Part I

Theory

1

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* leaded by *J. Robert Oppenheimer* [1].

1.1 Programming Languages

Programming is the art of writing computer programs. A program is a list of instructions for the computer often 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 a priori of 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, i.e. high - or low voltage. Every character, number, color, etc. is on the lowest level represented as a sequence of binary numbers referred to as *bits*. Programming languages, in other words, 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 were the most used languages throughout this thesis.

1.1.1 High-level Languages

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

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

languages [2].

High-level codes are often short and structured 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 together different programs which are meant to be run sequentially or in parallel. These short specific codes are referred to as *scripts*, and hence high-level languages designed for these tasks often are referred to as *scripting languages*[2, 3].

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 designed for optimized development time through very clean and rigorous coding syntax [4, 3].

To demonstrate the simplicity of Python, let us have a look at a simple implementation and execution of the following expression

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

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

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

For details regarding the basics of Python, I suggest reading Ref. [4], or more advanced topics in Ref. [2].

1.1.2 Low-level Languages

Scientific programming 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 e.g. linear algebra, the problem can be boiled down to solving $\mathbf{Ax} = \mathbf{b}$, however, the complexity lies in the dimensionality of the problem at hand. Matrix dimensions 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.

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 crash

[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 ones. However, for the purpose of this thesis I will not go into assembly languages, and keep the distinction at a higher level.

runtime 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 3.7.

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. [5]. The following code is a C++ implementation of the sum in Eq. 1.1.1:

```
1  //Sum100C++.cpp
2  #include <iostream>
3
4  int main(){
5
6          int S = 0;
7          for (int i = 1; i <= 100; i++){
8                  S += i;
9          }
10
11         std::cout << S << std::endl;
12
13         return 0;
14 }
```

```
~$ 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 larger generalized coding frameworks.

1.2 Object Orientation

The concept 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 [5]. Object orientation quickly became the state-of-the-art in programming, and has 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*, e.g. 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 introductory to class syntax is given.

1.2.1 A Brief Introduction to Essential Concepts

Members

As mentioned, 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

```
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;`.

Ref. [4] is suggested for further introduction to basic concepts of classes.

1.2.2 Inheritance

Consider the abstract idea of a keyboard: A board and keys (obviously). In object orientation terms, the *superclass* of keyboards describe 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.

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

[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()".

keyboard described previously: They are both *subclasses* of the same superclass, inheriting the basic concepts, but overloading the abstract parts with specific implementations.

The following examples will assume a basic knowledge of programming concepts such as constructors and functions. 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. Potential subclasses are e.g. computer - and musical keyboards. An easy way to visualize this inheritance relation is by drawing an *inheritance diagram* as in figure 1.1. A python implementation of these subclasses are given on the next page.

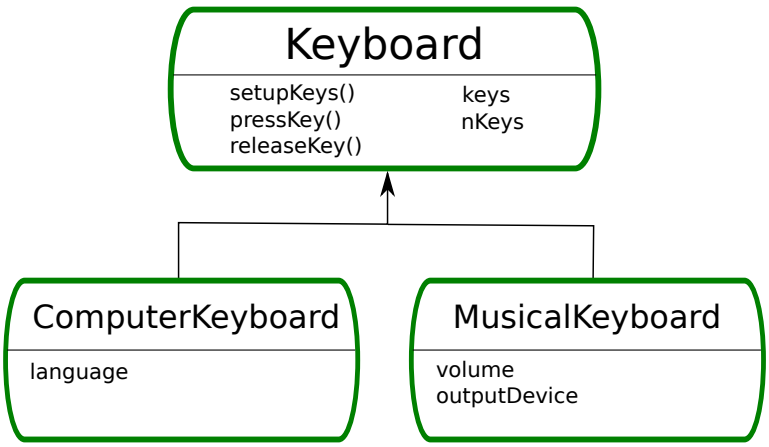

Figure 1.1: Inheritance diagram for a Keyboard superclass. Class members are listed below the class name.

```
#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 harmonic wave with frequency and amplitude
        #representing key pressed and 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 keyboard are at least different in 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*[5]. The other two functions does 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.

[5]We could simply choose not to call the superclass constructor in the given example, however, let's assume for the sake of proving a point that we can't.

1.2.3 Pointers, Typecasting and Virtual Functions

A pointer is a hexadecimal number representing a memory address where some *type* of object is stored, e.g. a `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, e.g.

