`	
Relative run time used prior to optimization	40%
Relative run time used after optimization	5.24%
Relative function speedup	7.63

Exploiting the symmetries of the Padé Jastrow gradient, in addition to calculating the new gradient based on the old, are in other words extremely efficient. Keep in mind that these results are for a high number of particles. For two particles, this optimization would not matter at all.

Storing the orbital derivatives

This optimization is covered in detail in Section 4.6.2. Much like for the Slater matrix, the optimization in this case comes from the fact that the function itself is called fewer times, rather than being faster.

The percentages listed in the following table represent the ratio between the total time spent inside the given function and the total run time.

`Orbitals.dell_phi`	
Relative run time used prior to optimization	56.27%
Relative run time used after optimization	7.31%
Relative function speedup	7.70

Storing quantum number independent terms

This optimization is covered in detail in Section 4.6.4. The result of the optimization is a reduction in the number of exponential function calls, which means a more efficient calculation of single-paticle states, their gradients and Laplacians.

The percentages listed in the following table represent the ratio between the total time spent inside the given functions and the total run time.

`Orbitals::phi` & `Orbitals::dell_phi`	
Relative run time used prior to optimization	5.85%
Relative run time used after optimization	0.13%
Relative function speedup	45

This result is not surprisingly equal to $15 \cdot 3$, since a 30-particle quantum dot has 15 unique quantum numbers. One set is used by the orbitals, and two by their gradients (the Laplacian is not a part of the diffusion). Prior to this optimization, 45 exponential calls were needed to fill a row in the Slater matrix and the derivative matrix; this has been reduced to one.

Overall optimization and final scaling

Combining all the optimizations listed in this section, the final run time was reduced to 5% of the original. The final scaling is presented in Figure 6.1.

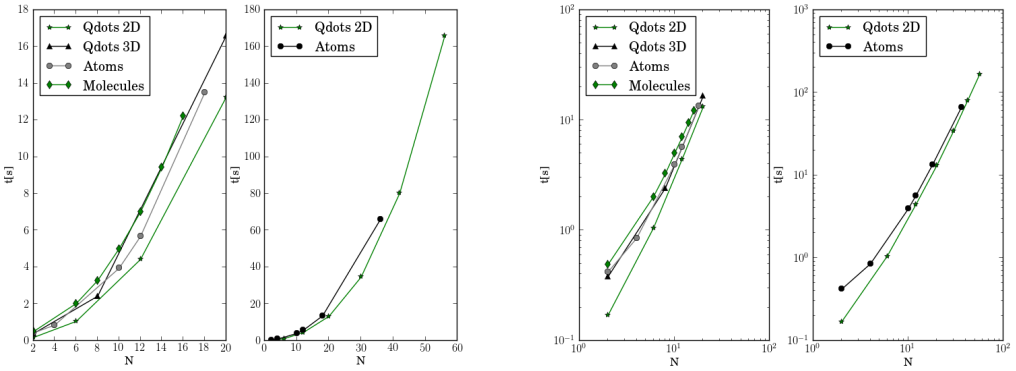


Figure 6.1: Scaling of the code with respect to the number of particles N based on VMC calculations with 10^6 cycles with 10^5 thermalization steps run on eight processors. The figures are split into a low N region and a high N region. Only two-dimensional quantum dots and atoms are displayed in the high N figure. The figures to the right contain the same data as the figures to the left, however, displayed using logarithmic axes. As expected, the two-dimensional quantum dots (denoted Qdots 2D) are lowest on run time and the homonuclear diatomic molecules are highest (denoted Molecules). The logarithmic figures clearly show a linear trend, implying a underlying power law.

The following power laws are deduced based on linear regression of the above figures for $N > 2$

System	Scaling
Two dimensional quantum dots	$N^{2.1038}$
Three dimensional quantum dots	$N^{2.1008}$
Atoms	$N^{1.8119}$
Homonuclear diatomic molecules	$N^{1.8437}$

As the number of particles N increases, the scaling with respect to the number of spatial dimensions d becomes negligible compared to the scaling with N, rendering two-dimensional quantum dots and atoms similar in run time. This is expected since there are far more matrices in the code of dimensions $N \times N$ than $N \times d$.

The Jastrow factor, inverse updating, etc., all involve the same computations for all systems, hence the reason why the atomic systems scale better than the quantum dots has to originate from the efficiency of the single-paticle wave functions. Consider for example the third single-paticle wave function for the hydrogen-like orbitals (omitting exponential factors):

$$\phi_3 = x. \tag{6.1}$$

The corresponding expression for a two-dimensional quantum dot is

$$\phi_3 = 2k^2y^2 - 1. \tag{6.2}$$

It is obvious that the orbital for quantum dots contains a higher computational cost for the processor than the one for atoms. Comparing the expressions listed for quantum dots in Appendix E and Appendix D with those for atoms in Appendix F, it is apparent that this trend is consistent.

The fact that the difference in the cost of the single-paticle wave functions govern the scaling demonstrates the efficiency in the general framework. Moreover, having the molecular system scaling almost identically to the atomic one demonstrates the efficiency of the system's implementation.

Both Variational Monte-Carlo and Adaptive Stochastic Gradient Descent scale linearly with the number of processors, due to the fact that the processes do not require any communication besides adding the final results. Diffusion Monte-Carlo, on the other hand, is parallelized by spreading the original walker population across multiple nodes. Depending on whether some nodes have more deaths or newborns than others, there is a high communication cost. What is seen in practice, however, is that as long as the average number of walkers per node does not go below ~ 250, the scaling is approximately linear.

6.2 The Non-interacting Case

In the case of non-interacting particles, that is, the case with no electron-electron interaction, the trial wave function represents the exact wave function both in case of quantum dots and atoms. For molecules and the double-well quantum dot, the additional requirement that $R \to \infty$ should also be applied, where R is the distance between the atoms in the case of molecules, and the distance between the well centers in the case of the quantum dot. All of the presented systems are covered in detail in Chapter 5.

Exact solutions serve as powerful guides, since results can be benchmarked against these, that is, the code can be validated. In the non-interacting case, Adaptive Stochastic Gradient Descent (ASGD) should always provide a variational parameter equal to unity, i.e. $\alpha = 1$. Variational Monte-Carlo (VMC) and Diffusion Monte-Carlo (DMC) should reproduce the exact solutions from Eq. (5.26) in the case of quantum dots and Eq. (5.11) in the case of atomic systems to machine precision.

In Table 6.1, validation runs for the three lowest lying closed-shell quantum dots are run for both two and three dimensions. Figure 6.2 shows ASGD finding the exact minimum. Table 6.3 shows similar results for atoms. As required, the closed form energies are reproduced to machine precision.

The double-well quantum dots results reproduce the non-interactive energies when the wells are placed far enough apart. This is demonstrated in Table 6.2. A separation equal to $R = 20$ was sufficient. For molecules, on the other hand, the atomic nuclei interactions are very strong, implying the need for a greater separation of the atoms than what was needed for the wells. Table 6.4 shows this effect; the convergence is nice for H_2, however, for the heavier molecules, where the atomic nuclei interaction is stronger, the convergence to the non-interacting limit is slower.

DMC should in the case of an exact wave function be perfectly stable. The trial energy should equal the ground state energy through all time steps and zero fluctuations in the number of walkers should occur. This trend is shown for the neon atom in Figure 6.3.

A final non-interacting case is run for DMC without the exact wave function. As discussed in Chapter 3, DMC should result in a better estimate of the ground state energy than VMC in the cases where the trial wave function does not equal the exact ground state. A test case demonstrating this is presented in Figure 6.4.

	2D					3D				
ω	N	E_{VMC}	E_{DMC}	α	E_0	N	E_{VMC}	E_{DMC}	α	E_0
0.5	2	1.0	1.0	1.0	1	2	3.0	3.0	1.0	3
1.0		2.0	2.0	1.0	2		1.5	1.5	1.0	1.5
0.5	6	5.0	5.0	1.0	5	8	18.0	18.0	1.0	18
1.0		10.0	10.0	1.0	10		9.0	9.0	1.0	9
0.5	12	14.0	14.0	1.0	14	20	60.0	60.0	1.0	60
1.0		28.0	28.0	1.0	28		30.0	30.0	1.0	30

Table 6.1: Validation results for N-particle quantum dots with no electron-electron interaction and frequency ω. The left-hand side shows the results for two dimensions, while the results for three dimensions are listed on the right-hand side. The last column for each dimension lists the exact energies E_0 calculated from Eq. (5.26). The exact solution to α is unity. As required, all methods reproduce the exact results. The variance is zero to machine precision for all listed results.

ω	N	E_{VMC}	E_{DMC}	α	$E_0(R \to \infty)$
0.5		4.0	4.0	1.0	4
1	4	2.0	2.0	1.0	2
0.5		20.0	20.0	1.0	20
1	12	10.0	10.0	1.0	10
0.5		28.0	28.0	1.0	28
1	24	56.0	56.0	1.0	56

Table 6.2: Validation results for N-particle double-well quantum dots with no electron-electron interaction and frequency ω. The exact energy E_0, calculated from Eq. (5.26), is listed in the last column. The calculations were run with the wells separated at a distance $R = 20$ in the x-direction. The exact solution to α is unity. As for the single-well quantum dots in Table 6.1, all methods reproduce the exact solutions. The variance is zero to machine precision for all listed results.

Atom	N	E_{VMC}	E_{DMC}	α	E_0
He	2	-4.0	-4.0	1.0	-4
Be	4	-20.0	-20.0	1.0	-20
Ne	10	-200.0	-200.0	1.0	-200

Table 6.3: Validation results for different atoms consisting of N electrons with no electron-electron interaction. The exact energies E_0 are calculated from Eq. (5.11). The exact solution of the variational parameter α is unity. As required, all methods reproduce the exact solutions. The variance is zero to machine precision for all listed results.

Molecule	N	R	E_{VMC}	E_{DMC}	$E_0(R \to \infty)$
H_2	2	10	-0.847	-0.9968	-1
		100	-0.979	-0.995	
		325	-1.000	-1.000	
Be_2	8	10	-41.596	-41.608	-40
		100	-40.298	-40.231	
		325	-40.123	-40.112	
Ne_2	20	10	-409.999	-410.010	-400
		100	-401.390	-401.049	
		325	-	-	

Table 6.4: Validation results for homonuclear diatomic molecules separated at a distance R with no electron-electron interaction. The last column lists the exact energies E_0 calculated from Eq. (5.11) for $R \to \infty$. Choosing R too high results in a singular Slater determinant due to finite machine precision. This happens already for $R = 325$ in the case of Ne_2. It is apparent that increasing R brings the solutions closer to the exact energy. The statistical errors are skipped.

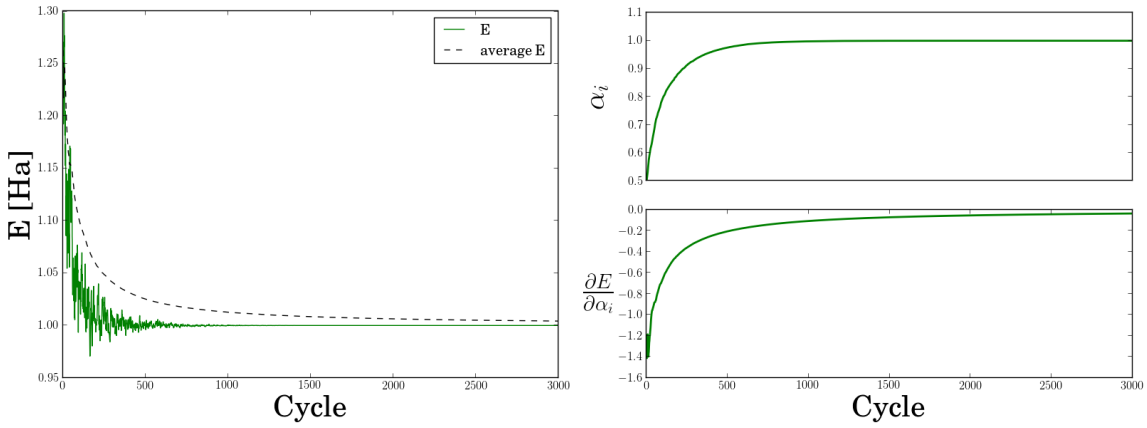


Figure 6.2: Adaptive Stochastic Gradient Descent (ASGD) results for a two-particle two-dimensional quantum dot with frequency $\omega = 0.5$ and no electron-electron interaction. The exact energy $E_0 = 1$ is reached after approximately 1000 cycles, where the variational parameter α has converged close to unity. Due to enormous fluctuations, the variational derivative is plotted as an accumulated average. The gradient is approximately zero after ~ 1000 cycles, which is in agreement with the behavior of the energy. The variational principle described in Section 3.6.3 is governing the trend of the energy convergence, however, a lot of statistical noise is present in the first 1000 cycles due to a high variance and a small number of samples.

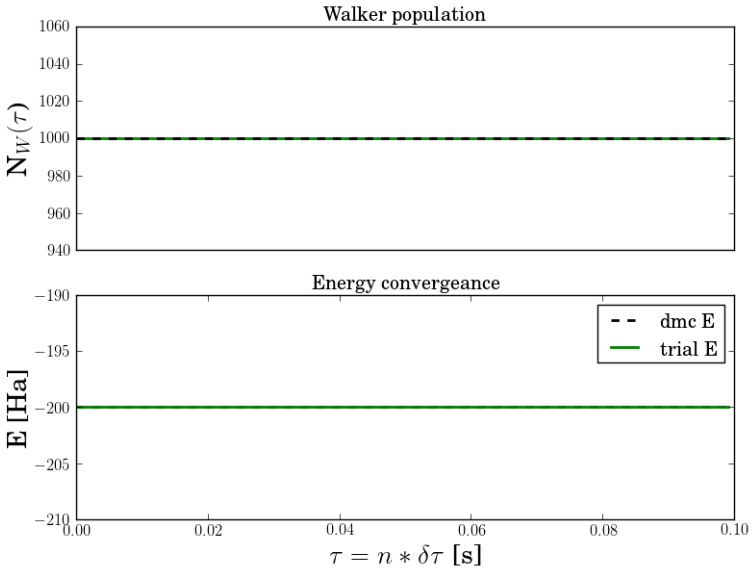


Figure 6.3: Illustration of the Diffusion Monte-Carlo (DMC) energy convergence for the neon atom with no electron-electron interaction listed in Table 6.3. The trial energy is fixed at the exact ground state energy as required. The number of walkers are constant, implying an approximately zero variance in the samples.

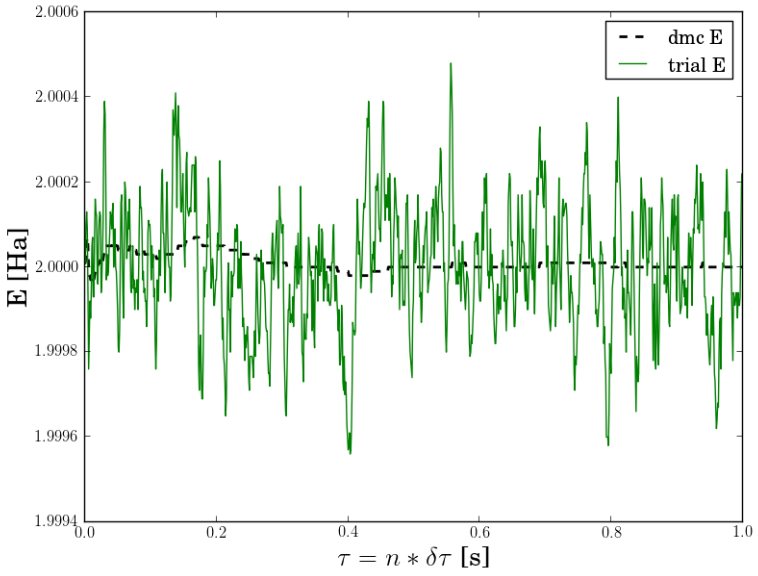


Figure 6.4: Illustration of the Diffusion Monte-Carlo (DMC) energy convergence for a two-particle two-dimensional quantum dot with frequency $\omega = 1$. The calculations are done with a variational parameter $\alpha = 0.75$, whereas the exact wave function is given for $\alpha = 1$. Unlike the case with the exact wave function presented in Figure 6.3, the trial energy oscillates around the exact value $E_0 = 2.0$. The final result reveals a DMC energy of 2.00000(2), where the original Variational Monte-Carlo (VMC) energy was 2.0042(3). This illustrates the power of DMC contra VMC in the interesting cases where the exact wave function is unknown. The calculation was done using 10000 random walkers.

6.3 Quantum Dots

The focus regarding quantum dots has been on studying the electron distribution as a function of the level of confinement. In addition, ground state energies are provided and compared to other many-body methods to demonstrate the efficiency and precision of Variational Monte-Carlo (VMC) and Diffusion Monte-Carlo (DMC). In the case of two-dimensional quantum dots, there are multiple published results, however, for three dimensions this is not the case. An introduction to quantum dots is given in Section 5.2.

The double-well quantum dot has not been a focus in this thesis, however, some simple results are provided to demonstrate the flexibility of the code.

6.3.1 Ground State Energies

Two-dimensional quantum dots

Table 6.5 presents the calculated ground state energies for two-dimensional quantum dots in addition to corresponding results from methods such as Similarity Renormalization Group theory (SRG), Coupled Cluster Singles and Doubles (CCSD) and Full Configuration Interaction (FCI). In addition, some previously published DMC results are supplied. The references are listed in the table caption.

In light of the variational nature of DMC and VMC, the results show that DMC provides a more precise estimate for the ground state energy than VMC, both in terms of lower energies and lower errors. The exact energy in the case of two electrons with $\omega = 1$ has been calculated in Ref. [45] and is $E_0 = 3$, which is in excellent agreement with the presented results.

The statistical errors in the DMC energies calculated in this thesis are lower than those provided in Ref. [46]. This may only be due to the fact that the calculations in this thesis have been run on a super computer. Running smaller simulations on fewer processors result in larger errors. Both the implementations successfully agree with the FCI result for two particles, which strongly indicates that the disagreements in results are a result of systematic errors.

In the case of two particles, DMC and FCI agree up to five decimals, which leads to the conclusion that DMC indeed is a very precise method. The SRG method is not variational in the sense that it can undershoot the exact energy. The DMC result should thus not be read as less precise in the cases where SRG provides a lower energy estimate. Diffusion Monte-Carlo and SRG are in excellent agreement for a large number of particles compared to FCI and CCSD, which drift away from the DMC results as their basis sizes shrink.

For high frequencies, the VMC energy is higher than the CCSD energy. The fact that both the methods are variational implies that CCSD performs better than VMC in this frequency range. However, looking at the results the lower frequency range, it is clear that VMC performs better than CCSD. This is due to the fact that CCSD struggles with convergence as the correlations within the system increase, indicated by the decrease in number of shells used to perform the calculations.

The DMC energy is overall smaller than the CCSD energy, which, due to the variational nature of the methods, implies that DMC performs better than CCSD. Nevertheless, the results for 56 particles are in excellent agreement.