```
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 1.2.4, 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 does not make sense in a dynamically (automatically) typed language. 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 is set to retrieve a superClass pointer, then calls all the functions.
void testFunc(superClass* someObject){
        someObject->pureVirtual(); someObject->justVirtual(); someObject->notVirtual();
}
```

```
int main(){

        cout << "-Calling subClass object of type superClass*" << endl;
        superClass* object = new subClass(); testFunc(object);

        cout << endl << "-Calling subClass object of type subClass*" << endl;
        subClass* object2 = new subClass(); testFunc(object);

        cout << endl << "-Directly calling object of type subclass*" << endl;
        object2->pureVirtual(); object2->justVirtual(); object2->notVirtual();

        return 0;
}
```

```
~$ ./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.

From the first standard output block it is clear that creating subclass objects of superclass type will indeed overload non-virtual functions, and keep the virtual ones. The second block is as expected identical to the first block, since the function on input typecasts the subclass type into a superclass type, rendering the two examples identical. The third block does not involve typecasting; the subclass functions are called whether they are virtual in the superclass or not.

1.2.4 Polymorphism

The previous example involved a concept referred to as *polymorphism*, a concept tightly 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 code developed in this thesis:

```
class Potential {
protected:
    int n_p;
    int dim;

public:
    Potential(int n_p, int dim);

    //Pure virtual function
    virtual double get_pot_E(const Walker* walker) const = 0;

};

class Coulomb : public Potential {
public:

    Coulomb(GeneralParams &);

    //Returns the sum 1/r_i
    double get_pot_E(const Walker* walker) const;

};

class Harmonic_osc : public Potential {
protected:
    double w;

public:

    Harmonic_osc(GeneralParams &);

    //return the sum 0.5*w*r_i^2
    double get_pot_E(const Walker* walker) const;

};

...
```

Assume an object `Potential* potential` is sent to an energy function. Since `get_pot_E()` is virtual, the potential can take any form. Is it therefore possible to write a code for a general potential while maintaining excellent readability. Moreover, the code can quite easily be adapted to handle any combination of any potential by storing the potential objects in a vector and simply accumulate the contributions:

```
//Simple compiler definition to clean up the code
#define potvec std::vector<Potential*>

class System {
protected:

    double get_potential_energy(const Walker* walker) const;
    potvec potentials;

    ...
};

double System::get_potential_energy(const Walker* walker) const {
    double potE = 0;

    //Iterates through all loaded potentials and accumulate energies.
    for (potvec::iterator pot = potentials.begin(); pot != potentials.end(); ++pot) {
        potE += (*pot)->get_pot_E(walker);
    }

    return potE;
}
```

1.2.5 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 major bugs will arise if changes are made to the object.

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 in the previous example.

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.

1.2.6 Accessibility levels and Friend classes

Upon declaration of a C++ class, 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, for any standardized application (app), the `app::execute_app()` function needs to be public, i.e. accessible from the main file (executing applications from the constructor directly is generally considered bad coding). On the other hand, `app::dump_progress()` should be controlled by the application itself, and should 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 would lead to undesired situations, 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` needs to be made available to the `Distrubution` class. This is implemented in the following fashion:

```

class QMC {
protected:

    arma::mat dist; //!< Matrix holding positional data for the distribution.
    int last_inserted; //!< Index of last inserted positional data.
    ...

public:
    ...

    //Gives Distribution access to protected members of QMC.
    friend class Distribution;

};
```

```
void Distribution::finalize() {

    //scrap out all the over-allocated space (DMC)
    qmc->dist.resize(qmc->last_inserted, dim);

    if (dim == 3) {
        generate_distribution3D(qmc->dist, qmc->n_p);
    } else {
        generate_distribution2D(qmc->dist, qmc->n_p);
    }

    qmc->dist.reset();

}
```

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 intuitive way, forcing an organized implementation of a framework involving several classes. Put in other words: If you have to break an accessibility level to implement a desired functionality, there probably exists a better way to implement it.

1.2.7 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 aviable"

        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)
```

```python
#potionClass.py

class Potion:

    def __init__(self, amount):
        self.amount = amount
        self.setName()

    def applyPotionTo(self, player):
        raise NotImplementedError("Member function applyPotion not implemented.")