N	ω	E_{VMC}	E_{DMC}	$E_{\mathrm{ref}}^{(a)}$	$E_{\mathrm{ref}}^{(b)}$	$E_{\mathrm{ref}}^{(c)}$	$E_{\mathrm{ref}}^{(d)}$
2	0.01	0.07406(5)	0.073839(2)	-	-	0.0738 {23}	0.07383505 {19}
	0.1	0.44130(5)	0.44079(1)	-	-	0.4408 {23}	0.44079191 {19}
	0.28	1.02215(5)	1.02164(1)	-	0.99263 {19}	1.0217 {23}	1.0216441 {19}
	0.5	1.66021(5)	1.65977(1)	1.65975(2)	1.643871 {19}	1.6599 {23}	1.6597723 {19}
	1.0	3.00030(5)	3.00000(1)	3.00000(3)	2.9902683 {19}	3.0002 {23}	3.0000001 {19}
6	0.1	3.5690(3)	3.55385(5)	-	3.49991 {18}	3.5805 {22}	3.551776 {9}
	0.28	7.6216(4)	7.60019(6)	7.6001(1)	7.56972 {18}	7.6254 {22}	7.599579 {6}
	0.5	11.8103(4)	11.78484(6)	11.7888(2)	11.76228 {18}	11.8055 {22}	11.785915 {6}
	1.0	20.1902(4)	20.15932(8)	20.1597(2)	20.14393 {18}	20.1734 {22}	20.160472 {8}
12	0.1	12.3162(5)	12.26984(8)	-	12.2253 {17}	12.3497 {21}	12.850344 {3}
	0.28	25.7015(6)	25.63577(9)	-	25.61084 {17}	25.7095 {21}	26.482570 {2}
	0.5	39.2343(6)	39.1596(1)	39.159(1)	39.13899 {17}	39.2194 {21}	39.922693 {2}
	1.0	65.7905(7)	65.7001(1)	65.700(1)	65.68304 {17}	65.7399 {21}	66.076116 {3}
20	0.1	30.0729(8)	29.9779(1)	-	29.95345 {16}	30.2700 {8}	34.204867 {1}
	0.28	62.0543(8)	61.9268(1)	61.922(2)	61.91368 {16}	62.0676 {20}	67.767987 {1}
	0.5	94.0236(9)	93.8752(1)	93.867(3)	93.86145 {16}	93.9889 {20}	100.93607 {1}
	1.0	156.062(1)	155.8822(1)	155.868(6)	155.8665 {16}	155.9569 {20}	164.61280 {1}
30	0.1	60.584(1)	60.4205(2)	-	60.43000 {15}	61.3827 {9}	-
	0.28	124.181(1)	123.9683(2)	-	123.9733 {15}	124.2111 {9}	-
	0.5	187.294(1)	187.0426(2)	-	187.0408 {15}	187.2231 {19}	-
	1.0	308.858(1)	308.5627(2)	-	308.5536 {15}	308.6810 {19}	-
42	0.1	107.881(1)	107.6389(2)	-	-	111.7170 {8}	-
	0.28	220.161(1)	219.8426(2)	-	219.8836 {14}	222.1401 {8}	-
	0.5	331.002(1)	330.6306(2)	-	330.6485 {14}	331.8901 {8}	-
	1.0	544.2(8)	542.9428(8)	-	542.9528 {14}	543.1155 {18}	-
56	0.1	176.269(2)	175.9553(7)	-	-	186.1034 {9}	-
	0.28	358.594(2)	358.145(2)	-	-	363.2048 {9}	-
	0.5	538.5(6)	537.353(2)	-	-	540.3430 {9}	-
	1	880.2(7)	879.3986(6)	-	-	879.6386 {17}	-

Table 6.5: Ground state energy results for two-dimensional N-electron quantum dots with frequency ω. Refs. (*a*): F. Pederiva [46] (DMC), (*b*): S. Reimann [47] (Similarity Renormalization Group theory), (*c*): C. Hirth [2] (Coupled Cluster Singles and Doubles), (*d*): V. K. B. Olsen [8] (Full Configuration Interaction). The numbers inside curly brackets denote the number of shells used above the last filled shell, i.e. above the so-called *Fermi-level* [18], to construct the basis for the corresponding methods.

N	ω	E_{VMC}	E_{DMC}	E_{ref}
2	0.01	0.07939(3)	0.079206(3)	-
	0.1	0.50024(8)	0.499997(3)	0.5
	0.28	1.20173(5)	1.201725(2)	-
	0.5	2.00005(2)	2.000000(2)	2.0
	1.0	3.73032(8)	3.730123(3)	-
8	0.1	5.7130(6)	5.7028(1)	-
	0.28	12.2040(8)	12.1927(1)	-
	0.5	18.9750(7)	18.9611(1)	-
	1.0	32.6842(8)	32.6680(1)	-
20	0.1	27.316(2)	27.2717(2)	-
	0.28	56.440(2)	56.3868(2)	-
	0.5	85.714(2)	85.6555(2)	-
	1.0	142.951(2)	142.8875(2)	-

Table 6.6: Ground state energy results for three-dimensional N-electron quantum dots with frequency ω. The values in the fifth column is exact calculations taken from Ref. [45]. The VMC result is as expected always higher than the corresponding DMC result.

Three-dimensional quantum dots

The results for three-dimensional quantum dots are presented in Table 6.6. Three-dimensional quantum dots do not have the same foothold in literature as the two-dimensional ones, hence no results are listed except for some exact solutions taken from Ref. [45].

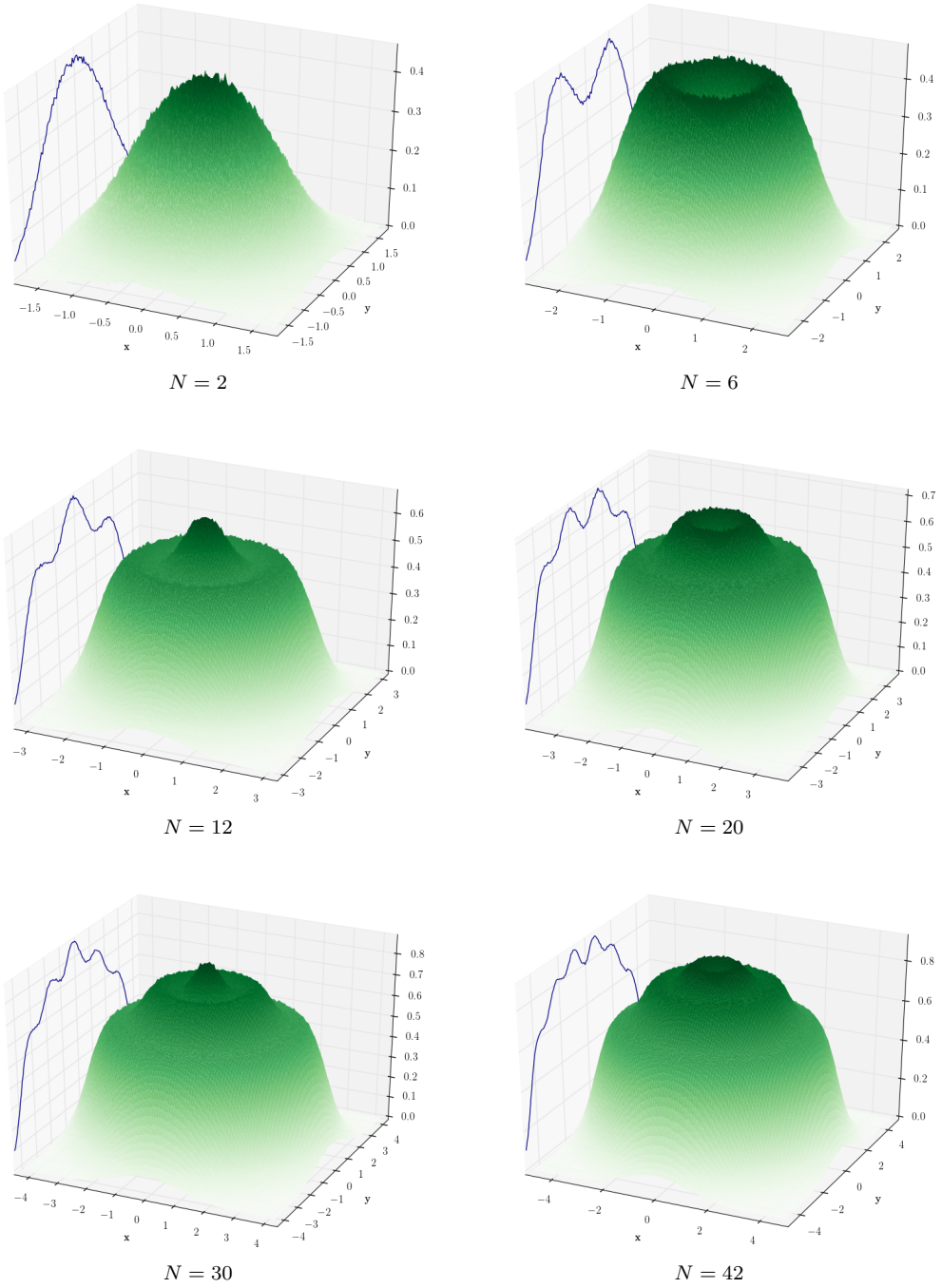
As expected, DMC reproduces the exact results for two particles. Compared to the exact results for two dimensions, which was reproduced with five digit precision, the exact results are reproduced with six decimal precision for three dimensions. However, for higher number of particles, the errors are of the same order of magnitude as for two dimensions, leading to the conclusion that DMC performs equally good in either case.

6.3.2 One-body Densities

The one-body densities are calculated using the methods described in Section 3.10.

Figure 6.5 presents the one-body densities for two-dimensional quantum dots. It is clear that the distributions are following a trend: The densities in the left column, that is, the densities for $N = 2$, 12, and 30 particles, are all similar in shape. The shape for the $N = 2$ density can be seen as the top of the $N = 12$ density, which in turn can be seen as the top of the $N = 30$ density. A physical explanation to this is that the shapes are conserved due to the fact that they represent energetically favorable configurations.

The same trend is present for the distributions in the right column, that is, the densities for $N = 6$, 20, and 42 particles. Viewing the distributions as a sequence of images, from the lowest number of particles to the highest, it is apparent that the shape propagates very much like water ripples. It is remarkable how the solutions to the most complex of equations can come in the form of simple patterns found all around nature.



$N = 2$ $N = 6$

$N = 12$ $N = 20$

$N = 30$ $N = 42$

Figure 6.5: Diffusion Monte-Carlo one-body densities for two-dimensional quantum dots with frequency $\omega = 1$. The number of particles N are listed below each density. It is apparent that the density behaves much like water ripples as the number of particles increase, conserving the shape in an oscillatory manner.

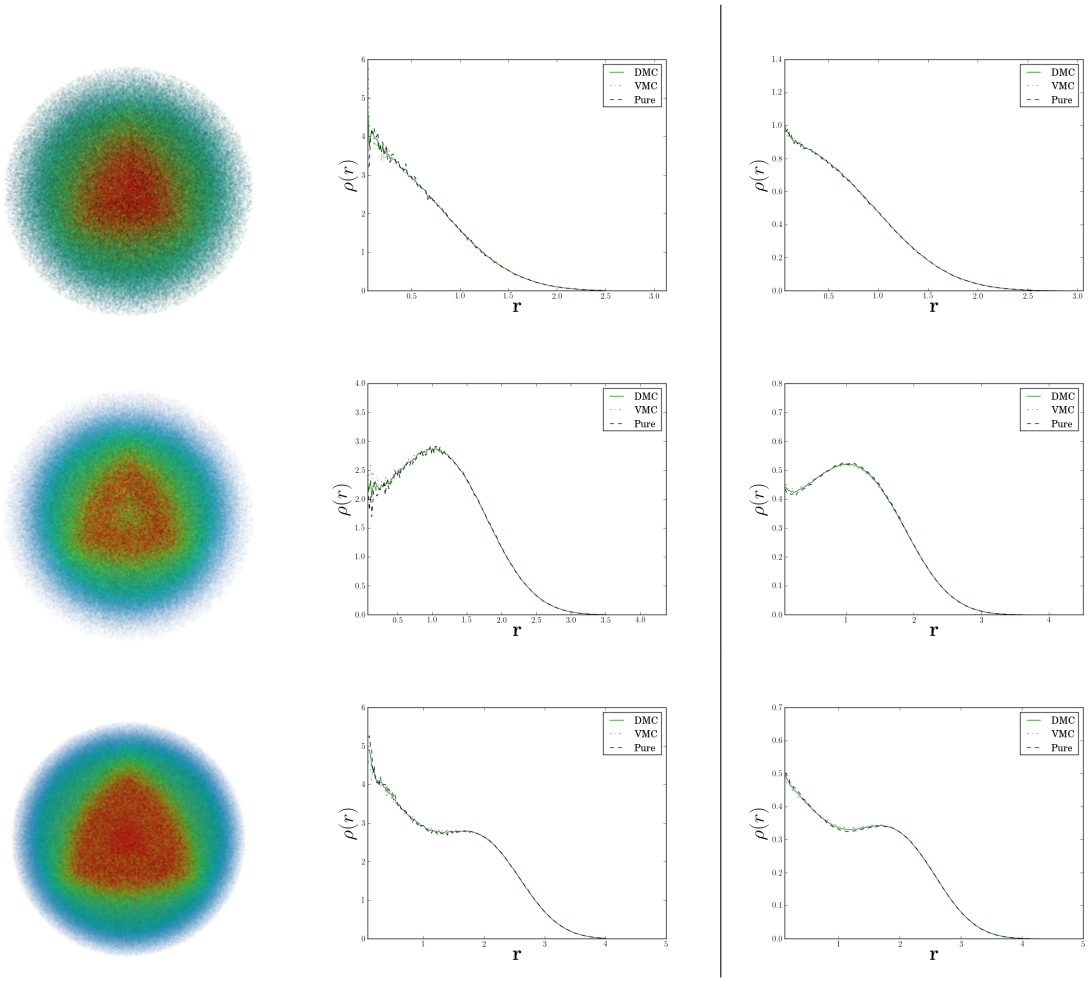


Figure 6.6: Left and middle column: One-body densities for quantum dots in three dimensions with frequency $\omega = 1$. A quarter of the spherical density is removed to present a better view of the core. Red and blue color indicate a low and high electron density, respectively. From top to bottom, the number of particles are 2, 8 and 20. Right column: One-body densities for two-dimensional quantum dots for $N = 2$, 6 and 12 electrons (from the top) with $\omega = 1$. It is apparent that the shape of the density is conserved as the third dimension is added. The radial densities are not normalized. Normalizing the densities would only change the vertical extent.

Due to the electron-electron interaction, the Schrödinger equation is not separable in Cartesian coordinates. It is therefore not given that the insights from two dimensions can be transferred to the three-dimensional case. Nevertheless, by looking at the one-body densities for three dimensions in Figure 6.6, it is apparent that the general density profile is independent of the dimension. The only thing separating two - and three-dimensional quantum dots is the number of electrons in the closed shells.

Note however, that this similarity only holds when the number of closed shells are equal. Comparing the two-dimensional density for $N = 20$ electrons from Figure 6.5 with the three-dimensional one for $N = 20$ electrons given above, it is apparent that the shape of the densities are not conserved with respect to the number of particles N alone.

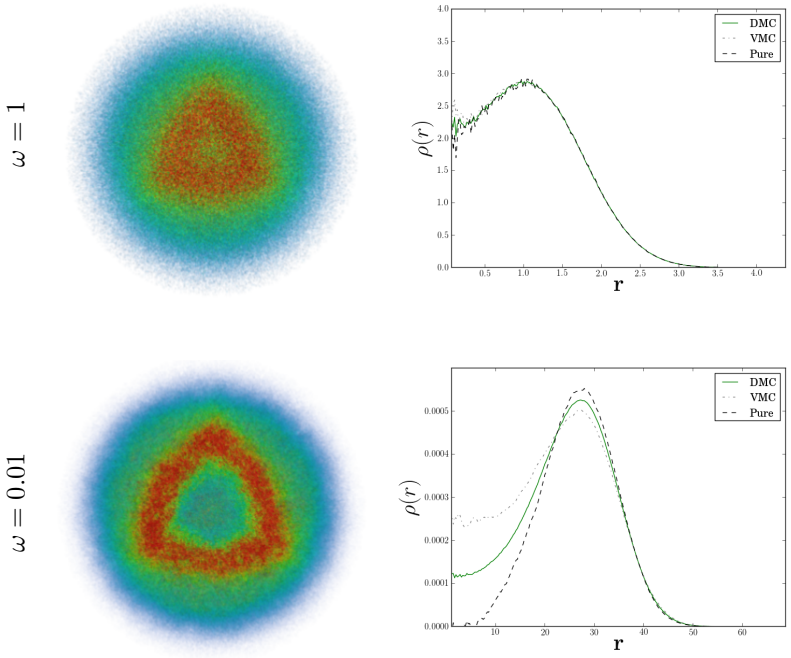


Figure 6.7: Comparison of the one-body densities for quantum dots in three dimensions for $N = 8$ electrons for a high - and low frequency ω displayed in the left column. It is apparent that the distribution becomes more narrow as the frequency is reduced. Red and blue color indicate a low and high electron density, respectively. A quarter of the spherical density is removed to present a better view of the core.

6.3.3 Lowering the frequency

An interesting effect of lowering the frequency is that the two - and three-dimensional densities no longer match. For example, the radial density for the three-dimensional 8-particle quantum dot from Figure 6.6 was a near perfect match to the two-dimensional one for $N = 6$ electrons, however, comparing the same densities for $\omega = 0.01$ from Figures 6.7 and 6.8, it is apparent that this is no longer the case; the two-dimensional density has a peak in the center region, whereas the three-dimensional density is zero in the center region.

From Figure 6.7 it is apparent that lowering the frequency increases the radial extent of the quantum dot, and thus lowers the electron density. Moreover, Figure 6.8 reveals that the electron density becomes similar in height and more localized across the quantum dot, which implies that the electrons on average are spread evenly in shell structures. The localization of the electrons is further verified in Figure 6.9, where is is clear that the expectation value of the total potential energy becomes larger than the corresponding kinetic energy.

In other words, an evenly spread and localized electron density give rise to *crystallization*[1]. The idea of an electron crystal was originally proposed by Wigner [48], hence the currently discussed phenomenon is referred to as a *Wigner molecule* or a *Wigner crystal*, which is expected for quantum dots in the limit of low electron densities where the total average potential energy becomes dominant over the corresponding kinetic energy [49–53]. These electronic crystals have been observed in experiments with for example liquid helium [54] and semiconductors [55].

[1]Unless at least one particle is frozen in the QMC simulations, the quantum dot densities should always be rotationally symmetric. Crystallization in a QMC perspective comes thus not in the form of actual "crystals", but rather as a rotated crystallized state.