    #This function should be overwritten
    def setName(self):
        self.name = "Undefined"


class HealthPotion(Potion):

    #Constructor is inherited

    #Calls back to the player object's functions to change the health
    def applyPotionTo(self, player):
        player.changeHealth(self.amount)

    def setName(self):
        self.name = "Health Potion (%d)" % self.amount


class EnergyPotion(Potion):

    def applyPotionTo(self, player):
        player.changeEnergy(self.amount)

    def setName(self):
        self.name = "Energy Potion (%d)" % self.amount
```

The `Player` class keeps track of everything a player needs of personal data, such as the name, health- and energy levels, potions etc. Bringing another player into the game is now simply done by creating another `Player` object. A player holds a number of `Potion` objects in a list. These objects are subclass implementations which overwrites the virtual function which described the potion's effect on a given player object. 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 e.g. potions and players in the current example 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.

```
#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 36 using 55 energy
Sigve: Insuficcient energy to attack.
Sigve consumes Energy Potion (10).
Sigve hit Karl for 39 using 55 energy

Karl consumes Health Potion (20).
Karl hit Sigve for 42 using 55 energy

Jorgen hit Karl for 30 using 55 energy
Jorgen consumes Energy Potion (20).
Jorgen hit Sigve for 35 using 55 energy

Round 1:
  Sigve (hp/e=23/0):
  No potions aviable

  Jorgen (hp/e=64/10):
  Health Potion (20)

  Karl (hp/e=51/45):
  No potions aviable
```

1.3 Structuring the code

Structuring a code is another source of compromises. If the code is short and has a direct purpose, e.g. to calculate the sum from Eq. (1.1.1), structure is not an issue at all (given that reasonable variable names are provided). 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 Doxygen documentation provided in Ref. [6].

1.3.1 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 the potion game, the gain of transparency is not immense, however, when the class structures span thousands of lines (20000 lines in the case of the QMC library), 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 involves 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 object oriented 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 the 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 a *src* folder, with one folder per distinct class. Subclasses should appear as folders inside the superclass folder. Figure 1.2 shows an example setup for the framework of an object oriented code.

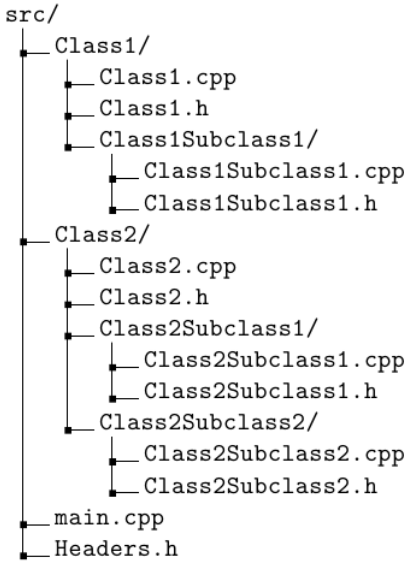

Figure 1.2: An illustration of a standard way to organize source code. The file endings represent C++ code.

Another gain by this type of structuring is that through tools such as Make, QMake, etc., only the actually changed files will be recompiled once changes are made to the code. This will save a lot of time in the development process once the total compilation time starts to span several minutes.

1.3.2 Class Structures

In scientific programming, the simulated process often has a physical or mathematical interpretation. Examples are e.g. 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 we have estimates such as energy, entropy and temperature, all of which are 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_energy` 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 1.2.4, abstracting e.g. 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 used in this thesis, see Section 3.1.

2

Quantum Monte-Carlo

Quantum Monte-Carlo (QMC) is a method of solving Schrödinger's equation using statistical *Markov Chain* (random walk) simulations, i.e. Monte-Carlo 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 ways 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 modeling 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 transparency, this approach will be mentioned only briefly in Section 2.2.3.

In this chapter *Dirac Notation* will be used. See Appendix **??** for an introduction. The equations will be in atomic units, i.e. $\hbar = m_e = e = 1$.

2.1 Modeling Diffusion

Like any phenomena involving a probability distribution, Quantum Mechanics may be modeled 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 making up our system.

The idea is to have an ensemble of *Random Walkers* in which each walker is represented by a position in space and 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 Quantum Monte Carlo is described.

The purpose of this section is to motivate the use of diffusion theory in Quantum Mechanics, and to

[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.

derive the sampling rules needed in order to model Quantum Mechanical distributions by diffusion of random walkers correctly. I will be using natural units, that is $\hbar$, m_e, etc. are all set to unity, in order to simplify the expressions.

2.1.1 Stating the Schrödinger Equation as a Diffusion Problem

Consider the time-dependent Schrödinger equation for an arbitrary many-body wave function[2] $\Phi(\mathbf{r}, t)$ with an arbitrary energy shift E'

$$-\frac{\partial \Phi(\mathbf{r}, t)}{i\partial t} = (\widehat{\mathbf{H}} - E')\Phi(\mathbf{r}, t). \tag{2.1}$$

Given that the Hamiltonian is time-independent, the formal solution is found by separation of variables in $\Phi(\mathbf{r}, t)$ (see ref. [7] for more details)

$$\widehat{\mathbf{H}}\Phi(\mathbf{r}, t_0) = E\Phi(\mathbf{r}, t_0) \tag{2.2}$$

$$\Phi(\mathbf{r}, t) = \exp\left(-i(\widehat{\mathbf{H}} - E')(t - t_0)\right)\Phi(\mathbf{r}, t_0). \tag{2.3}$$

From Eq. (2.3) it is apparent that the time evolution operator is on the form

$$\widehat{\mathbf{U}}(t, t_0) = \exp\left(-i(\widehat{\mathbf{H}} - E')(t - t_0)\right). \tag{2.4}$$

The time-independent equation are solved for the ground state energy through methods such as *Full Configuration Interaction* or similar methods based on diagonalizing the Hamiltonian. The time-dependent equation is used by methods such as *Time-Dependent Multi-Configuration Hartree-Fock* in order to obtain solutions for the time-development of Quantum states. However, none of these originate from or resemble diffusion equations.