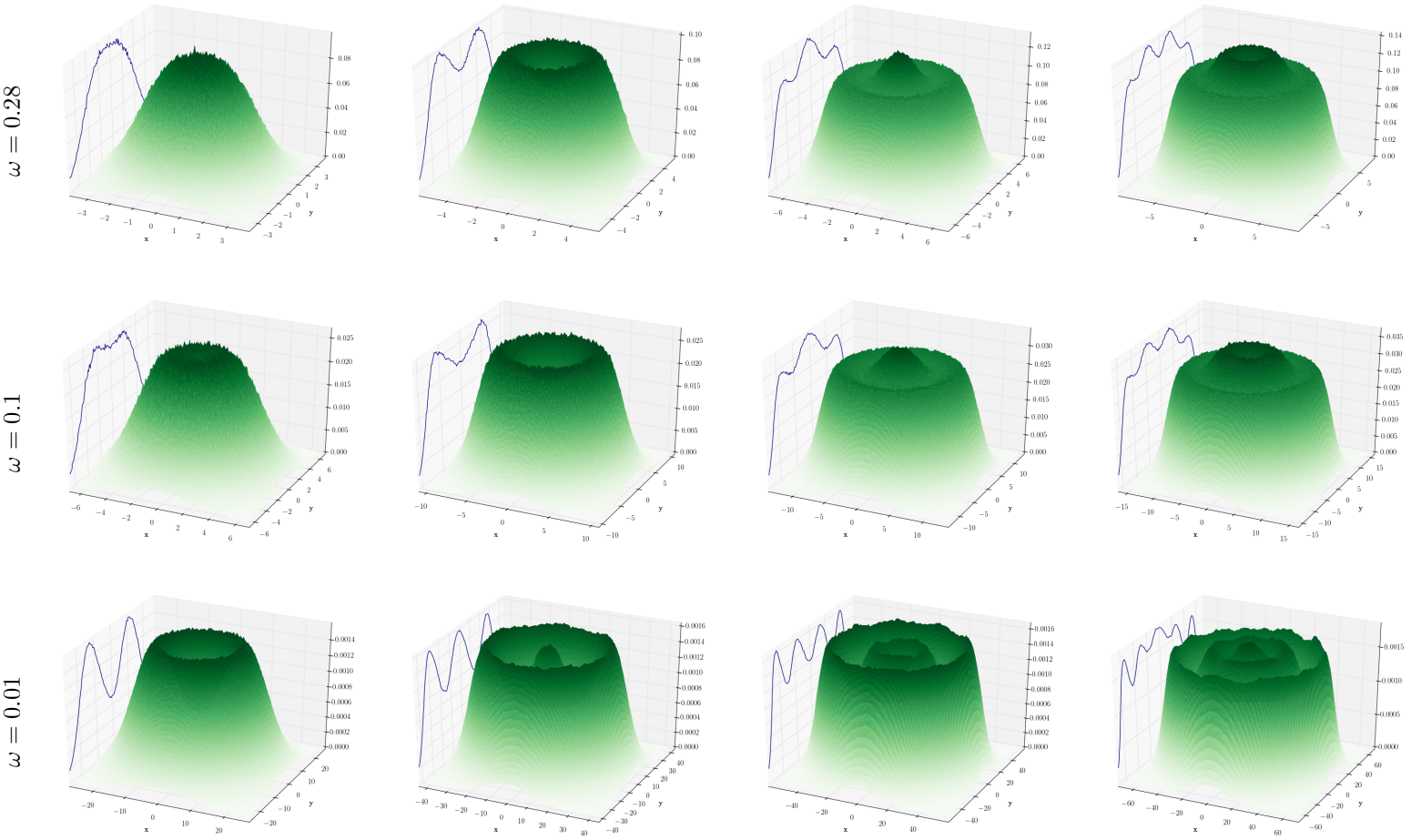
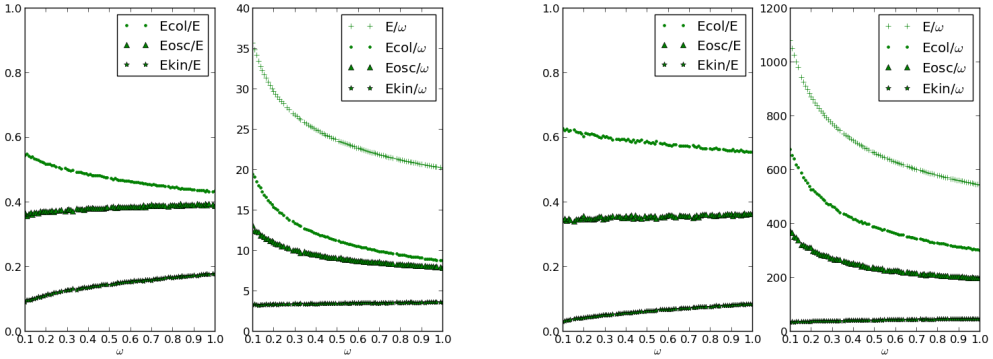


Figure 6.8: DMC One-body densities for Quantum Dots for decreasing oscillator frequencies ω and increasing number of particles N. Each row represents a given ω, and each column represents a given N. Notice that the densities for $\omega = 1$ (from Figure 6.5) are indistinguishable from those of $\omega = 0.28$ except for their radial extent. This trend has been verified in the case of $N = 30$, 42 and 56 electrons as well as for $\omega = 0.5$, however, for the sake of transparency, these results are left out of the current figure.



$N = 6$ $N = 42$

Figure 6.9: The relative magnitude of the expectation value of the different energy sources as a function of the frequency ω (left) together with the magnitude of the sources' energy contributions scaled with the oscillator frequency (right). The plots are supplied with legends to increase the readability. The different energy sources are the kinetic energy denoted *Ekin*, the oscillator potential energy denonted *Eosc*, and the electron-electron interaction energy denoted *Ecol*. Note that all given energies are expectation values. The values are calculated using two-dimensional quantum dots. The number of electrons N is displayed beneath each respective plot. It is apparent that the kinetic energy contribution is approximately constant in both cases. Moreover, the oscillator potential contribution is more or less constant for the relative energies (left sub-figures). The figure clearly indicates that the potential energy contributions from the oscillator and the electron-electron interaction tends to dominate over the kinetic energy at lower frequencies.

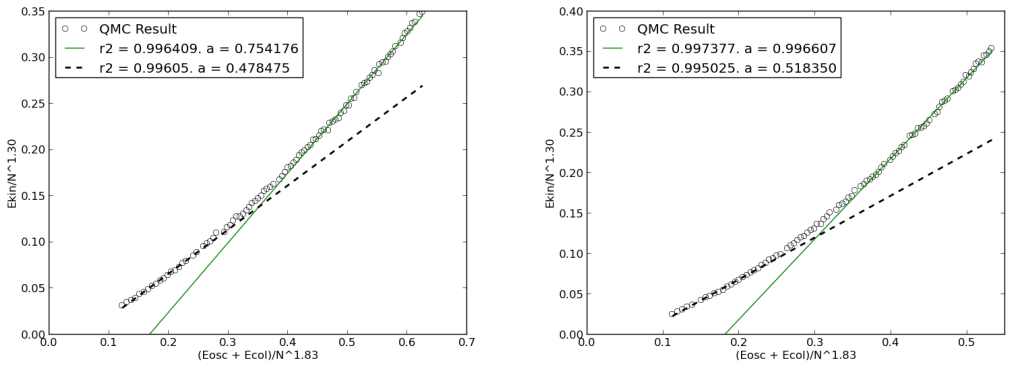
It is expected that the QMC Wigner crystal corresponds to the electrons localizing around the equilibrium positions of the classical Wigner crystal [49]. These classical Wigner crystals for two dimensional quantum dots given in Ref. [56] match the QMC densities for two-dimensional quantum dots at $\omega = 0.01$ given in Figure 6.8 very well.

It was mentioned previously that the Wigner crystallization of quantum dots came as a consequence of the average total potential energy being larger than the corresponding kinetic energy. This relationship between kinetic - and potential energy is closely related to the *virial theorem* from classical mechanics. The quantum mechanical version of the virial theorem was proven by Fock in 1930 [57] and reads

$$\widehat{\mathbf{V}}(\mathbf{r}) \propto r^{\gamma} \quad \longrightarrow \quad \langle \widehat{\mathbf{T}} \rangle = \frac{\gamma}{2} \langle \widehat{\mathbf{V}} \rangle, \tag{6.3}$$

where $\widehat{\mathbf{T}}$ and $\widehat{\mathbf{V}}$ denote the kinetic - and total potential energy operators, respectively. The important conclusion which can be drawn from this is that if two systems have an equal ratio of kinetic - to total potential energy, the systems behave identically in the sense that they follow the same effective potential, and thus have similar eigenstates.

From Figure 6.10 it is apparent that there is a remarkably constant slope for two different regions in the case of quantum dots, namely high - and low kinetic energy, which by looking at Figure 6.9 corresponds to high - and low frequencies. In light of previous discussions, one may suggest that the change in the slopes of Figure 6.10 corresponds to the quantum dot system making a transition into a Wigner crystallized state.



$N = 6$ $N = 42$

Figure 6.10: The total kinetic energy vs. the total potential energy of two-dimensional quantum dots. The linear lines are included as visual aids. The number of electrons N are displayed beneath each respective plot. The axes are scaled with a power of N to collapse the data to the same axis span. Once the kinetic energy drops below a certain energy dependent on the number of particles, the slope changes, which in light of the virial theorem from Eq. (6.3) indicates that the overall system changes properties. The data is fitted to linear lines with resulting slopes a displayed in the legend. The parameter $r2$ indicates how well the data fits a linear line. An exact fit yields $r2 = 1$.

6.3.4 Simulating a Double-well

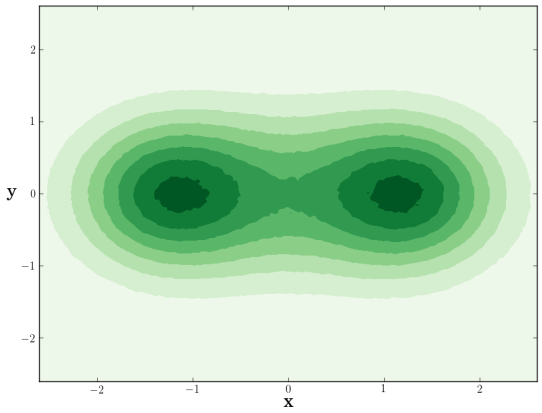


Figure 6.11: A countour plot of the trial wave function for a two-particle double-well quantum dot with the wells separated at a distance $R = 2$ in the x-direction using $m^* = \omega_0 = 1$. See Section 5.2 for an introduction to the double well potential. It is apparent that there is one electron located in each well, however, with a slight overlap in the middle region.

Figure 6.11 shows the distribution for a two-particle simulation. It is apparent that despite the wells being separated, their local distributions overlap, indicating that the electrons can *tunnel*, that is, they have a non-zero probability of appearing on the other side of the barrier. Comparing the distribution to the potential in Figure 5.4, it is clear that they match very well.

The DMC result is $\mathrm{E_{DMC}} = 2.3496(1)$, whereas the non-interacting energy is $E_0 = 2$, and the single-well energy is $E_0 = 3.0$. It is expected that the energy lies in between these, as $R = 0$ corresponds to the single well and $R \to \infty$ corresponds to the non-interacting case.

Atom	E_{VMC}	E_{DMC}	Expt.	ϵ_{rel}
He	-2.8903(2)	-2.9036(2)	-2.9037	$3.44 \cdot 10^{-5}$
Be	-14.145(2)	-14.657(2)	-14.6674	$7.10 \cdot 10^{-4}$
Ne	-127.853(2)	-128.765(4)	-128.9383	$1.34 \cdot 10^{-3}$
Mg	-197.269(3)	-199.904(8)	-200.054	$7.50 \cdot 10^{-4}$
Ar	-524.16(7)	-527.30(4)	-527.544	$4.63 \cdot 10^{-4}$
Kr	-2700(5)	-2749.9(2)	-2752.054976	$7.83 \cdot 10^{-4}$

Table 6.7: Ground state energies for Atoms calculated using Variational - and Diffusion Monte-Carlo. Experimental energies are listed in the last column. As we see, DMC is rather close to the experimental energy. The relative error $\epsilon_{\mathrm{rel}} = |E_{\mathrm{DMC}} - \mathrm{Expt.}|/|\mathrm{Expt.}|$ is as expected lowest in the case of helium. The experimental energies, that is, the best possible results available, are taken from Ref. [5] for He through Ar, and [6] for Kr.

6.4 Atoms

The focus regarding atoms has been on simulating heavy atoms using a simple ansatz for the trial wave function, and thus test its limits. Due to the importance of atoms in nature, precise calculations which are believed to be very close to the exact result for the given Hamiltonian are done. These results will be featured as experimental results in the following discussions. For heavier atoms, relativistic effects become important due to the high energies of the valance electrons. Hence atoms heavier than krypton have not been studied. The specifics regarding the model used for atoms are given in Section 5.1.

6.4.1 Ground State Energies

Table 6.7 presents the ground state energy results for different atoms together with the experimental results. As expected, helium has the best match with the corresponding experimental result out of all the atoms. The relative precision of the heavier atoms are in the range 10^{-3} - 10^{-4}, indicating that DMC performs equally well in all cases. However, the error in the calculations increases as the atoms become heavier. The calculations were done on a single node; running the calculations on several nodes with an increased number of walkers could reduce the errors somewhat.

In comparison to quantum dots, where the VMC and DMC results were relatively similar, it is evident that VMC performs rather poorly compared to DMC for atoms. Unlike quantum dots, the atomic systems allow for unbound states. This implies that the atomic systems in this thesis have an additional approximation in the trial wave function due to the fact that all the orbitals represent bound states. Nevertheless, this only further demonstrate the strengths of DMC to predict accurate results without much knowledge of the system at hand.

6.4.2 One-body densities

The one-body densities for the *noble gases*, that is, the closed shell atoms, are presented in Figure 6.13. Comparing these to the one-body densities for the alkaline earth metals, i.e. Be, Mg, etc., in Figure 6.12,

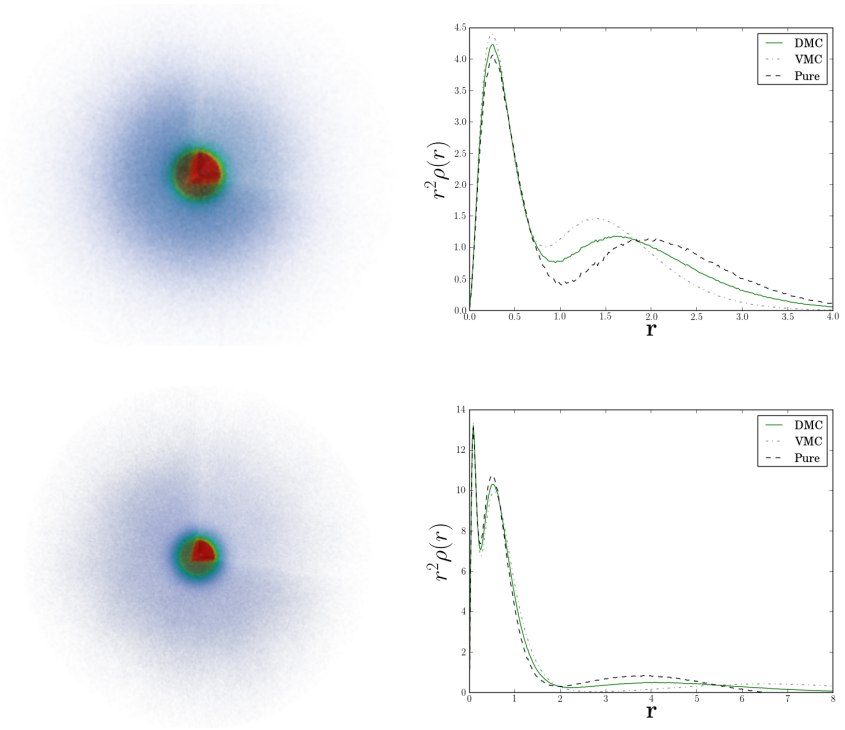


Figure 6.12: Three dimensional one-body densities (left column) and radial densities (right column) for alkaline earth metals; beryllium (top) and magnesium (bottom). A quarter of the spherical density is removed to present a better view of the core. Notice that the radial one-body densities in the right column are multiplied by the radius squared. This is done in order to reveal the characteristics behind a density which otherwise is a generic peak around the origin. Compared to the noble gases in Figure 6.13, the alkaline earth metals have a surrounding dispersed probability cloud due to having easily excitable valence electrons. The element is thus more unstable and potent for chemical reactions and molecular formations through covalent - and ionic bonds [58]. Red and blue color indicate a low and high electron density, respectively.

it is clear that the noble gases have a more confined electron distribution. This corresponds well to the fact that noble gases do not form compound materials, i.e. molecules [58]. The alkaline earth metals, on the other hand, are found naturally as parts of compound materials. The one-body densities of the alkaline earth metals spreading out in space are thus in excellent agreement with what is expected.

It is apparent that the VMC distribution and the pure distribution differ more in the case of alkaline earth metals than for noble gases. This implies that the trial wave function is better in the case of noble gases. To explain this phenomenon, it is important to realize that for closed shell systems, which is the case of noble gases, the energy needed to excite an electron into the next n-shell is higher than the energy needed to excite an electron to the next l-level in an open shell system such as the alkaline earth metals. The result of this is that the contributions from the exited states to the total wave function in Eq. (3.53) are higher for the alkaline earth metals than for the noble gases. This is exactly the same scenario as for high and low frequency quantum dots.

The approximation made in this thesis is that the trial wave function consists of a single determinant, thus neglecting the contribution from excited states. In light of the above discussion, this approximation is in other words better for noble gases than for the alkaline earth metals.

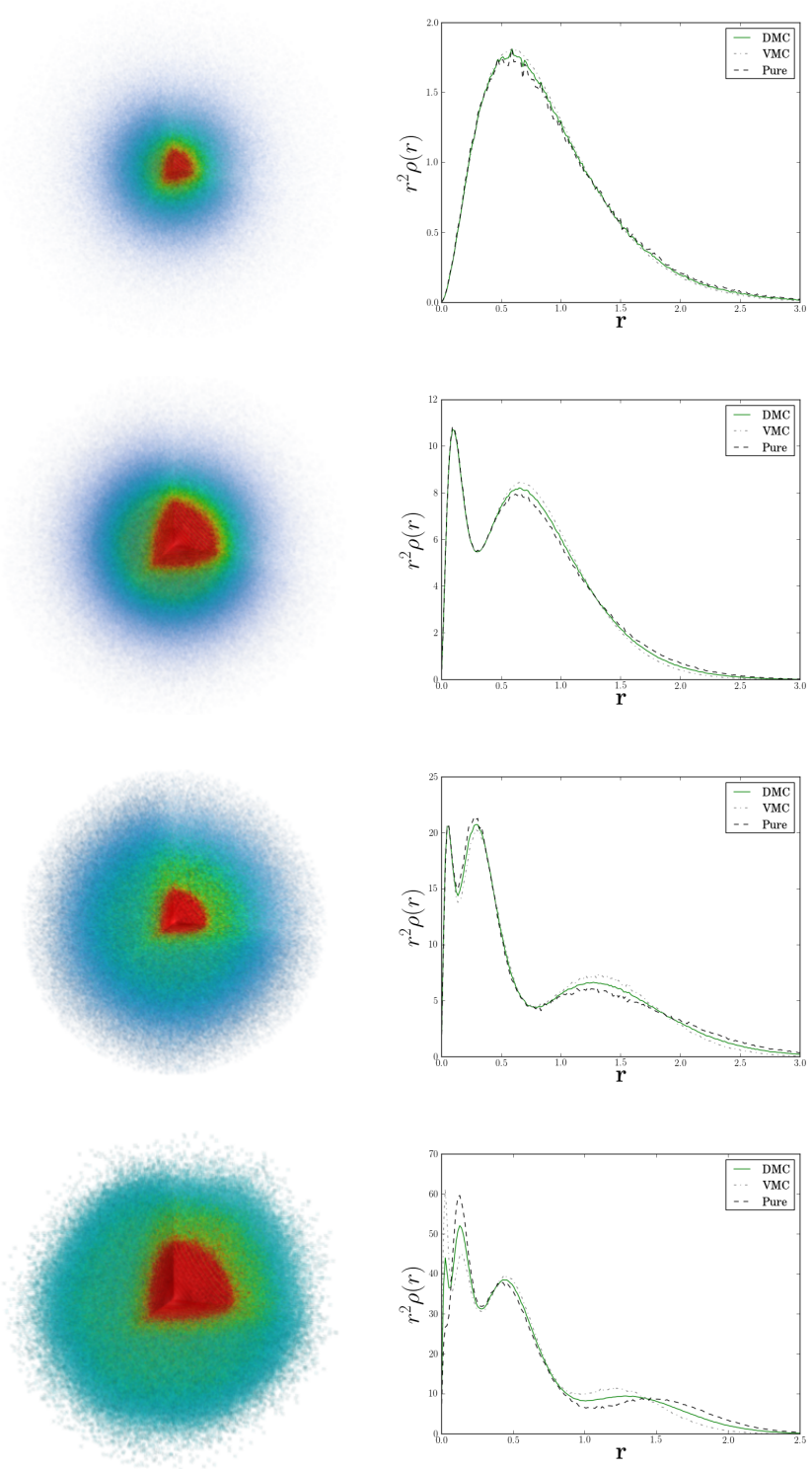


Figure 6.13: One-body densities for noble gases. Counting top to bottom: Helium, neon, argon and krypton. A quarter of the spherical density is removed to present a better view of the core. Red and blue color indicate a low and high electron density, respectively. Notice that the radial one-body densities in the right column are multiplied by the radius squared. This is done in order to reveal the characteristics behind a density which otherwise is a generic peak around the origin.

Molecule	R	E_{VMC}	E_{DMC}	Expt.	ϵ_{rel}
H_2	1.4	-1.1551(3)	-1.1745(3)	-1.1746	$8.51 \cdot 10^{-5}$
Li_2	5.051	-14.743(3)	-14.988(2)	-14.99544	$4.96 \cdot 10^{-4}$
Be_2	4.63	-28.666(5)	-29.301(5)	$-29.33854(5)$	$1.28 \cdot 10^{-3}$
B_2	3.005	-47.746(7)	-49.155(5)	-49.4184	$5.33 \cdot 10^{-3}$
C_2	2.3481	-72.590(8)	-74.95(1)	$-75.923(5)$	$1.28 \cdot 10^{-2}$
N_2	2.068	-102.78(1)	-106.05(2)	-109.5423	$3.19 \cdot 10^{-2}$
O_2	2.282	-143.97(2)	-148.53(2)	-150.3268	$1.2 \cdot 10^{-2}$

Table 6.8: Ground state energies for homonuclear diatomic molecules calculated using VMC and DMC. The distance between the atoms R are taken from Ref. [3] for H_2 and from Ref. [30] for Li_2 to O_2. The experimental energies, that is, the best possible results available, are taken from Ref. [3] for H_2 and from Ref. [4] for Li_2 to O_2. As expected DMC is closer to the experimental energy than VMC. Moreover, the relative error $\epsilon_{\text{rel}} = |E_{\text{DMC}} - \text{Expt.}|/|\text{Expt.}|$ is as expected lowest in the case of H_2, and increases with atomic number.

6.5 Homonuclear Diatomic Molecules

The focus regarding homonuclear diatomic molecules, from here on referred to as molecules, has been similar to the focus on atoms, with the exception of parameterizing atomic force fields which can be applied in molecular dynamics simulations. The implementation of molecular systems was achieved by adding ~ 200 lines of code. This fact by itself represents a successful result regarding the code structure. As for atoms, the optimal calculations are referred to as experimental results. Details regarding the transformation from atomic to molecular systems are given in Section 5.1.3.

6.5.1 Ground State Energies

Table 6.8 lists the VMC and DMC results with the corresponding experimental energies for H_2 through O_2. As expected, the two-particle result is very close to the experimental value with the same precision as the result for the helium atom in Table 6.7. The relative error from the experimental energy increases with atomic number, and is far higher than the errors in the case of pure atoms. This comes as a result of the trial wave function being less optimal due to the fact that it does not account for the atomic nuclei interaction term in the molecular Hamiltonian. Nevertheless, taking the simple nature of the trial wave function into consideration, the calculated energies are satisfyingly close to the experimental ones.

As with atoms, the energies were calculated on a single node, resulting in a rather big statistical error in DMC. Doing the calculations on a supercomputer with an increase in the number of walkers should decrease the errors.

6.5.2 One-body densities

Figure 6.14 presents the one-body densities of Li_2, Be_2 and O_2. The densities have strong peaks located at a distance equal to half of the listed core separation R, indicating that the atomic nuclei interaction still dominates the general shape of the distributions. Moreover, it is clear by looking at the figure that most of the electrons are on the side facing the opposite nucleus, leading to the conclusion that the molecules share a covalent bond [58]. This is especially clear in the case of the oxygen molecule, where there is a small formation of electrons on the inner side of the nuclei.

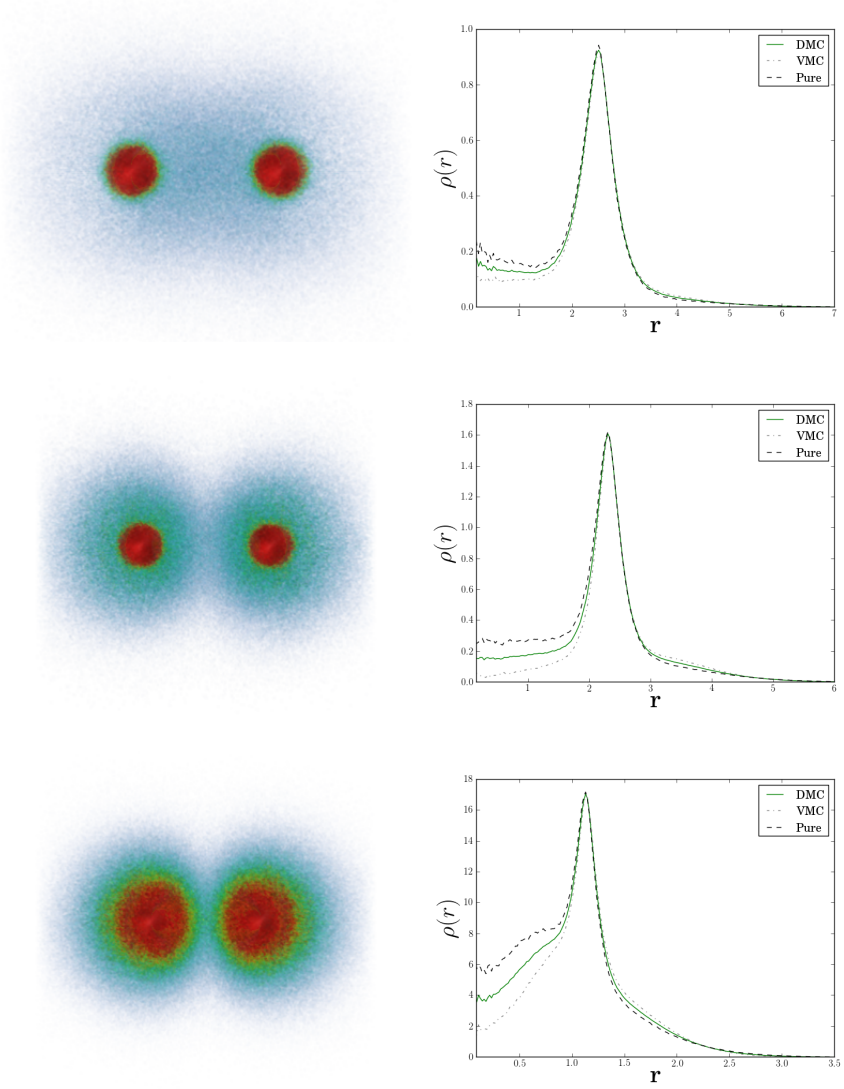


Figure 6.14: One-body densities of Li_2 (top), Be_2 (middle) and O_2 (bottom). The figures to the left are spherical densities sliced through the middle to better reveal the core structure. The figures to the right are radial one-body densities projected on the nucleus-nucleus axis. Red and blue color indicate a high and low electron density, respectively. The right-hand figures are symmetric around the origin.

6.5.3 Parameterizing Force Fields

In molecular dynamics, it is custom to use the *Lennard Jones 12-6 potential* as an ansatz to the interaction between pairs of atoms [10, 11]

$$V(R) = 4\epsilon\left(\left(\frac{\sigma}{R}\right)^{12} - \left(\frac{\sigma}{R}\right)^{6}\right), \tag{6.4}$$

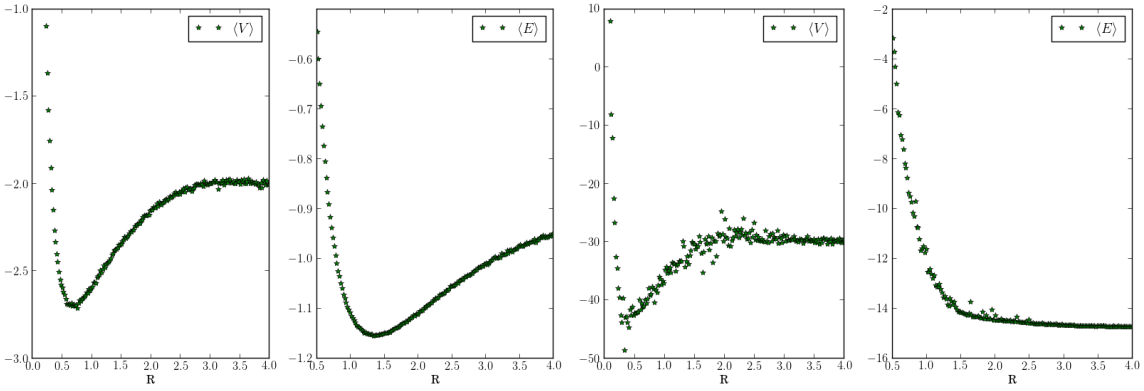
where ϵ and σ are parameters which can be fit to a given system.

However, the force field can be parameterized in greater detail using QMC calculations, resulting in a more precise molecular dynamics simulation [7]. The quantity of interest is the *force*, that is, the gradient of the potential. The classical potential in molecular dynamics does not correspond to the potential in the Schrödinger equation, due to the fact that the kinetic energy contribution from the electrons is not counted as part of the total kinetic energy in the molecular dynamics simulation. Hence it is the total energy of the Schrödinger equation which corresponds to the potential energy in molecular dynamics. In the case of diatomic molecules this means that

$$F_{\mathrm{MD}} = \frac{\mathrm{d}\langle E\rangle}{\mathrm{d}R}. \tag{6.5}$$

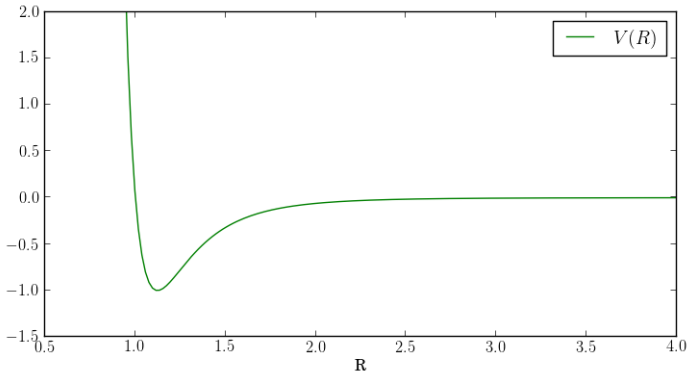
Expressions for this derivative can be obtained in ab-initio methods by using the Hellmann-Feynman theorem [7]. However, the derivative can be approximated by the slope of the energy in Figure 6.15. The figure shows that there are clear similarities between the widely used Lennard-Jones 12-6 potential and the results of QMC calculations done in this thesis, leading to the conclusion that the current state of the code can in fact produce approximations to atomic force fields for use in molecular dynamics.

For more complicated molecules, modelling the force using a single parameter R does not serve as a good approximation. However, the force can be found as a function of several angles, radii, etc., which in turn can be used to parameterize a more complicated molecular dynamics potential. An example of such a potential is the *ReaxFF* potential for hydrocarbons [59].



(a) H_2 (b) Li_2

Figure 6.15: Top figures: The distance between the atoms R vs. the potential and total energy calculated using QMC. To the left: H_2. To the right: Li_2. It is evident that there exists a well-defined energy minimum in the case of hydrogen. For lithium this is not the case, which is expected since lithium does not appear naturally in a diatomic gas phase, but rather as an ionic compound in other molecules [58]. Bottom figure: The general shape of the Lennard-Jones potential commonly used in molecular dynamics simulations as an approximation to the potential between atoms. The top figures clearly resemble the Lennard-Jones 12-6 potential, leading to the conclusion that QMC calculations can be used to parameterize a more realistic potential.



(c) The Lennard-Jones 12-6 potential with $\sigma = \epsilon = 1$.

7

Conclusions

The focus of this thesis was to develop an efficient and general Quantum Monte-Carlo (QMC) solver which could simulate various systems with a large number of particles. This was achieved by using object oriented C++, resulting in ~ 15000 lines of code. The code was redesigned a total of four times. The final structure was carefully planned in advance of the coding process. The linear algebra library *Armadillo* [60] was used in order to have a structured and easy code regarding all the matrices related to the walkers and the system in general.

It became apparent that in order to maintain efficiency for a high number of particles, closed form expressions for the single-particle wave function derivatives were necessary. For two-dimensional quantum dots, 112 of these expressions were needed to simulate the 56-particle system. Needless to say, this process had to be automated. This was achieved by using *SymPy*, an open source symbolic algebra package for Python, wrapped in a script which generated all the C++ necessary code within seconds. This task is described in detail in Appendix C. A total of 252 expressions were generated.

As the solver became more and more flexible, the dire need of a control script arose. Hence the current state of the code is 100% controlled by a Python script. The script translates configuration files to instructions which are fed to the pre-compiled C++ program. In addition, the script sets up the proper environment for the simulation, that is, it makes sure the simulation output is stored in a folder stamped with the current date and time, as well as with a supplied name tag. All in all, this made it possible to run an immense amount of different simulations without loosing track of the data.

In order to handle the increase in data, a script which automatically converted the output files to Latex tables were written. However, this would not uncover whether or not the simulations had converged. An additional script was thus written, which made the process of visualizing immense amounts of data very simple. Moreover, the data could be analyzed real-time, which meant that failing simulations could be detected and aborted before they were complete, saving a lot of time. Expanding the script to handle new output files was by design unproblematic. This script is covered in high detail in Appendix B.

Several Master projects over the years have involved QMC simulations of some sort, like Variational Monte-Carlo (VMC) calculations of two-dimensional quantum dots up to 42 particles [28], and VMC calculations of atoms up to Silicon (14 particles) [61]. In this thesis, the step was taken to full Diffusion Monte-Carlo (DMC) studies of both systems, adding three-dimensional - and double-well quantum dots in addition to homonuclear diatomic molecules. Additionally, the simulation sizes were increased to 56 electrons and krypton (36 particles) for two-dimensional quantum dots and atoms, respectively.

The optimization of the code was done by profiling the code, focusing on the parts which took the most time. Due to the general structure of the code, the function responsible for diffusing the particles was responsible for almost all of the run time. This function is at the core of Adaptive Stochastic Gradient

Descent (ASGD), VMC and DMC. In other words, the task of optimization the full code decreased down to the task of optimizing this specific function. By optimizing one part of the diffusion process at the time, the final run time was successfully reduced to 5% of the original.

Having successfully implemented five different systems demonstrates the code's generality. The implementation of molecules and the double-well was done by adding no more than 200 lines of code. Additionally, implementing three-dimensional quantum dots was done in an afternoon. Overall the code has lived up to every single one of the initial aims, with the exception of simulating bosons. Studying new fermionic systems were considered more interesting, hence specific implementations for bosons were abandoned.

For quantum dots, both VMC and DMC perform very well. The results from Full Configuration Interaction [8] for two particles, which are believed to be very close to the exact solution, are reproduced to five digits using DMC, with the VMC result being a little higher (2-3 digits). For atomic systems, the difference in results in VMC and DMC increase. This is expected since the trial wave function does not include the unbound states of the atoms. Nevertheless, DMC performs very well, reproducing the experimental results for two particles with an error at the level of 10^{-4}. For heavier atoms, the error increases somewhat, however, taking into consideration the simple trial wave function, the results are remarkably good.

Moreover, it was found that the radial distributions of two - and three-dimensional quantum dots with the same number of closed shells were remarkably similar. This, however, is only the case at high frequencies. For lower frequencies, both systems were found to transition into Wigner crystallized states, however, the distributions were no longer similar. This breaking of symmetry is an extremely interesting phenomenon, which can be studied in greater detail in order to say something general about how the number of spatial dimensions affect systems of confined electrons. The Wigner crystallization is covered in high detail in the literature [49–53], however, a new approach using the virial theorem was investigated in order to describe the transition.

It is clear that the energy of H_2 graphed as a function of the core separation resembles the Lennard-Jones 12-6 potential. This demonstrates that the code can be used to parameterize potential energies for use in molecular dynamics simulations, however, to produce realistic potentials, support for more complicated molecules must be implemented.

Prospects and future work

Shortly after handing in this thesis, I will focus my effort on studying the relationship between two - and three-dimensional quantum dots in higher detail. The goal is to publish my findings, and possibly say something general regarding how the number of dimensions affect a system of confined electrons.

Additionally, the double-well quantum dot will be studied in higher detail using realistic values for the parameters. These results can then be benchmarked with the results of Sigve Bøe Skattum and his *Multi-configuration Time Dependent Hartree-Fock* solver [19].

My supervisor and I will at the same time work with implementing a momentum space version of QMC. This has the potential of describing nuclear interactions in great detail [62].

I will continue my academic career as a PhD student in the field of *multi-scale physics*. The transition from QMC to molecular dynamics will thus be of high interest. The plan is to expand the code to general molecules. However, in order to maintain a reasonable precision, the single-particle wave functions need to be optimized. Hence implementing a Hartree-Fock solver [18] or using a Coupled Cluster wave function [63] will be prioritized.

Appendices

A

Dirac Notation

Calculations involving sums over inner products of orthogonal states are common in Quantum Mechanics. This due to the fact that eigenfunctions of Hermitian operators, which is the kind of operators which represent observables [17], are necessarily orthogonal [31]. These inner-products will in many cases result in either zero or one, i.e. the *Kronecker-delta* function δ_{ij}; explicitly calculating the integrals is unnecessary.

Dirac notation is a notation in which quantum states are represented as abstract components of a *Hilbert space*, i.e. an inner product space. This implies that the inner-product between two states are represented by these states alone, without the integral over a specific basis, which makes derivations a lot cleaner and general in the sense that no specific basis is needed.

Extracting the abstract state from a wave function is done by realizing that the wave function can be written as the inner product between the position basis eigenstates $|x\rangle$ and the abstract quantum state $|\psi\rangle$

$$\psi(x) = \langle r, \psi\rangle \equiv \langle x|\psi\rangle = \langle x| \times |\psi\rangle \,.$$

The notation is designed to be simple. The right hand side of the inner product is called a *ket*, while the left hand side is called a *bra*. Combining both of them leaves you with an inner product bracket, hence Dirac notation is commonly referred to as *bra-ket* notation.