The original Schrödinger equation, however, does resemble a diffusion equation in complex time[3]. 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.

Applying the complex time Wick rotation to the time evolution operator in Eq. (2.4) and choosing the energy shift E' equal to the true ground state energy E_0 of $\widehat{\mathbf{H}}$ yields the *projection operation* whose name will soon be apparent

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

Consider an arbitrary state $\Psi_T(\mathbf{r})$. Applying the new operator yields a new state $\Phi(\mathbf{r}, \tau)$ and expanding $\Psi_T(\mathbf{r})$ in the eigenstates $\Psi_i(\mathbf{r})$ of $\widehat{\mathbf{H}}$ yields

[2] See Section 2.6.1 for details regarding many-body wave functions.

[3] 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 REFERER TIL TING :(

$$\begin{aligned}\Phi(\mathbf{r},\tau) &= \langle \mathbf{r}|\,\widehat{\mathbf{P}}(\tau)\,|\Psi_T\rangle \qquad (2.5)\\ &= \langle \mathbf{r}|\exp\left(-(\widehat{\mathbf{H}}-E_0)\tau\right)|\Psi_T\rangle\\ &= \sum_i C_i\,\langle r|\exp\left(-(\widehat{\mathbf{H}}-E_0)\tau\right)|\Psi_i\rangle\\ &= \sum_i C_i\Psi_i(\mathbf{r})\exp\left(-(E_i-E_0)\tau\right)\\ &= C_0\Psi_0(x)+\sum_{i=1}^{\infty} C_i\Psi_k(x)e^{-\delta E_i\tau}, \qquad (2.6)\end{aligned}$$

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.

$$\begin{aligned}\lim_{\tau\to\infty}\Phi(\mathbf{r},\tau) &= \lim_{\tau\to\infty}\langle \mathbf{r}|\,\widehat{\mathbf{P}}(\tau)\,|\Psi_T\rangle\\ &= C_0\Psi_0(x). \qquad (2.7)\end{aligned}$$

The projection operator can in other words be used to transform an arbitrary wave function $\Psi_T(\mathbf{r})$[4] 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

$$\widehat{\mathbf{P}}(\tau) = \prod_{k=1}^{n}\exp\left(-(\widehat{\mathbf{H}}-E_0)\delta\tau\right), \qquad (2.8)$$

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

$$\widehat{\mathbf{P}}(\tau+\delta\tau) = \exp\left(-(\widehat{\mathbf{H}}-E_0)\delta\tau\right)\widehat{\mathbf{P}}(\tau). \qquad (2.9)$$

Using this relation in combination with Eq. (2.5), the effect of the projection operator in a single time-step is revealed

$$\begin{aligned}\Phi(\mathbf{r},\tau+\delta\tau) &= \langle \mathbf{r}|\,\widehat{\mathbf{P}}(\tau+\delta\tau)\,|\Psi_T\rangle\\ &= \langle \mathbf{r}|\exp\left(-(\widehat{\mathbf{H}}-E_0)\delta\tau\right)\widehat{\mathbf{P}}(\tau)\,|\Psi_T\rangle\\ &= \langle \mathbf{r}|\exp\left(-(\widehat{\mathbf{H}}-E_0)\delta\tau\right)|\Phi(\tau)\rangle\\ &= \int_{\mathbf{r}'}\int_{\mathbf{r}''}\langle \mathbf{r}|\mathbf{r}''\rangle\,\langle \mathbf{r}''|\exp\left(-(\widehat{\mathbf{H}}-E_0)\delta\tau\right)|\mathbf{r}'\rangle\,\langle \mathbf{r}'|\Phi(\tau)\rangle\,\mathrm{d}\mathbf{r}'\mathbf{r}''\\ &= \int_{\mathbf{r}'}\langle \mathbf{r}|\exp\left(-(\widehat{\mathbf{H}}-E_0)\delta\tau\right)|\mathbf{r}'\rangle\,\Phi(\mathbf{r}',\tau)\mathrm{d}\mathbf{r}' \qquad (2.10)\end{aligned}$$

[4]This is called the *trial wave function* and will be covered in detail in Section 2.6.

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 (2.6) it is apparent that the projection will still converge as long as $E_T < E_1$, i.e. the energy of the first excitation.

$$\Phi(\mathbf{r}, \tau + \delta\tau) = \int_{\mathbf{r}'} \langle \mathbf{r}| \exp\left(-(\widehat{\mathbf{H}} - E_T)\delta\tau\right) |\mathbf{r}'\rangle \, \Phi(\mathbf{r}', \tau) \mathrm{d}\mathbf{r}' \tag{2.11}$$

$$\equiv \int_{\mathbf{r}'} G(\mathbf{r}, \mathbf{r}'; \delta\tau)\Phi(\mathbf{r}', \tau)\mathrm{d}\mathbf{r}' \tag{2.12}$$

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. (2.12) 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 $\widehat{\mathbf{V}}$ and the energy shift. This is known as the *short time approximation*[8]

$$G(\mathbf{r}, \mathbf{r}'; \delta\tau) = \langle \mathbf{r}| \exp\left(-(\widehat{\mathbf{H}} - E_T)\delta\tau\right) |\mathbf{r}'\rangle \tag{2.13}$$

$$= \langle \mathbf{r}| \, e^{-\widehat{\mathbf{T}}\delta\tau} e^{-(\widehat{\mathbf{V}} - E_T)\delta\tau} \, |\mathbf{r}'\rangle + \frac{1}{2}[\widehat{\mathbf{V}}, \widehat{\mathbf{T}}]\delta\tau^2 + \mathcal{O}(\delta\tau^3). \tag{2.14}$$

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, distributing the correct weights to certain walkers. In other words:

$$G_{\mathrm{Diff}} = e^{\frac{1}{2}\nabla^2\delta\tau} \tag{2.15}$$

$$G_{\mathrm{B}} = e^{-(\widehat{\mathbf{V}} - E_T)\delta\tau}, \tag{2.16}$$

where B denotes *branching*. Modeling weights by branching will be covered in detail in Section 2.5.