To demonstrate the simplicity introduced with this notation, imagine a coupled two-level spin-$\frac{1}{2}$ system in the following state

$$|\chi\rangle = N\Big[\,|\uparrow\downarrow\rangle - i\,|\downarrow\uparrow\rangle\,\Big] \tag{A.1}$$

$$\langle\chi| = N\Big[\,\langle\uparrow\downarrow| + i\,\langle\downarrow\uparrow|\,\Big] \tag{A.2}$$

Using the fact that both the $|\chi\rangle$ state and the two-level spin states should be orthonormal, the normalization factor can be calculated without explicitly setting up any integrals

$$\begin{aligned}
\langle\chi|\chi\rangle &= N^2\Big[\langle\uparrow\downarrow| + i\langle\downarrow\uparrow|\Big]\Big[|\uparrow\downarrow\rangle - i|\downarrow\uparrow\rangle\Big] \\
&= N^2\Big[\langle\uparrow\downarrow|\uparrow\downarrow\rangle + i\langle\downarrow\uparrow|\uparrow\downarrow\rangle - i\langle\uparrow\downarrow|\downarrow\uparrow\rangle + \langle\downarrow\uparrow|\downarrow\uparrow\rangle\Big] \\
&= N^2\Big[1 + 0 - 0 + 1\Big] \\
&= 2N^2 \\
&= 1,
\end{aligned}$$

This implies the trivial solution $N = 1/\sqrt{2}$. With this powerful notation at hand, important properties such as the *completeness relation* of a set of states can be shown. A standard strategy is to start by expanding one state $|\phi\rangle$ in a complete set of different states $|\psi_i\rangle$:

$$\begin{aligned}
|\phi\rangle &= \sum_i c_i |\psi_i\rangle \\
\langle\psi_k|\phi\rangle &= \sum_i c_i \underbrace{\langle\psi_k|\psi_i\rangle}_{\delta_{ik}} \\
&= c_k \\
|\phi\rangle &= \sum_i \langle\psi_i|\phi\rangle |\psi_i\rangle \\
&= \left[\sum_i |\psi_i\rangle\langle\psi_i|\right]|\phi\rangle
\end{aligned}$$

which implies that

$$\sum_i |\psi_i\rangle\langle\psi_i| = \mathbb{1} \tag{A.3}$$

for any complete set of orthonormal states $|\psi_i\rangle$. Calculating the corresponding identity for a continuous basis like e.g. the position basis yields

$$\int |\psi(x)|^2 dx = 1 \tag{A.4}$$

$$\begin{aligned}
\int |\psi(x)|^2 dx &= \int \psi^*(x)\psi(x)\mathrm{d}x \\
&= \int \langle\psi|x\rangle\langle x|\psi\rangle\,\mathrm{d}x \\
&= \langle\psi|\left[\int |x\rangle\langle x|\,dx\right]|\psi\rangle\,.
\end{aligned} \tag{A.5}$$

Combining eq. A.4 and eq. A.5 with the fact that $\langle\psi|\psi\rangle = 1$ yields the identity

$$\int |x\rangle\langle x|\,dx = \mathbb{1}. \tag{A.6}$$

Looking back at the introductory example, this identity is exactly what is extracted when a wave function is described as an inner product instead of an explicit function.

B

DCViz: Visualization of Data

With a code framework increasing in complexity comes an increasing need for tools to ease the interface between the code and the developer(s). In computational science, a must-have tool is a tool for efficient visualization of data; there is only so much information a single number can hold. To supplement the QMC code, a visualization tool named DCViz (**D**ynamic **C**olumn data **Vi**sualiz**e**r) has been developed.

The tool is written in Python, designed to plot data stored in columns. The tool is not designed explicitly for the QMC framework, and has been successfully applied to codes by several Master students at the time of this thesis. The plot library used is *Matplotlib* [64] with a graphical user interface coded using *PySide* [65]. The data can be plotted dynamically at a specified interval, and designed to be run parallel to the main application, e.g. DMC.

DCViz is available at `https://github.com/jorgehog/DCViz`

B.1 Basic Usage

The application is centered around the `mainloop()` function, which handles the extraction of data, the figures and so on. The virtual function `plot()` is where the user specifies how the data is transformed into specified figures by creating a subclass which overloads it. The superclass handles everything from safe reading from dynamically changing files, efficient and safe re-plotting of data, etc. automatically. The tool is designed to be simple to use by having a minimalistic interface for implementing new visualization classes. The only necessary members to specify in a new implementation is described in the first three sections, from where the remaining sections will cover additional support.

The figure map

Represented by the member variable `figMap`, the figure map is where the user specifies the figure setup, that is, the names of the main-figures and their sub-figures. Consider the following figure map:

```
figMap = {"mainFig1": ["subFig1", "subFig2"], "mainFig2": []}
```

This would cause DCViz to create two main-figures `self.mainFig1` and `self.mainFig2`, which can be accessed in the plot function. Moreover, the first main-figure will contain two sub-figures accessible through `self.subFig1` and `self.subFig2`. These sub-figures will be stacked vertically if not `stack="H"` is specified, in which they will be stacked horizontally.

The name tag

Having to manually combine a data file with the correct subclass implementation is annoying, hence DCViz is designed to automate this process. Assuming a dataset to be specified by a unique pattern of characters, i.e. a *name tag*, this name tag can be tied to a specific subclass implementation, allowing DCViz to automatically match a specific filename with the correct subclass. Name tags are

```
nametag = "DMC_out_\d+\.dat"
```

The name tag has *regular expressions* (regex) support, which in the case of the above example allows DCViz to recognize any filename starting with "DMC_out_" followed by any integer and ending with ".dat" as belonging to this specific subclass. This is a necessary functionality, as filenames often differ between runs, that is, the filename is specified by e.g. a time step, which does not fit an absolute generic expression. The subclasses must be implemented in the file `DCViz_classes.py` in order for the automagic detection to work.

To summarize, the name tag invokes the following functionality

```
import DCvizWrapper, DCViz_classes

#DCViz automagically executes the mainloop for the
#subclass with a nametag matching 'filename'
DCVizWrapper.main(filename)

#This would be the alternative, where 'specific_class' needs to be manually selected.
specificClass = DCViz_classes.myDCVizClass #myDCVizClass matches 'filename'
specificClass(filename).mainloop()
```

The plot function

Now that the figures and the name tag has been specified, all that remains for a fully functional DCViz instance is the actual plot function

```
def plot(self, data)
```

where `data` contains the data harvested from the supplied filename. The columns can then be accessed easily by e.g.

```
col1, col2, col3 = data
```

which can then in turn be used in standard Matplotlib functions with the figures from `figMap`.

Additional (optional) support

Additional parameters can be overloaded for additional functionality

`nCols`	The number of columns present in the file. Will be automatically detected unless the data is stored in binary format.
`skipRows`	The number of initial rows to skip. Will be automatically detected unless the data is stored as a single column.
`skipCols`	The number of initial columns to skip. Defaults to zero.
`armaBin`	Boolean flag. If set to true, the data is assumed to be stored in Armadillo's binary format (doubles). Number of columns and rows will be read from the file header.
`fileBin`	Boolean flag. If set to true, the data is assumed to be stored in binary format. The number of columns must be specified.

The LaTeXsupport is enabled if the correct packages are installed.

An example

```
#DCViz_classes.py

from DCViz_sup import DCVizPlotter #Import the superclass

class myTestClass(DCVizPlotter): #Create a new subclass
    nametag =  'testcase\d\.dat' #filename with regex support

    #1 figure with 1 subfigure
    figMap = {'fig1': ['subfig1']}

    #skip first row (must be supplied since the data is 1D).
    skipRows = 1

    def plot(self, data):
        column1 = data[0]

        self.subfig1.set_title('I have $\LaTeX$ support!')

        self.subfig1.set_ylim([-1,1])

        self.subfig1.plot(column1)

        #exit function
```

Families

A specific implementation can be flagged as belonging to a family of similar files, that is, files in the same directory matching the same name tag. Flagging a specific DCViz subclass as a family is achieved by setting the class member variable `isFamilyMember` to true. When a family class is initialized with a file, DCViz scans the file's folder for additional matches to this specific class. If several matches are found, all of these are loaded into the `data` object given as input to the plot function. In this case `data[i]` contains the column data of file i.

To keep track of which file a given data-set was loaded from, a list `self.familyFileNames` is created, where element i is the filename corresponding to `data[i]`. To demonstrate this, consider the following example

```
isFamilyMember = True
def plot(self, data)

    file1, file2 = data
    fileName1, fileName2 = self.familyFileNames

    col1_1, col_2_1 = file1
    col1_2, col_2_2 = file2
    #...
```

A class member string `familyName` can be overridden to display a more general name in the auto-detection feedback.

Families are an important functionality in the cases where the necessary data is spread across several files. For instance, in the QMC library, the radial distributions of both VMC and DMC are needed in order to generate the plots shown in Figure 6.6 of the results chapter. These results may be generated in separate runs, which implies that they either needs to be loaded as a family, or be concatenated beforehand. Which dataset belongs to VMC and DMC can be extracted from the list of family file names.

All the previous mentioned functionality is available for families.

Family example

```
#DCViz_classes.py

from DCViz_sup import DCVizPlotter

class myTestClassFamily(DCVizPlotter):
    nametag =  'testcaseFamily\d\.dat' #filename with regex support

    #1 figure with 3 subfigures
    figMap = {'fig1': ['subfig1', 'subfig2', 'subfig3']}

    #skip first row of each data file.
    skipRows = 1

    #Using this flag will read all the files matching the nametag
    #(in the same folder.) and make them aviable in the data arg
    isFamilyMember = True
    familyName = "testcase"

    def plot(self, data):

        mainFig = self.fig1
        mainFig.suptitle('I have $\LaTeX$ support!')
        subfigs = [self.subfig1, self.subfig2, self.subfig3]

        #Notice that fileData.data is plotted (the numpy matrix of the columns)
        #and not fileData alone, as fileData is a 'dataGenerator' instance
        #used to speed up file reading. Alternatively, data[:] could be sent
        for subfig, fileData in zip(subfigs, data):
            subfig.plot(fileData.data)
            subfig.set_ylim([-1,1])
```

loading e.g. `testcaseFamily0.dat` would automatically load `testcaseFamily1.dat` etc. as well.

Dynamic mode

Dynamic mode in DCViz is enabled on construction of the object

```
DCVizObj = myDCVizClass(filename, dynamic=True)
DCVizObj.mainloop()
```

This flag lets the mainloop know that it should not stop after the initial plot is generated, but rather keep on reading and plotting the file(s) until the user ends the loop with either a keyboard-interrupt (which is caught and safely handled), or in the case of using the GUI, with the stop button.

In order to make this functionality more CPU-friendly, a `delay` parameter can be adjusted to specify a pause period in between re-plotting.

Saving figures to file

The generated figures can be saved to file by passing a flag to the constructor

```
DCVizObj = myDCVizClass(filename, toFile=True)
DCVizObj.mainloop()
```

In this case, dynamic mode is disabled and the figures will not be drawn on screen, but rather saved in a subfolder of the supplied filename's folder called `DCViz_out`.

B.1.1 The Terminal Client

The `DCVizWrapper.py` script is designed to be called from the terminal with the path to a datafile specified as command line input. From here it automatically selects the correct subclass based on the filename:

```
jorgen@teleport:~$ python DCVizWrapper.py ./ASGD_out.dat

[ Detector ] Found subclasses 'myTestClass', 'myTestClassFamily', 'EnergyTrail',
                              'Blocking', 'DMC_OUT', 'radial_out', 'dist_out',
                              'R_vs_E', 'E_vs_w', 'testBinFile', 'MIN_OUT'
[  DCViz   ] Matched [ASGD_out.dat] with [MIN_OUT]
[  DCViz   ] Press any key to exit
```

If the option `-d` is supplied, dynamic mode is activated:

```
jorgen@teleport:~$ python DCVizWrapper.py ./ASGD_out.dat -d

[ Detector ] Found subclasses ......
[  DCViz   ] Matched [ASGD_out.dat] with [MIN_OUT]
[  DCViz   ] Interrupt dynamic mode with CTRL+C
^C[  DCViz   ] Ending session...
```

Saving figures through the terminal client is done by supplying the flag `-f` to the command line together with a folder `aDir`, whose content will then be traversed recursively. For every file matching a DCViz name tag, the file data will be loaded and its figure(s) saved to `aDir/DCViz_out/`. In case of family members, only one instance needs to be run (they would all produce the same image), hence "family portraits" are taken only once:

```
jorgen@teleport:~$ python DCVizWrapper.py ~/scratch/QMC_SCRATCH/ -f

[ Detector ] Found subclasses ......
[  DCViz   ] Matched [ASGD_out.dat] with [MIN_OUT]
[  DCViz   ] Figure(s) successfully saved.
[  DCViz   ] Matched [dist_out_QDots2c1vmc_edge3.05184.arma] with [dist_out]
[  DCViz   ] Figure(s) successfully saved.
[  DCViz   ] Matched [dist_out_QDots2c1vmc_edge3.09192.arma] with [dist_out]
[  DCViz   ] Family portait already taken, skipping...
[  DCViz   ] Matched [radial_out_QDots2c1vmc_edge3.05184.arma] with [radial_out]
[  DCViz   ] Figure(s) successfully saved.
[  DCViz   ] Matched [radial_out_QDots2c1vmc_edge3.09192.arma] with [radial_out]
[  DCViz   ] Family portait already taken, skipping...
```

The terminal client provides extremely efficient and robust visualization of data. When e.g. blocking data from 20 QMC runs, the automated figure saving functionality is gold.

B.1.2 The Application Programming Interface (API)

DCViz has been developed to interface nicely with any Python script. Given a path to the data file, all that is needed in order to visualize it is to include the wrapper function used by the terminal client:

```
import DCVizWrapper as viz
dynamicMode = False #or true

...
#Generate some data and save it to the file myDataFile (including path)

#DCVizWrapper.main() automatically detects the subclass implementation
#matching the specified file. Thread safe and easily interruptable.
viz.main(myDataFile, dynamic=dynamicMode, toFile=toFile)
```

If on the other hand the data needs to be directly visualized without saving it to file, the pure API function `rawDataAPI` can be called directly with a numpy array `data`. If the plot should be saved to file, this can be enabled by supplying an arbitrary file-path (e.g. `/home/me/superDuper.png`) and setting `toFile=True`.

```
from DCViz_classes import myDCVizClass

#Generate some data
myDCVizObj = myDCVizClass(saveFileName, toFile=ToFile)
myDCVizObj.rawDataAPI(data)
```

The GUI

The script `DCVizGUI.py` sets up a GUI for visualizing data using DCViz. The GUI is implemented using PySide (python wrapper for Qt), and is designed to be simple. Data files are loaded from an open-file dialog (`Ctrl+s` for entire folders or `Ctrl+o` for individual files), and will appear in a drop-down menu once loaded labeled with the corresponding class name. The play button executes the main loop of the currently selected data file. Dynamic mode is selected though a check-box, and the pause interval is set by a slider (from zero to ten seconds). Dynamic mode is interrupted by pressing the stop button. Warnings can be disabled through the configuration file. A screenshot of the GUI in action is presented in Figure B.1.

The GUI can be opened from any Python script by calling the `main` function (should be threaded if used as part of another application). If a path is supplied to the function, this path will be default in all file dialogues. Defaults to the current working directory.

The following is a tiny script executing the GUI for a QMC application. If no path is supplied at the command line, the default path is set to the scratch path.

```
import sys, os
from pyLibQMC import paths #contains all files specific to the QMC library

#Adds DCVizGUI to the Python path
sys.path.append(os.path.join(paths.toolsPath, "DCViz", "GUI"))

import DCVizGUI

if __name__ == "__main__":

    if len(sys.argv) > 1:
        path = sys.argv[1]
    path = paths.scratchPath

    sys.exit(DCVizGUI.main(path))
```

The python script responsible for starting the QMC program and setting up the environments for simulations in this thesis automatically starts the GUI in the simulation main folder, which makes the visualizing the simulation extremely easy.

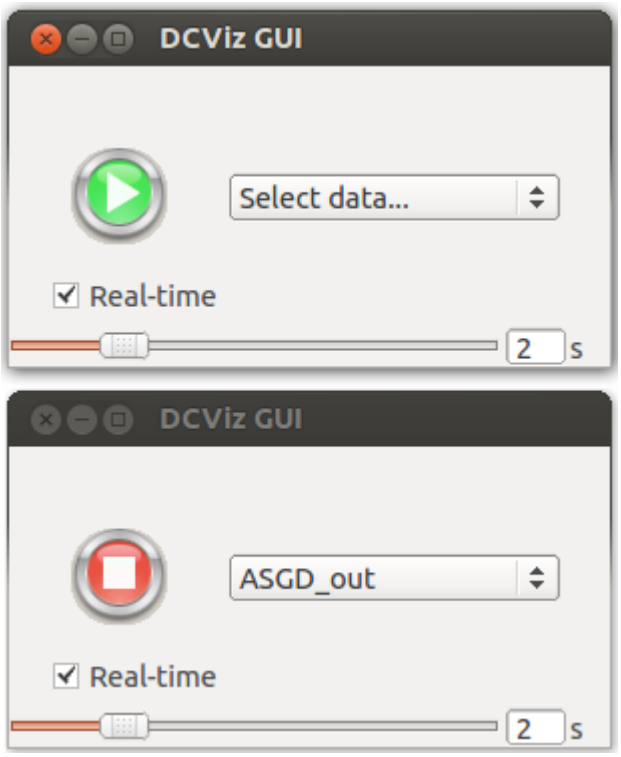


Figure B.1: Two consequent screen shots of the GUI. The first (top) is taken directly after the detector is finished loading files into the drop-down menu. The second is taken directly after the job is started.

Alternatively, `DCVizGUI.py` can be executed directly from the terminal with an optional default path as first command line argument.

The following is the terminal feedback supplied from opening the GUI

```
 .../DCViz/GUI$ python DCVizGUI.py
[ Detector ]:  Found subclasses 'myTestClass', 'myTestClassFamily', 'EnergyTrail',
'Blocking', 'DMC_OUT', 'radial_out', 'dist_out', 'testBinFile', 'MIN_OUT'
[   GUI    ]:  Data reset.
```

Selecting a folder from the open-folder dialog initializes the detector on all file content

```
[ Detector ]:  matched [  DMC_out.dat  ] with [    DMC_OUT    ]
[ Detector ]:  matched [  ASGD_out.dat ] with [    MIN_OUT    ]
[ Detector ]:  matched [blocking_DMC_out.dat] with [    Blocking   ]
[ Detector ]:  'blocking_MIN_out0_RAWDATA.arma' does not match any DCViz class
[ Detector ]:  'blocking_DMC_out_RAWDATA.arma' does not match any DCViz class
[ Detector ]:  matched [blocking_VMC_out.dat] with [    Blocking   ]
```

Executing a specific file selected from the drop-down menu starts a threaded job, hence several non-dynamic jobs can be ran at once. The limit is set to one dynamic job pr. application due to the high CPU cost (in case of a low pause timer).

The terminal output can be silenced through to configuration file to not interfere with the standard output of an application. Alternatively, the GUI thread can redirect its standard output to file.

C

Auto-generation with SymPy

"*SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible. SymPy is written entirely in Python and does not require any external libraries.*"

- The SymPy Home Page [66]

Thie aim of this appendix will be on using SymPy to calculate closed form expressions for single-particle wave functions needed to optimize the calculations of the Slater gradient and Laplacian. For systems of many particles, it is crucial to have these expressions in order for the code to remain efficient.

Calculating these expressions by hand is a waste of time, given that the complexity of the expressions is proportional to the magnitude of the quantum number, which again scales with the number of particles, and little new insights are gained from doing the calculations. In the case of a 56 particle Quantum Dot, the number of unique derivatives involved in the simulation is 112.

C.1 Usage

SymPy is, as described in the introductory quote, designed to be simple to use. This section will cover the basics needed to calculate gradients and Laplacians, auto-generating C++ - and Latex code.

C.1.1 Symbolic Algebra

In order for SymPy to recognize e.g. `x` as a symbol, that is, a *mathematical variable*, special action must be made. In contrast to programming variables, symbols are not initialized to a value. Initializing symbols can be done in several ways, the two most common are listed below

```
In [1]: from sympy import Symbol, symbols

In [2]: x = Symbol('x')

In [3]: y, z = symbols('y z')

In [4]: x*x+y
Out[4]: 'x**2 + y'
```

The `Symbol` function handles single symbols, while `symbols` can initialize several symbols simultaneously. The string argument might seem redundant, however, this represents the *label* displayed using print functions, which is neat to control. In addition, key word arguments can be sent to the symbol functions, flagging variables as e.g. positive, real, etc.

```
In [1]: from sympy import Symbol, symbols, im

In [2]: x2 = Symbol('x^2', real=True, positive=True) #Flagged as real. Note the label.

In [3]: y, z = symbols('y z') #Not flagged as real

In [4]: x2+y #x2 is printed more nicely given a describing label
Out[4]: 'x^2 + y'

In [5]: im(z) #Imaginary part cannot be assumed to be anything.
Out[5]: 'im(z)'

In [6]: im(x2) #Flagged as real, the imaginary part is zero.
Out[6]: 0
```

C.1.2 Exporting C++ and Latex Code

Exporting code is extremely simple: SymPy functions exist in the `sympy.printing` module, which simply takes a SymPy expression on input and returns the requested code-style equivalent. Consider the following example

```
In [1]: from sympy import symbols, printing, exp

In [2]: x, x2 = symbols('x x^2')

In [3]: printing.ccode(x*x*x*x*exp(-x2*x))
Out[3]: 'pow(x, 4)*exp(-x*x^2)'

In [4]: printing.ccode(x*x*x*x)
Out[4]: 'pow(x, 4)'

In [5]: print printing.latex(x*x*x*x*exp(-x2))
\frac{x^{4}}{e^{x^{2}}}
```

The following expression is the direct output from line five compiled in Latex

$$\frac{x^4}{e^{x^2}}$$

C.1.3 Calculating Derivatives

The $2s$ orbital from hydrogen (not normalized) is chosen as an example for this section

$$\phi_{2s}(\vec{r}) = (Zr - 2)e^{-\frac{1}{2}Zr} \tag{C.1}$$

$$r^2 = x^2 + y^2 + z^2 \tag{C.2}$$

Calculating the gradients and Laplacian is very simply by using the `sympy.diff` function

```
In [1]: from sympy import symbols, diff, exp, sqrt

In [2]: x, y, z, Z = symbols('x y z Z')

In [3]: r = sqrt(x*x + y*y + z*z)

In [4]: r
Out[4]: '(x**2 + y**2 + z**2)**(1/2)'

In [5]: phi = (Z*r - 2)*exp(-Z*r/2)

In [6]: phi
Out[6]: '(Z*(x**2 + y**2 + z**2)**(1/2) - 2)*exp(-Z*(x**2 + y**2 + z**2)**(1/2)/2)'

In [7]: diff(phi, x)
Out[7]: '-Z*x*(Z*(x**2 + y**2 + z**2)**(1/2) - 2)*exp(-Z*(x**2 + y**2 + z**2)**(1/2)/2)
    /(2*(x**2 + y**2 + z**2)**(1/2)) + Z*x*exp(-Z*(x**2 + y**2 + z**2)**(1/2)/2)/(x**2 +
    y**2 + z**2)**(1/2)'
```

Now, this looks like a nightmare. However, SymPy has great support for simplifying expressions through factorization, collecting, substituting etc. The following code demonstrated this quite nicely

```
...

In [6]: phi
Out[6]: '(Z*(x**2 + y**2 + z**2)**(1/2) - 2)*exp(-Z*(x**2 + y**2 + z**2)**(1/2)/2)'

In [7]: from sympy import factor, Symbol, printing

In [8]: R = Symbol('r') #Creates a symbolic equivalent of the mathematical r

In [9]: diff(phi, x).factor() #Factors out common factors
Out[9]: '-Z*x*(Z*(x**2 + y**2 + z**2)**(1/2) - 4)*exp(-Z*(x**2 + y**2 + z**2)**(1/2)/2)
    /(2*(x**2 + y**2 + z**2)**(1/2))'

In [10]: diff(phi, x).factor().subs(r, R) #replaces (x^2 + y^2 + z^2)^(1/2) with r
Out[10]: '-Z*x*(Z*r - 4)*exp(-Z*r/2)/(2*r)'

In [11]: print printing.latex(diff(phi, x).factor().subs(r, R))
- \frac{Z x \left(Z r -4\right)}{2 r e^{\frac{1}{2} Z r}}
```

This version of the expression is much more satisfying to the eye. The output from line 11 compiled in Latex is

$$-\frac{Zx\left(Zr-4\right)}{2re^{\frac{1}{2}Zr}}$$

SymPy has a general method for simplifying expressions `sympy.simplify`, however, this function is extremely slow and does not behave well on general expressions. SymPy is still young, so nothing can be expected to work perfectly. Moreover, in contrast to *Wolfram Alpha* and *Mathematica*, SymPy is open source, which means that much of the work, if not all of the work, is done by ordinary people on their spare time. The ill behaving simplify function is not really a great loss; full control for a Python programmer is never considered a bad thing, whether it is enforced or not.

Estimating the Laplacian is just a matter of summing double derivatives

```
...

In [12]: (diff(diff(phi, x), x) +
    ....:  diff(diff(phi, y), y) +
    ....:  diff(diff(phi, z), z)).factor().subs(r, R)
Out[12]: 'Z*(Z**2*x**2 + Z**2*y**2 + Z**2*z**2 - 10*Z*r + 16)*exp(-Z*r/2)/(4*r)'

In [13]: (diff(diff(phi, x), x) + #Not quite satisfying.
    ....:  diff(diff(phi, y), y) + #Let's collect the 'Z' terms.
    ....:  diff(diff(phi, z), z)).factor().collect(Z).subs(r, R)
Out[13]: 'Z*(Z**2*(x**2 + y**2 + z**2) - 10*Z*r + 16)*exp(-Z*r/2)/(4*r)'

In [14]: (diff(diff(phi, x), x) + #Still not satisfying.
    ....:  diff(diff(phi, y), y) + #The r^2 terms needs to be substituted as well.
    ....:  diff(diff(phi, z), z)).factor().collect(Z).subs(r, R).subs(r**2, R**2)
Out[14]: 'Z*(Z**2*r**2 - 10*Z*r + 16)*exp(-Z*r/2)/(4*r)'

In [15]: (diff(diff(phi, x), x) + #Let's try to factorize once more.
    ....:  diff(diff(phi, y), y) +
    ....:  diff(diff(phi, z), z)).factor().collect(Z).subs(r, R).subs(r**2, R**2).factor()
Out[15]: 'Z*(Z*r - 8)*(Z*r - 2)*exp(-Z*r/2)/(4*r)'
```

Getting the right factorization may come across as tricky, but with minimal training this poses no real problems.

C.2 Using the auto-generation Script

The superclass `orbitalsGenerator` aims to serve as an interface with the QMC C++ `BasisFunctions` class, automatically generating the C++ code containing all the implementations of the derivatives for the given single-particle states. The single-particle states are implemented in the generator by subclasses overloading system specific virtual functions which will be described in the following sections.

C.2.1 Generating Latex code

The following methods are user-implemented functions used to calculate the expressions which are in turn automagically converted to Latex code. Once they are implemented, the following code can be executed in order to create the latex output

```
orbitalSet = HO_3D.HOOrbitals3D(N=40) #Creating a 3D harm. osc. object
orbitalSet.closedFormify()
orbitalSet.TeXToFile(outPath)
```

The constructor

The superclass constructor takes on input the maximum number of particles for which expressions should be generated and the name of the orbital set, e.g. `hydrogenic`. Calling a superclass constructor from a subclass constructor is done in the following way

```
class hydrogenicOrbitals(orbitalGenerator):

    def __init__(self, N):

        super(hydrogenicOrbitals, self).__init__(N, "hydrogenic")
        #...
```

makeStateMap

This function takes care of the mapping of a set of quantum numbers, e.g. nlm to a specific index i. The Python dictionary `self.stateMap` must be filled with values for every unique set of quantum numbers (not counting spin) in order for the Latex and C++ files to be created successfully. For the three-dimensional harmonic oscillator wave functions, the state map looks like this

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
n_x	0	0	0	1	0	0	0	1	1	2	0	0	0	0	1	1	1	2	2	3
n_y	0	0	1	0	0	1	2	0	1	0	0	1	2	3	0	1	2	0	1	0
n_z	0	1	0	0	2	1	0	1	0	0	3	2	1	0	2	1	0	1	0	0

setUpOrbitals

Within this function, the orbital elements corresponding to the quantum number mappings made in `makeStateMap` needs to be implemented in a matching order. The quantum numbers from `self.stateMap` are calculated prior to this function being called, and can thus be accessed in case they are needed, as is the case for the n-dependent exponential factor of the hydrogen-like orbitals.

The i'th orbital needs to be implemented in `self.orbitals[i]`, using the `x`, `y` and `z` variables defined in the superclass. For the three-dimensional harmonic oscillator, the function is simply

```
def setupOrbitals(self):

    for i, stateMap in self.stateMap.items():
        nx, ny, nz = stateMap

        self.orbitals[i] = self.Hx[nx]*self.Hy[ny]*self.Hz[nz]*self.expFactor
```

where `self.Hx` and the exponential factor are implemented in the constructor. After the orbitals are created, the gradients and Laplacians cam be calculated by calling the `closedFormify()` function, however, unless the following member function is implemented, they are going to look messy.

simplifyLocal

As demonstrated in the previous example, SymPy expressions are messy when they are fresh out of the derivative functions. Since every system needs to be treated differently when it comes to cleaning up their expressions, this function is available. For hydrogen-like wave functions, the introductory example's strategy can be applied up to the level of neon. Going higher will require more advanced strategies for cleaning up the expressions.

The expression and the corresponding set of quantum numbers are given on input. In addition, there is an input argument `subs`, which if set to false should make the function return the expression in terms of `x`, `y` and `z` without substituting e.g. $x^2 + y^2 = r^2$.

genericFactor

The method serves as convenient function for describing generic parts of the expressions, e.g. the exponentials, which are often reused. A set of quantum numbers are supplied on input in case the generic expression depends on these. In addition, a flag `basic` is supplied on input, which if set to true should, as in the simplify function, return the generic factor in Cartesian coordinates. This generic factor can

then easily be taken out of the Latex expressions and mentioned in the caption in order to clean up the expression tables.

__str__

This method is invoked by calling `str(obj)` on an arbitrary Python object `obj`. In the case of the orbital generator class, this string will serve as an introductory text to the latex output.

C.2.2 Generating C++ code

A class `CPPbasis` is constructed to supplement the orbitals generator class. This objects holds the empty shells of the C++ constructors and implementations. After the functions described in this section are implemented, the following code can be executed to generate the C++ files

```
orbitalSet = HO_3D.HOOrbitals3D(N=40) #Creating a 3D harm. osc. object
orbitalSet.closedFormify()
orbitalSet.TeXToFile(outPath)
orbitalSet.CPPToFile(outPath)
```

initCPPbasis

Sets up the variables in the `CPPbasis` object needed in order to construct the C++ file, such as the dimension, the name, the constructor input variables and the C++ class members. The following function is the implementation for the two-dimensional harmonic oscillator

```
def initCPPbasis(self):

    self.cppBasis.dim = 2

    self.cppBasis.setName(self.name)

    self.cppBasis.setConstVars('double* k',          #sqrt(k2)
                               'double* k2',         #scaled oscillator freq.
                               'double* exp_factor') #The exponential

    self.cppBasis.setMembers('double* k',
                             'double* k2',
                             'double* exp_factor',
                             'double H',             #The Hermite polynomial part
                             'double x',
                             'double y',
                             'double x2',            #Squared Cartesian coordinates
                             'double y2')
```

getCPre and getCreturn

The empty shell of the `BasisFunctions::eval` functions in the `CPPbasis` class is implemented as below

```
self.evalShell = """
double __name__::eval(const Walker* walker, int i) {

__necessities__

    //__simpleExpr__
```

```
7
8  __preCalc__
9      return __return__
10
11 }
12 """
```

where `__preCalc__` is a generated C++ expression returned from `getCpre()`, and `__return__` is the returned C++ expression from `getCreturn()`. The commented `__simpleExpr__` will be replaced by the expression in nicely formatted SymPy output code. `__necessities__` is automatically detected by the script, and represents the Cartesian variable expressions needed by the expressions.

The functions take a SymPy generated expression on input, i.e. an orbital, gradient or Laplacian, and the corresponding index of the expression i. The reason these functions are split into a precalculation and a return expression is purely cosmetic. Consider the following example output for the hydrogen-like wave functions:

```
1  double dell_hydrogenic_9_y::eval(const Walker* walker, int i) {
2
3      y = walker->r(i, 1);
4      z = walker->r(i, 2);
5
6      z2 = z*z;
7
8      //-y*(k*(-r^2 + 3*z^2) + 6*r)*exp(-k*r/3)/(3*r)
9
10     psi = -y*((*k)*(-walker->get_r_i2(i) + 3*z2) + 6*walker->get_r_i(i))/(3*walker->
           get_r_i(i));
11     return psi*(*exp_factor);
12
13 }
```

The `*exp_factor` is the precalculated $n = 3$ exponential which is then simply multiplied by the non-exponential terms before being returned. The commented line is a clean version of the full expression. The required Cartesian components are retrieved prior to the evaluation.

The full implementation of `getCpre()` and `getCreturn()` for the hydrogen-like wave functions are given below

```
1  def getCReturn(self, expr, i):
2       return "psi*(*exp_factor);"
3
4  def getCPre(self, expr, i):
5       qNums = self.stateMap[i]
6       return "    psi = %s;" % printing.ccode(expr/self.genericFactor(qNums))
```

makeOrbConstArg

Loading the generated `BasisFunctions` objects into the `Orbitals` object in the QMC code is rather a dull job, and is not designed to be done manually. The function `makeOrbConstArg` is designed to automate this process. This is best demonstrated by an example: Consider the following constructor of the hydrogen-like wave function's orbital class

```
1  basis_functions[0] = new hydrogenic_0(k, k2, exp_factor_n1);
2  basis_functions[1] = new hydrogenic_1(k, k2, exp_factor_n2);
3  //...
4  basis_functions[5] = new hydrogenic_5(k, k2, exp_factor_n3);
5  //..
```

```
basis_functions[14] = new hydrogenic_14(k, k2, exp_factor_n4);
//..
basis_functions[17] = new hydrogenic_17(k, k2, exp_factor_n4);
```

where `exp_factor_nk` represents $\exp(-Zr/k)$, which is saved as a pointer reference for reasons explained in Section 4.6.4. The same procedure is applied to the gradients and the Laplacians as well, leaving a total of 90 sequential initializations. Everything needed in order to auto-generate the code is the following implementation

```
def makeOrbConstArgs(self, args, i):
    n = self.stateMap[i][0]
    args = args.replace('exp_factor', 'exp_fa-ctor_n%d' % n)
    return args
```

which ensures that the input arguments to e.g. element 1 is `(k, k2, exp_factor_n2)`, since the single-particle orbital `self.phi[1]` has a principle quantum number $n = 2$. The input argument `args` is the default constructor arguments set up the the `initCPPbasis`, and is in the case of hydrogen-like wave functions `(k, k2, exp_factor)`.

The tables listed in Appendix D, E and F are all generated within seconds using this framework. The generated C++ code for these span 8975 lines not counting blank ones.

D

Harmonic Oscillator Orbitals 2D

Orbitals are constructed in the following fashion:

$$\phi(\vec{r})_{n_x,n_y} = H_{n_x}(kx)H_{n_y}(ky)e^{-\frac{1}{2}k^2r^2}$$

where $k = \sqrt{\omega\alpha}$, with ω being the oscillator frequency and α being the variational parameter.

$H_0(kx)$	1
$H_1(kx)$	$2kx$
$H_2(kx)$	$4k^2x^2 - 2$
$H_3(kx)$	$8k^3x^3 - 12kx$
$H_4(kx)$	$16k^4x^4 - 48k^2x^2 + 12$
$H_5(kx)$	$32k^5x^5 - 160k^3x^3 + 120kx$
$H_6(kx)$	$64k^6x^6 - 480k^4x^4 + 720k^2x^2 - 120$
$H_0(ky)$	1
$H_1(ky)$	$2ky$
$H_2(ky)$	$4k^2y^2 - 2$
$H_3(ky)$	$8k^3y^3 - 12ky$
$H_4(ky)$	$16k^4y^4 - 48k^2y^2 + 12$
$H_5(ky)$	$32k^5y^5 - 160k^3y^3 + 120ky$
$H_6(ky)$	$64k^6y^6 - 480k^4y^4 + 720k^2y^2 - 120$

Table D.1: Hermite polynomials used to construct orbital functions

$\phi_0 \to \phi_{0,0}$	
$\phi(\vec{r})$	1
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2 x$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2 y$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 2\right)$

Table D.2: Orbital expressions HOOrbitals : 0, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_1 \to \phi_{0,1}$	
$\phi(\vec{r})$	y
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2 xy$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\left(ky - 1\right)\left(ky + 1\right)$
$\nabla^2\phi(\vec{r})$	$k^2 y\left(k^2r^2 - 4\right)$

Table D.3: Orbital expressions HOOrbitals : 0, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_2 \to \phi_{1,0}$	
$\phi(\vec{r})$	x
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\left(kx - 1\right)\left(kx + 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2 xy$
$\nabla^2\phi(\vec{r})$	$k^2 x\left(k^2r^2 - 4\right)$

Table D.4: Orbital expressions HOOrbitals : 1, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_3 \to \phi_{0,2}$	
$\phi(\vec{r})$	$2k^2y^2 - 1$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2 x\left(2k^2y^2 - 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2 y\left(2k^2y^2 - 5\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 6\right)\left(2k^2y^2 - 1\right)$

Table D.5: Orbital expressions HOOrbitals : 0, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_4 \to \phi_{1,1}$	
$\phi(\vec{r})$	xy
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-y\left(kx - 1\right)\left(kx + 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-x\left(ky - 1\right)\left(ky + 1\right)$
$\nabla^2\phi(\vec{r})$	$k^2 xy\left(k^2r^2 - 6\right)$

Table D.6: Orbital expressions HOOrbitals : 1, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_5 \rightarrow \phi_{2,0}$	
$\phi(\vec{r})$	$2k^2x^2 - 1$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2x\left(2k^2x^2 - 5\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2y\left(2k^2x^2 - 1\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 6\right)\left(2k^2x^2 - 1\right)$

Table D.7: Orbital expressions HOOrbitals : 2, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_6 \rightarrow \phi_{0,3}$	
$\phi(\vec{r})$	$y\left(2k^2y^2 - 3\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2y^2 - 3\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-2k^4y^4 + 9k^2y^2 - 3$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2 - 8\right)\left(2k^2y^2 - 3\right)$

Table D.8: Orbital expressions HOOrbitals : 0, 3. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_7 \rightarrow \phi_{1,2}$	
$\phi(\vec{r})$	$x\left(2k^2y^2 - 1\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\left(kx - 1\right)\left(kx + 1\right)\left(2k^2y^2 - 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2y^2 - 5\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2 - 8\right)\left(2k^2y^2 - 1\right)$

Table D.9: Orbital expressions HOOrbitals : 1, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_8 \rightarrow \phi_{2,1}$	
$\phi(\vec{r})$	$y\left(2k^2x^2 - 1\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2x^2 - 5\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\left(ky - 1\right)\left(ky + 1\right)\left(2k^2x^2 - 1\right)$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2 - 8\right)\left(2k^2x^2 - 1\right)$

Table D.10: Orbital expressions HOOrbitals : 2, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_9 \rightarrow \phi_{3,0}$	
$\phi(\vec{r})$	$x\left(2k^2x^2 - 3\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-2k^4x^4 + 9k^2x^2 - 3$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2x^2 - 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2 - 8\right)\left(2k^2x^2 - 3\right)$

Table D.11: Orbital expressions HOOrbitals : 3, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{10} \to \phi_{0,4}$	
$\phi(\vec{r})$	$4k^4y^4 - 12k^2y^2 + 3$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2x\left(4k^4y^4 - 12k^2y^2 + 3\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2y\left(4k^4y^4 - 28k^2y^2 + 27\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 10\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$

Table D.12: Orbital expressions HOOrbitals : 0, 4. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{11} \to \phi_{1,3}$	
$\phi(\vec{r})$	$xy\left(2k^2y^2 - 3\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-y\left(kx - 1\right)\left(kx + 1\right)\left(2k^2y^2 - 3\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-x\left(2k^4y^4 - 9k^2y^2 + 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2xy\left(k^2r^2 - 10\right)\left(2k^2y^2 - 3\right)$

Table D.13: Orbital expressions HOOrbitals : 1, 3. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{12} \to \phi_{2,2}$	
$\phi(\vec{r})$	$\left(2k^2x^2 - 1\right)\left(2k^2y^2 - 1\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2x\left(2k^2x^2 - 5\right)\left(2k^2y^2 - 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2y\left(2k^2x^2 - 1\right)\left(2k^2y^2 - 5\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 10\right)\left(2k^2x^2 - 1\right)\left(2k^2y^2 - 1\right)$

Table D.14: Orbital expressions HOOrbitals : 2, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{13} \to \phi_{3,1}$	
$\phi(\vec{r})$	$xy\left(2k^2x^2 - 3\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-y\left(2k^4x^4 - 9k^2x^2 + 3\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-x\left(ky - 1\right)\left(ky + 1\right)\left(2k^2x^2 - 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2xy\left(k^2r^2 - 10\right)\left(2k^2x^2 - 3\right)$

Table D.15: Orbital expressions HOOrbitals : 3, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{14} \to \phi_{4,0}$	
$\phi(\vec{r})$	$4k^4x^4 - 12k^2x^2 + 3$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2x\left(4k^4x^4 - 28k^2x^2 + 27\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2y\left(4k^4x^4 - 12k^2x^2 + 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 10\right)\left(4k^4x^4 - 12k^2x^2 + 3\right)$

Table D.16: Orbital expressions HOOrbitals : 4, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{15} \to \phi_{0,5}$	
$\phi(\vec{r})$	$y\left(4k^4y^4 - 20k^2y^2 + 15\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(4k^4y^4 - 20k^2y^2 + 15\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-4k^6y^6 + 40k^4y^4 - 75k^2y^2 + 15$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2 - 12\right)\left(4k^4y^4 - 20k^2y^2 + 15\right)$

Table D.17: Orbital expressions HOOrbitals : 0, 5. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{16} \to \phi_{1,4}$	
$\phi(\vec{r})$	$x\left(4k^4y^4 - 12k^2y^2 + 3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\left(kx - 1\right)\left(kx + 1\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(4k^4y^4 - 28k^2y^2 + 27\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2 - 12\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$

Table D.18: Orbital expressions HOOrbitals : 1, 4. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{17} \to \phi_{2,3}$	
$\phi(\vec{r})$	$y\left(2k^2x^2 - 1\right)\left(2k^2y^2 - 3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(2k^2x^2 - 5\right)\left(2k^2y^2 - 3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-\left(2k^2x^2 - 1\right)\left(2k^4y^4 - 9k^2y^2 + 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2 - 12\right)\left(2k^2x^2 - 1\right)\left(2k^2y^2 - 3\right)$

Table D.19: Orbital expressions HOOrbitals : 2, 3. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{18} \to \phi_{3,2}$	
$\phi(\vec{r})$	$x\left(2k^2x^2 - 3\right)\left(2k^2y^2 - 1\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\left(2k^2y^2 - 1\right)\left(2k^4x^4 - 9k^2x^2 + 3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(2k^2x^2 - 3\right)\left(2k^2y^2 - 5\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2 - 12\right)\left(2k^2x^2 - 3\right)\left(2k^2y^2 - 1\right)$

Table D.20: Orbital expressions HOOrbitals : 3, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{19} \to \phi_{4,1}$	
$\phi(\vec{r})$	$y\left(4k^4x^4 - 12k^2x^2 + 3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(4k^4x^4 - 28k^2x^2 + 27\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-\left(ky - 1\right)\left(ky + 1\right)\left(4k^4x^4 - 12k^2x^2 + 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2 - 12\right)\left(4k^4x^4 - 12k^2x^2 + 3\right)$

Table D.21: Orbital expressions HOOrbitals : 4, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{20} \to \phi_{5,0}$	
$\phi(\vec{r})$	$x\left(4k^4x^4 - 20k^2x^2 + 15\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-4k^6x^6 + 40k^4x^4 - 75k^2x^2 + 15$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(4k^4x^4 - 20k^2x^2 + 15\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2 - 12\right)\left(4k^4x^4 - 20k^2x^2 + 15\right)$

Table D.22: Orbital expressions HOOrbitals : 5, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{21} \to \phi_{0,6}$	
$\phi(\vec{r})$	$8k^6y^6 - 60k^4y^4 + 90k^2y^2 - 15$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2x\left(8k^6y^6 - 60k^4y^4 + 90k^2y^2 - 15\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2y\left(8k^6y^6 - 108k^4y^4 + 330k^2y^2 - 195\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 14\right)\left(8k^6y^6 - 60k^4y^4 + 90k^2y^2 - 15\right)$

Table D.23: Orbital expressions HOOrbitals : 0, 6. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{22} \to \phi_{1,5}$	
$\phi(\vec{r})$	$xy\left(4k^4y^4 - 20k^2y^2 + 15\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-y\left(kx - 1\right)\left(kx + 1\right)\left(4k^4y^4 - 20k^2y^2 + 15\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-x\left(4k^6y^6 - 40k^4y^4 + 75k^2y^2 - 15\right)$
$\nabla^2\phi(\vec{r})$	$k^2xy\left(k^2r^2 - 14\right)\left(4k^4y^4 - 20k^2y^2 + 15\right)$

Table D.24: Orbital expressions HOOrbitals : 1, 5. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{23} \to \phi_{2,4}$	
$\phi(\vec{r})$	$\left(2k^2x^2 - 1\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2x\left(2k^2x^2 - 5\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2y\left(2k^2x^2 - 1\right)\left(4k^4y^4 - 28k^2y^2 + 27\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 14\right)\left(2k^2x^2 - 1\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$

Table D.25: Orbital expressions HOOrbitals : 2, 4. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{24} \to \phi_{3,3}$	
$\phi(\vec{r})$	$xy\left(2k^2x^2 - 3\right)\left(2k^2y^2 - 3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-y\left(2k^2y^2 - 3\right)\left(2k^4x^4 - 9k^2x^2 + 3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-x\left(2k^2x^2 - 3\right)\left(2k^4y^4 - 9k^2y^2 + 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2xy\left(k^2r^2 - 14\right)\left(2k^2x^2 - 3\right)\left(2k^2y^2 - 3\right)$

Table D.26: Orbital expressions HOOrbitals : 3, 3. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{25} \to \phi_{4,2}$	
$\phi(\vec{r})$	$\left(2k^2y^2-1\right)\left(4k^4x^4-12k^2x^2+3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2x\left(2k^2y^2-1\right)\left(4k^4x^4-28k^2x^2+27\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2y\left(2k^2y^2-5\right)\left(4k^4x^4-12k^2x^2+3\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2-14\right)\left(2k^2y^2-1\right)\left(4k^4x^4-12k^2x^2+3\right)$

Table D.27: Orbital expressions HOOrbitals : 4, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{26} \to \phi_{5,1}$	
$\phi(\vec{r})$	$xy\left(4k^4x^4-20k^2x^2+15\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-y\left(4k^6x^6-40k^4x^4+75k^2x^2-15\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-x\left(ky-1\right)\left(ky+1\right)\left(4k^4x^4-20k^2x^2+15\right)$
$\nabla^2\phi(\vec{r})$	$k^2xy\left(k^2r^2-14\right)\left(4k^4x^4-20k^2x^2+15\right)$

Table D.28: Orbital expressions HOOrbitals : 5, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{27} \to \phi_{6,0}$	
$\phi(\vec{r})$	$8k^6x^6-60k^4x^4+90k^2x^2-15$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2x\left(8k^6x^6-108k^4x^4+330k^2x^2-195\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2y\left(8k^6x^6-60k^4x^4+90k^2x^2-15\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2-14\right)\left(8k^6x^6-60k^4x^4+90k^2x^2-15\right)$

Table D.29: Orbital expressions HOOrbitals : 6, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

E

Harmonic Oscillator Orbitals 3D

Orbitals are constructed in the following fashion:

$$\phi(\vec{r})_{n_x,n_y,n_z} = H_{n_x}(kx)H_{n_y}(ky)H_{n_z}(kz)e^{-\frac{1}{2}k^2r^2}$$

where $k = \sqrt{\omega\alpha}$, with ω being the oscillator frequency and α being the variational parameter.

$H_0(kx)$	1
$H_1(kx)$	$2kx$
$H_2(kx)$	$4k^2x^2 - 2$
$H_3(kx)$	$8k^3x^3 - 12kx$
$H_0(ky)$	1
$H_1(ky)$	$2ky$
$H_2(ky)$	$4k^2y^2 - 2$
$H_3(ky)$	$8k^3y^3 - 12ky$
$H_0(kz)$	1
$H_1(kz)$	$2kz$
$H_2(kz)$	$4k^2z^2 - 2$
$H_3(kz)$	$8k^3z^3 - 12kz$

Table E.1: Hermite polynomials used to construct orbital functions

$\phi_0 \to \phi_{0,0,0}$	
$\phi(\vec{r})$	1
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2 x$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2 y$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-k^2 z$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 3\right)$

Table E.2: Orbital expressions HOOrbitals3D : 0, 0, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_1 \to \phi_{0,0,1}$	
$\phi(\vec{r})$	z
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2 xz$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2 yz$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\left(kz - 1\right)\left(kz + 1\right)$
$\nabla^2\phi(\vec{r})$	$k^2 z\left(k^2r^2 - 5\right)$

Table E.3: Orbital expressions HOOrbitals3D : 0, 0, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_2 \to \phi_{0,1,0}$	
$\phi(\vec{r})$	y
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2 xy$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\left(ky - 1\right)\left(ky + 1\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-k^2 yz$
$\nabla^2\phi(\vec{r})$	$k^2 y\left(k^2r^2 - 5\right)$

Table E.4: Orbital expressions HOOrbitals3D : 0, 1, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_3 \to \phi_{1,0,0}$	
$\phi(\vec{r})$	x
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\left(kx - 1\right)\left(kx + 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2 xy$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-k^2 xz$
$\nabla^2\phi(\vec{r})$	$k^2 x\left(k^2r^2 - 5\right)$

Table E.5: Orbital expressions HOOrbitals3D : 1, 0, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_4 \to \phi_{0,0,2}$	
$\phi(\vec{r})$	$4k^2z^2 - 2$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-2k^2 x\left(2k^2z^2 - 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-2k^2 y\left(2k^2z^2 - 1\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-2k^2 z\left(2k^2z^2 - 5\right)$
$\nabla^2\phi(\vec{r})$	$2k^2\left(k^2r^2 - 7\right)\left(2k^2z^2 - 1\right)$

Table E.6: Orbital expressions HOOrbitals3D : 0, 0, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_5 \rightarrow \phi_{0,1,1}$	
$\phi(\vec{r})$	yz
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2xyz$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-z\left(ky-1\right)\left(ky+1\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-y\left(kz-1\right)\left(kz+1\right)$
$\nabla^2\phi(\vec{r})$	$k^2yz\left(k^2r^2-7\right)$

Table E.7: Orbital expressions HOOrbitals3D : 0, 1, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_6 \rightarrow \phi_{0,2,0}$	
$\phi(\vec{r})$	$4k^2y^2-2$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-2k^2x\left(2k^2y^2-1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-2k^2y\left(2k^2y^2-5\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-2k^2z\left(2k^2y^2-1\right)$
$\nabla^2\phi(\vec{r})$	$2k^2\left(k^2r^2-7\right)\left(2k^2y^2-1\right)$

Table E.8: Orbital expressions HOOrbitals3D : 0, 2, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_7 \rightarrow \phi_{1,0,1}$	
$\phi(\vec{r})$	xz
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-z\left(kx-1\right)\left(kx+1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2xyz$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-x\left(kz-1\right)\left(kz+1\right)$
$\nabla^2\phi(\vec{r})$	$k^2xz\left(k^2r^2-7\right)$

Table E.9: Orbital expressions HOOrbitals3D : 1, 0, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_8 \rightarrow \phi_{1,1,0}$	
$\phi(\vec{r})$	xy
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-y\left(kx-1\right)\left(kx+1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-x\left(ky-1\right)\left(ky+1\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-k^2xyz$
$\nabla^2\phi(\vec{r})$	$k^2xy\left(k^2r^2-7\right)$

Table E.10: Orbital expressions HOOrbitals3D : 1, 1, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_9 \rightarrow \phi_{2,0,0}$	
$\phi(\vec{r})$	$4k^2x^2-2$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-2k^2x\left(2k^2x^2-5\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-2k^2y\left(2k^2x^2-1\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-2k^2z\left(2k^2x^2-1\right)$
$\nabla^2\phi(\vec{r})$	$2k^2\left(k^2r^2-7\right)\left(2k^2x^2-1\right)$

Table E.11: Orbital expressions HOOrbitals3D : 2, 0, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{10} \rightarrow \phi_{0,0,3}$	
$\phi(\vec{r})$	$z\left(2k^2z^2-3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2xz\left(2k^2z^2-3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2yz\left(2k^2z^2-3\right)$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-2k^4z^4+9k^2z^2-3$
$\nabla^2\phi(\vec{r})$	$k^2z\left(k^2r^2-9\right)\left(2k^2z^2-3\right)$

Table E.12: Orbital expressions HOOrbitals3D : 0, 0, 3. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{11} \rightarrow \phi_{0,1,2}$	
$\phi(\vec{r})$	$y\left(2k^2z^2-1\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(2k^2z^2-1\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-\left(ky-1\right)\left(ky+1\right)\left(2k^2z^2-1\right)$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-k^2yz\left(2k^2z^2-5\right)$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2-9\right)\left(2k^2z^2-1\right)$

Table E.13: Orbital expressions HOOrbitals3D : 0, 1, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{12} \rightarrow \phi_{0,2,1}$	
$\phi(\vec{r})$	$z\left(2k^2y^2-1\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2xz\left(2k^2y^2-1\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2yz\left(2k^2y^2-5\right)$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-\left(kz-1\right)\left(kz+1\right)\left(2k^2y^2-1\right)$
$\nabla^2\phi(\vec{r})$	$k^2z\left(k^2r^2-9\right)\left(2k^2y^2-1\right)$

Table E.14: Orbital expressions HOOrbitals3D : 0, 2, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{13} \rightarrow \phi_{0,3,0}$	
$\phi(\vec{r})$	$y\left(2k^2y^2-3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(2k^2y^2-3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-2k^4y^4+9k^2y^2-3$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-k^2yz\left(2k^2y^2-3\right)$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2-9\right)\left(2k^2y^2-3\right)$

Table E.15: Orbital expressions HOOrbitals3D : 0, 3, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{14} \rightarrow \phi_{1,0,2}$	
$\phi(\vec{r})$	$x\left(2k^2z^2-1\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\left(kx-1\right)\left(kx+1\right)\left(2k^2z^2-1\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(2k^2z^2-1\right)$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-k^2xz\left(2k^2z^2-5\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2-9\right)\left(2k^2z^2-1\right)$

Table E.16: Orbital expressions HOOrbitals3D : 1, 0, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{15} \rightarrow \phi_{1,1,1}$	
$\phi(\vec{r})$	xyz
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-yz\,(kx-1)\,(kx+1)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-xz\,(ky-1)\,(ky+1)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-xy\,(kz-1)\,(kz+1)$
$\nabla^2\phi(\vec{r})$	$k^2xyz\left(k^2r^2-9\right)$

Table E.17: Orbital expressions HOOrbitals3D : 1, 1, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{16} \rightarrow \phi_{1,2,0}$	
$\phi(\vec{r})$	$x\left(2k^2y^2-1\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\left(kx-1\right)\left(kx+1\right)\left(2k^2y^2-1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2y^2-5\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-k^2xz\left(2k^2y^2-1\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2-9\right)\left(2k^2y^2-1\right)$

Table E.18: Orbital expressions HOOrbitals3D : 1, 2, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{17} \rightarrow \phi_{2,0,1}$	
$\phi(\vec{r})$	$z\left(2k^2x^2-1\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2xz\left(2k^2x^2-5\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2yz\left(2k^2x^2-1\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\left(kz-1\right)\left(kz+1\right)\left(2k^2x^2-1\right)$
$\nabla^2\phi(\vec{r})$	$k^2z\left(k^2r^2-9\right)\left(2k^2x^2-1\right)$

Table E.19: Orbital expressions HOOrbitals3D : 2, 0, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{18} \rightarrow \phi_{2,1,0}$	
$\phi(\vec{r})$	$y\left(2k^2x^2-1\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2x^2-5\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\left(ky-1\right)\left(ky+1\right)\left(2k^2x^2-1\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-k^2yz\left(2k^2x^2-1\right)$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2-9\right)\left(2k^2x^2-1\right)$

Table E.20: Orbital expressions HOOrbitals3D : 2, 1, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{19} \rightarrow \phi_{3,0,0}$	
$\phi(\vec{r})$	$x\left(2k^2x^2-3\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-2k^4x^4+9k^2x^2-3$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2x^2-3\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-k^2xz\left(2k^2x^2-3\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2-9\right)\left(2k^2x^2-3\right)$

Table E.21: Orbital expressions HOOrbitals3D : 3, 0, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

F

Hydrogen Orbitals

Orbitals are constructed in the following fashion:

$$\phi(\vec{r})_{n,l,m} = L_{n-l-1}^{2l+1}\Big(\frac{2r}{n}k\Big)S_l^m(\vec{r})e^{-\frac{r}{n}k}$$

where n is the principal quantum number, $k = \alpha Z$ with Z being the nucleus charge and α being the variational parameter.

$$l = 0,\ 1,\ ...,\ (n-1)$$

$$m = -l,\ (-l+1),\ ...,\ (l-1),\ l$$

$\phi_0 \to \phi_{1,0,0}$	
$\phi(\vec{r})$	1
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kx}{r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{ky}{r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kz}{r}$
$\nabla^2\phi(\vec{r})$	$\frac{k(kr-2)}{r}$

Table F.1: Orbital expressions hydrogenicOrbitals : 1, 0, 0. Factor e^{-kr} is omitted.

$\phi_1 \to \phi_{2,0,0}$	
$\phi(\vec{r})$	$kr-2$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kx(kr-4)}{2r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{ky(kr-4)}{2r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kz(kr-4)}{2r}$
$\nabla^2\phi(\vec{r})$	$\frac{k(kr-8)(kr-2)}{4r}$

Table F.2: Orbital expressions hydrogenicOrbitals : 2, 0, 0. Factor $e^{-\frac{1}{2}kr}$ is omitted.

$\phi_2 \to \phi_{2,1,0}$	
$\phi(\vec{r})$	z
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kxz}{2r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{kyz}{2r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$\frac{-kz^2+2r}{2r}$
$\nabla^2\phi(\vec{r})$	$\frac{kz(kr-8)}{4r}$

Table F.3: Orbital expressions hydrogenicOrbitals : 2, 1, 0. Factor $e^{-\frac{1}{2}kr}$ is omitted.

$\phi_3 \to \phi_{2,1,1}$	
$\phi(\vec{r})$	x
$\vec{i} \cdot \nabla\phi(\vec{r})$	$\frac{-kx^2+2r}{2r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{kxy}{2r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kxz}{2r}$
$\nabla^2\phi(\vec{r})$	$\frac{kx(kr-8)}{4r}$

Table F.4: Orbital expressions hydrogenicOrbitals : 2, 1, 1. Factor $e^{-\frac{1}{2}kr}$ is omitted.

$\phi_4 \rightarrow \phi_{2,1,-1}$	
$\phi(\vec{r})$	y
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kxy}{2r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$\frac{-ky^2+2r}{2r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kyz}{2r}$
$\nabla^2\phi(\vec{r})$	$\frac{ky(kr-8)}{4r}$

Table F.5: Orbital expressions hydrogenicOrbitals : 2, 1, -1. Factor $e^{-\frac{1}{2}kr}$ is omitted.

$\phi_5 \rightarrow \phi_{3,0,0}$	
$\phi(\vec{r})$	$2k^2r^2 - 18kr + 27$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kx\left(2k^2r^2-30kr+81\right)}{3r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{ky\left(2k^2r^2-30kr+81\right)}{3r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kz\left(2k^2r^2-30kr+81\right)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{k(kr-18)\left(2k^2r^2-18kr+27\right)}{9r}$

Table F.6: Orbital expressions hydrogenicOrbitals : 3, 0, 0. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_6 \rightarrow \phi_{3,1,0}$	
$\phi(\vec{r})$	$z\,(kr-6)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kxz(kr-9)}{3r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{kyz(kr-9)}{3r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$\frac{3kr^2-kz^2(kr-9)-18r}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{kz(kr-18)(kr-6)}{9r}$

Table F.7: Orbital expressions hydrogenicOrbitals : 3, 1, 0. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_7 \rightarrow \phi_{3,1,1}$	
$\phi(\vec{r})$	$x\,(kr-6)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$\frac{3kr^2-kx^2(kr-9)-18r}{3r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{kxy(kr-9)}{3r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kxz(kr-9)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{kx(kr-18)(kr-6)}{9r}$

Table F.8: Orbital expressions hydrogenicOrbitals : 3, 1, 1. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_8 \rightarrow \phi_{3,1,-1}$	
$\phi(\vec{r})$	$y\,(kr-6)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\frac{kxy(kr-9)}{3r}$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$\frac{3kr^2-ky^2(kr-9)-18r}{3r}$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-\frac{kyz(kr-9)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{ky(kr-18)(kr-6)}{9r}$

Table F.9: Orbital expressions hydrogenicOrbitals : 3, 1, -1. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_9 \rightarrow \phi_{3,2,0}$	
$\phi(\vec{r})$	$-r^2+3z^2$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\frac{x\left(k\left(-r^2+3z^2\right)+6r\right)}{3r}$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-\frac{y\left(k\left(-r^2+3z^2\right)+6r\right)}{3r}$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-\frac{z\left(k\left(-r^2+3z^2\right)-12r\right)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{k\left(-r^2+3z^2\right)(kr-18)}{9r}$

Table F.10: Orbital expressions hydrogenicOrbitals : 3, 2, 0. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_{10} \rightarrow \phi_{3,2,1}$	
$\phi(\vec{r})$	xz
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\frac{z\left(kx^2-3r\right)}{3r}$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-\frac{kxyz}{3r}$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-\frac{x\left(kz^2-3r\right)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{kxz(kr-18)}{9r}$

Table F.11: Orbital expressions hydrogenicOrbitals : 3, 2, 1. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_{11} \rightarrow \phi_{3,2,-1}$	
$\phi(\vec{r})$	yz
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\frac{kxyz}{3r}$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-\frac{z\left(ky^2-3r\right)}{3r}$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-\frac{y\left(kz^2-3r\right)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{kyz(kr-18)}{9r}$

Table F.12: Orbital expressions hydrogenicOrbitals : 3, 2, -1. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_{12} \to \phi_{3,2,2}$	
$\phi(\vec{r})$	$x^2 - y^2$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{x\left(k\left(x^2-y^2\right)-6r\right)}{3r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{y\left(k\left(x^2-y^2\right)+6r\right)}{3r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kz\left(x^2-y^2\right)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{k\left(x^2-y^2\right)(kr-18)}{9r}$

Table F.13: Orbital expressions hydrogenicOrbitals : 3, 2, 2. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_{13} \to \phi_{3,2,-2}$	
$\phi(\vec{r})$	xy
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{y\left(kx^2-3r\right)}{3r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{x\left(ky^2-3r\right)}{3r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kxyz}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{kxy(kr-18)}{9r}$

Table F.14: Orbital expressions hydrogenicOrbitals : 3, 2, -2. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_{14} \to \phi_{4,0,0}$	
$\phi(\vec{r})$	$k^3r^3 - 24k^2r^2 + 144kr - 192$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kx\left(k^3r^3-36k^2r^2+336kr-768\right)}{4r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{ky\left(k^3r^3-36k^2r^2+336kr-768\right)}{4r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kz\left(k^3r^3-36k^2r^2+336kr-768\right)}{4r}$
$\nabla^2\phi(\vec{r})$	$\frac{k(kr-32)\left(k^3r^3-24k^2r^2+144kr-192\right)}{16r}$

Table F.15: Orbital expressions hydrogenicOrbitals : 4, 0, 0. Factor $e^{-\frac{1}{4}kr}$ is omitted.

$\phi_{15} \to \phi_{4,1,0}$	
$\phi(\vec{r})$	$z\left(k^2r^2 - 20kr + 80\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kxz(kr-20)(kr-8)}{4r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{kyz(kr-20)(kr-8)}{4r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$\frac{4k^2r^3-80kr^2-kz^2(kr-20)(kr-8)+320r}{4r}$
$\nabla^2\phi(\vec{r})$	$\frac{kz(kr-32)\left(k^2r^2-20kr+80\right)}{16r}$

Table F.16: Orbital expressions hydrogenicOrbitals : 4, 1, 0. Factor $e^{-\frac{1}{4}kr}$ is omitted.

$\phi_{16} \to \phi_{4,1,1}$	
$\phi(\vec{r})$	$x\left(k^2r^2 - 20kr + 80\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$\frac{4k^2r^3 - 80kr^2 - kx^2(kr-20)(kr-8) + 320r}{4r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{kxy(kr-20)(kr-8)}{4r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kxz(kr-20)(kr-8)}{4r}$
$\nabla^2\phi(\vec{r})$	$\frac{kx(kr-32)\left(k^2r^2-20kr+80\right)}{16r}$

Table F.17: Orbital expressions hydrogenicOrbitals : 4, 1, 1. Factor $e^{-\frac{1}{4}kr}$ is omitted.

$\phi_{17} \to \phi_{4,1,-1}$	
$\phi(\vec{r})$	$y\left(k^2r^2 - 20kr + 80\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kxy(kr-20)(kr-8)}{4r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$\frac{4k^2r^3 - 80kr^2 - ky^2(kr-20)(kr-8) + 320r}{4r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kyz(kr-20)(kr-8)}{4r}$
$\nabla^2\phi(\vec{r})$	$\frac{ky(kr-32)\left(k^2r^2-20kr+80\right)}{16r}$

Table F.18: Orbital expressions hydrogenicOrbitals : 4, 1, -1. Factor $e^{-\frac{1}{4}kr}$ is omitted.

Bibliography

[1] S. Reimann, J. Høgberget, M. Hjorth-Jensen, and S. K. Bogner. In-medium similarity renormalization group approach studies of quantum dots. In preparation, 2013.

[2] Christoffer Hirth. Studies of quantum dots: Ab initio coupled-cluster analysis using OpenCL and GPU programming. Master's thesis, University of Oslo, 2012.

[3] Moskowitz Kalos. A new Look at Correlations in Atomic and Molecular Systems. Application of Fermion Monte Carlo Variational Method. *Int. J. Quant. Chem.*, **XX**:1107, 1981.

[4] S. Datta, S. A. Alexander, and R. L. Coldwell. Properties of selected diatomics using variational Monte Carlo methods. *J. Chem. Phys.*, **120**:3642, 2004.

[5] Matthias Degroote. Faddeev random phase approximation applied to molecules. *Eur. Phys. J. ST*, **218**:1, 2013.

[6] Harry Partridge. Near Hartree–Fock quality GTO basis sets for the first- and third-row atoms. *J. Chem. Phys.*, **90**:1043, 1989.

[7] A Badinski, P D Haynes, J R Trail, and R J Needs. Methods for calculating forces within quantum Monte Carlo simulations. *J. Phys.: Condens. Matter*, **22**:074202, 2010.

[8] Veronica K. B. Olsen. Full Configuration Interaction Simulation of Quantum Dots. Master's thesis, University of Oslo, 2012.

[9] Jørgen Høgberget. *Git Repository: LibBorealis*, 2013. http://www.github.com/jorgehog/QMC2.

[10] Murat Barisik and Ali Beskok. Equilibrium molecular dynamics studies on nanoscale-confined fluids. *Microfluid Nanofluid*, **11**(3):269, September 2011.

[11] Karl P. Travis and Keith E. Gubbins. Poiseuille flow of Lennard-Jones fluids in narrow slit pores. *J. Chem. Phys*, **112**:1984, 2000.

[12] Charles J. Murray. *The SUPERMEN*. New York: Wiley, 1997.

[13] Hans Petter Langtangen. *Python Scripting for Computational Science*. Springer, 3rd edition, 2008.

[14] Mark Lutz. *Programming Python - powerful object-oriented programming.* O'Reilly, 3rd edition, 2006.

[15] Hans Petter Langtangen. *A primer on scientific programming with Python.* Springer, Berlin; Heidelberg; New York, 2011.

[16] Gerard O'Regan. *A Brief History of Computing, Second Edition.* Springer, 2012.

[17] David Griffiths. *Introduction to Quantum Mechanics.* Pearson, 2nd edition edition, 2005.

[18] I. Shavitt and R. J. Bartlett. *Many-Body Methods in Chemistry and Physics.* Cambridge University Press, Cambridge, 2009.

[19] Sigve B. Skattum. Time Evolution of Quantum Dots. Master's thesis, University of Oslo, 2013.

[20] J. J. Mares et al. Periodic reactions and quantum diffusion. http://www.fzu.cz/ sestak/yyx/periodic%20reactions.pdf.

[21] B.L. Hammond, Jr. W. A. Lester, and P. J. Reynolds. *Monte Carlo Methods in Ab Initio Quantum Chemistry.* World Scientific Publishing Co., 1994.

[22] C. W Gardiner. *Handbook of stochastic methods for physics, chemistry, and the natural sciences.* Springer-Verlag, Berlin, 3rd edition, 2004.

[23] H. Risken and H. Haken. *The Fokker-Planck Equation: Methods of Solution and Applications Second Edition.* Springer, 1989.

[24] W. T Coffey, Y. P Kalmykov, and J. T Waldron. *The Langevin Equation: With Applications to Stochastic Problems in Physics, Chemistry, and Electrical Engineering.* . World Scientific, Singapore, 2004.

[25] C.P. Robert and G. Casella. *Monte Carlo Statistical Methods.* Springer Verlag, 2004.

[26] Morten Hjorth-Jensen. Computational Physics. 2010.

[27] J. J. Sakurai. *Modern Quantum Mechanics.* Addison-Wesley, New York, Revised ed edition, 1994.

[28] Lars Eivind Lervåg. VMC Calculations of Two-dimensional Quantum Dots. Master's thesis, University of Oslo, 2010.

[29] Daniel Andres Nissenbaum. *The stochastic gradient approximation: an application to li nanoclusters.* PhD thesis, Northeastern University, 2008.

[30] Claudia Filippi and C. J. Umrigar. Multiconfiguration wave function for quantum Monte Carlo calculations of first-row diatomic molecules. *J. Chem. Phys*, **105**:213, July 1996.

[31] G.H. Golub and C.F. Van Loan. *Matrix computations*, volume 3. Johns Hopkins Univ Press, 1996.

[32] A. Harju, B. Barbiellini, S. Siljamäki, R. M. Nieminen, and G. Ortiz. Stochastic Gradient Approximation: An Efficient Method to Optimize Many-Body Wave Functions. *Phys. Rev. Lett.*, page 1173, August 1997.

[33] Stefan Klein, Josien P. W. Pluim, Marius Staring, and Max A. Viergever. Adaptive Stochastic Gradient Descent Optimisation for Image Registration. *Int. J. Comput. Vision*, page 227, 2009.

[34] Jon Magne Leinaas. Non-Relativistic Quantum Mechanics. lecture notes FYS4110.

[35] H. Flyvbjerg and H. G. Petersen. Error estimates on averages of correlated data. *J. Chem. Phys*, **91**:461, 1989.

[36] David C. Lay. *Linear Algebra and its Applications*. Pearson, 4 edition, 2012.

[37] Jaime Fernández Rico, Rafael López, Ignacio Ema, and Guillermo Ramírez. Translation of real solid spherical harmonics. *Int. J. Quant. Chem.*, **113**:1544, 2013.

[38] T. Helgaker, P. Jørgensen, and J. Olsen. *Molecular Electronic-Structure Theory*. Wiley, Chichester, 2000.

[39] A. Kongkanand, K. Tvrdy, K. Takechi, M. Kuno, and P. V. Kamat. Quantum dot solar cells. Tuning photoresponse through size and shape control of CdSe-TiO2 architecture. *J. Am. Chem. Soc.*, **130**:4007, March 2008.

[40] G. Park, O.B. Shchekin, D.L. Huffaker, and D.G. Deppe. Low-threshold oxide-confined 1.3- micrometer quantum-dot laser. *IEEE Photon. Technol. Lett.*, **12**:230, 2000.

[41] E.T. Ben-Ari. Nanoscale quantum dots hold promise for cancer applications. *J. Natl. Cancer Inst.*, **90**:502, 2003.

[42] D. Loss and D. P. Vincenzo. Quantum computation with quantum dots. *Phys. Rev. A*, **57**:120, 1998.

[43] R. V. Shenoi, J. Hou, Y. Sharma, J. Shao, T. E. Vandervelde, and S. Krishna. Low strain quantum dots in a double well infrared detector. pages 708207–708207–6, 2008.

[44] Yang Min Wang. Coupled-Cluster Studies of Double Quantum Dots. Master's thesis, University of Oslo, 2011.

[45] M. Taut. Two electrons in an external oscillator potential: Particular analytic solutions of a Coulomb correlation problem. *Phys. Rev. A*, **48**:3561, Nov 1993.

[46] M. Pedersen Lohne, G. Hagen, M. Hjorth-Jensen, S. Kvaal, and F. Pederiva. *Ab initio* computation of the energies of circular quantum dots. *Phys. Rev. B*, **84**:115302, Sep 2011.

[47] Sarah Reimann. Quantum-mechanical systems in traps and Similarity Renormalization Group theory. Master's thesis, University of Oslo, 2013.

[48] E. Wigner. On the Interaction of Electrons in Metals. *Phys. Rev.*, **46**:1002, Dec 1934.

[49] F Cavaliere, U De Giovannini, M Sassetti, and B Kramer. Transport properties of quantum dots in the Wigner molecule regime. *New J. Phys.*, **11**:123004, 2009.

[50] Lang Zeng, W. Geist, W. Y. Ruan, C. J. Umrigar, and M. Y. Chou. Path to Wigner localization in circular quantum dots. *Phys. Rev. B*, **79**:235334, Jun 2009.

[51] Constantine Yannouleas and Uzi Landman. Symmetry breaking and Wigner molecules in few-electron quantum dots. *phys. stat. sol. (a)*, **203**:1160, 2006.

[52] S.A. Mikhailova and K. Ziegler. Floating Wigner molecules and possible phase transitions in quantum dots. *Eur. Phys. J. B*, **28**:117, 2002.

[53] N. Akman and M. Tomak. The Wigner molecule in a 2D quantum dot. *Physica E: Low-dimensional Systems and Nanostructures*, **4**:277, 1999.

[54] C. C. Grimes and G. Adams. Evidence for a Liquid-to-Crystal Phase Transition in a Classical, Two-Dimensional Sheet of Electrons. *Phys. Rev. Lett.*, **42**:795, Mar 1979.

[55] Jongsoo Yoon, C. C. Li, D. Shahar, D. C. Tsui, and M. Shayegan. Wigner Crystallization and Metal-Insulator Transition of Two-Dimensional Holes in GaAs at $B = 0$. *Phys. Rev. Lett.*, **82**:1744, Feb 1999.

[56] F. Bolton and U. Rössler. Classical model of a Wigner crystal in a quantum dot. *Superlattices and Microstructures*, **13**:139, 1993.

[57] V Fock. Bemerkung zum Virialsatz. *Z. Phys.*, **63**:855, November 1930.

[58] H. D. Young, R.A. Freedman, and L.A. Ford. *Sears and Zemansky's University Physics*. Pearson, Addison-Wesley, 12 edition, 2008.

[59] Adri C. T. van Duin, Siddharth Dasgupta, Francois Lorant, and William A. Goddard. ReaxFF: A Reactive Force Field for Hydrocarbons. *J. Phys. Chem. A*, **105**:9396, 2001.

[60] Conrad Sanderson. Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments. Technical report, NICTA, 2010.

[61] Håvard Sandsdalen. Variational Monte Carlo studies of Atoms. Master's thesis, University of Oslo, 2010.

[62] Aurel Bulgac and Michael McNeil Forbes. Use of the discrete variable representation basis in nuclear physics. *Phys. Rev. C*, **87**:051301, May 2013.

[63] A. Roggero, F. Pederiva, and A. Mukherjee. Quantum Monte Carlo with Coupled-Cluster wave functions. arXiv:1304.1549 [nucl-th].

[64] Matplotlib website, May 2013. http://matplotlib.org.

[65] PySide website, May 2013. http://qt-project.org/wiki/Category:LanguageBindings::PySide.

[66] SymPy website, May 2013. http://sympy.org.