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 distributions 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.(2.15) 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 2.8 and 2.9. In either of these methods, diffusion is a key process.

2.2 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 2.2.3.

2.2.1 Isotropic Diffusion

Isotropic diffusion is a process in which diffusing particles sees all possible directions as an equally probable path. Eq. (2.17) is an example of this. This is the simplest form of a diffusion equation, that is, the case with a linear *diffusion constant*, D, and no drift terms.

$$\frac{\partial P(\mathbf{r},t)}{\partial t} = D\nabla^2 P(\mathbf{r},t). \tag{2.17}$$

From Eq. (2.15) 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. A crucial 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 $2D\delta t$ [8]

$$G^{\text{ISO}}_{\text{Diff}}(i \rightarrow j) \propto e^{-|\mathbf{r}_i - \mathbf{r}_j|^2/4D\delta\tau}. \tag{2.18}$$

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 total 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:

$$x_{i+1} = x_i + \xi, \qquad \begin{aligned} \text{Var}(\xi) &= 2D\delta t, \\ \langle \xi \rangle &= x_i, \end{aligned} \tag{2.19}$$

where ξ is a normal distributed number whose variance match that of the Green's function in Eq. (2.18). This relation is in agreement with the isotropy of Eq. (2.17) in the sense that the displacement is symmetric around the current position.

2.2.2 Anisotropic Diffusion: Fokker-Planck

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.

$$\frac{\partial P(\mathbf{r},t)}{\partial t} = D\nabla \cdot \Big[\Big(\nabla - \mathbf{F}(\mathbf{r},t)\Big)P(\mathbf{r},t)\Big]. \tag{2.20}$$

As will be derived in details in Section 2.2.3, 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.

A necessity to the calculation 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. (2.20) is zero:

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

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

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

Looking at 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 become

$$\begin{aligned}\mathbf{F}(\mathbf{r},t) &= \frac{1}{P(\mathbf{r},t)}\nabla P(\mathbf{r},t)\\ &= \frac{2}{|\psi(\mathbf{r},t)|}\nabla|\psi(\mathbf{r},t)|\end{aligned} \tag{2.21}$$

In QMC, the drift term is commonly referred to as the *quantum force*, 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, it's Green's function still has a closed form solution. This means that we can evaluate it efficiently. If this was not the case, the practical value would be reduced dramatically. The reason for this will become clear in Section 2.4. As expected, it is no longer symmetric

$$G^{\mathrm{FP}}_{\mathrm{Diff}}(i\to j) \propto e^{-(x_i - x_j - D\delta\tau F(x_i))^2/4D\delta\tau}. \tag{2.22}$$

The Langevin Equation for the Fokker-Planck Equation

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

$$\frac{\partial x_i}{\partial t} = DF(\mathbf{r})_i + \eta, \tag{2.23}$$

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

$$x_{i+1} = x_i + \xi + DF(\mathbf{r})_i\delta t, \tag{2.24}$$

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 ref. [9], [10] and [11].

2.2.3 The connection between 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

$$\begin{aligned} -\frac{\partial}{\partial t}f(\mathbf{r},t) \;=\; \Psi_T(\mathbf{r})\Big[-\frac{\partial}{\partial t}\Phi(\mathbf{r},t)\Big] &= \Psi_T(\mathbf{r})\left(\widehat{\mathbf{H}}-E_T\right)\Phi(\mathbf{r},t) \\ &= \Psi_T(\mathbf{r})\left(\widehat{\mathbf{H}}-E_T\right)\Psi_T(\mathbf{r})^{-1}f(\mathbf{r},t) \qquad (2.25)\\ &= -\frac{1}{2}\Psi_T(\mathbf{r})\nabla^2\left(\Psi_T(\mathbf{r})^{-1}f(\mathbf{r},t)\right)+\widehat{\mathbf{V}}f(\mathbf{r},t)-E_Tf(\mathbf{r},t). \end{aligned}$$

Expanding the Laplacian term further reveals

$$\begin{aligned} K(\mathbf{r},t) &\equiv -\frac{1}{2}\Psi_T(\mathbf{r})\nabla^2\left(\Psi_T(\mathbf{r})^{-1}f(\mathbf{r},t)\right) \\ &= -\frac{1}{2}\Psi_T(\mathbf{r})\nabla\cdot\left(\nabla\left[\Psi_T(\mathbf{r})^{-1}f(\mathbf{r},t)\right]\right) \qquad (2.26)\\ \nabla\left[\Psi_T(\mathbf{r})^{-1}f(\mathbf{r},t)\right] &= -\Psi_T(\mathbf{r})^{-2}\nabla\Psi_T(\mathbf{r})f(\mathbf{r},t)+\Psi_T(\mathbf{r})^{-1}\nabla f(\mathbf{r},t), \qquad (2.27) \end{aligned}$$

combining these two equations and using the product rule numerous time yields

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

Introducing the following identity helps clean up the messy calculations:

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

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

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

Inserting the expression for the quantum force $\vec{F}(\mathbf{r}) = 2\Psi_T(\mathbf{r})^{-1}\nabla\Psi_T(\mathbf{r})$ 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

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

Inserting everything back into Eq. (2.25) yields

$$\begin{aligned} -\frac{\partial}{\partial t}f(\mathbf{r},t) &= -\frac{1}{2}\nabla\cdot\left[\left(\nabla-\vec{F}(\mathbf{r})\right)f(\mathbf{r},t)\right] + K_L(\mathbf{r})f(\mathbf{r},t) + \widehat{\mathbf{V}}f(\mathbf{r},t) - E_T f(\mathbf{r},t) \\ \frac{\partial}{\partial t}f(\mathbf{r},t) &= \frac{1}{2}\nabla\cdot\left[\left(\nabla-\vec{F}(\mathbf{r})\right)f(\mathbf{r},t)\right] - \left(E_L(\mathbf{r})-E_T\right)f(\mathbf{r},t), \end{aligned} \tag{2.28}$$

which is a Fokker-Planck diffusion equation (Eq. (2.20)) with constant shift representing the branching Green's function in the case Fokker-Planck QMC (see Eq. (2.38).

Just as in traditional importance sampled Monte-Carlo integrals, optimized sampling is obtained by switching distributions into one which exploits known information about the problem at hand. In 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, where as in Quantum Monte-Carlo, 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 Quantum Monte-Carlo*.

The QMC energy estimated using the new distribution $f(\mathbf{r},t)$ will still yield the exact energy in the limit of convergence. Consider the following relation:

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

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

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

2.3 Diffusive Equilibrium Constraints

Upon convergence of a Markov process, the ensemble of walkers on average span the systems most likely state. This is exactly the behavior of a real 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, and it may not; either way its not optimal. Introducing rules of acceptance and rejection on top of the suggested transitions given by the Langevin equation (Eq. (2.19) and Eq. (2.24)) will result in an optimized sampling. Special care must be taken not to break 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.

2.3.1 Detailed Balance

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

$$
P_i W(i \to j) = P_j W(j \to i), \tag{2.29}
$$

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

2.3.2 Ergodicity

Another requirement is that the sampling must be ergodic, that is, the random walkers needs 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.

2.4 The Metropolis Algorithm

The Metropolis Algorithm is a simple set of acceptance/rejection rules used in order to make the thermalization more effective. 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 (see Eq. (2.29) 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

$$\begin{aligned} P_i W(i \to j) &= P_j W(j \to i) \\ P_i g(i \to j) A(i \to j) &= P_j g(j \to i) A(j \to i). \end{aligned} \tag{2.30}$$

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

$$\begin{aligned} |\psi_i|^2 G(i \to j) A(i \to j) &= |\psi_j|^2 G(j \to i) A(j \to i) \\ \frac{A(j \to i)}{A(i \to j)} &= \frac{G(i \to j)}{G(j \to i)} \frac{|\psi_i|^2}{|\psi_j|^2} \equiv R_G(j \to i) R_\psi(j \to i)^2, \end{aligned} \tag{2.31}$$

where the defined ratios correspond to the Green's function - and wave function ratios 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 effective 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. (2.31) for the case where $A(i \to j) = 1$, that is, the case where $P_i < P_j$. This yields

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

Concatenating both scenarios yields the following acceptance/rejection rules:

$$A(i \to j) = \begin{cases} R_G(i \to j) R_\psi(i \to j)^2 & R_G(i \to j) R_\psi(i \to j)^2 < 1 \\ 1 & \text{else} \end{cases} \tag{2.32}$$

Or more simplistic:

$$A(i \to j) = \min\{R_G(i \to j) R_\psi(i \to j)^2, 1\} \tag{2.33}$$

In the isotropic diffusion case, the Greens function ratio cancels due to symmetry, $R_G(i \to j) = 1$, leaving us with the standard Metropolis algorithm:

$$A(i \to j) = \min\{R_\psi(i \to j)^2, 1\} \tag{2.34}$$

On the other hand, for Fokker-Planck diffusion, there will be no cancellation of the Green's function. Inserting Eq. (2.22) into Eq. (2.32) results in the *Metropolis Hastings algorithm*. The ratio of Green's function can be evaluated efficiently by simply subtracting the exponents of the exponentials. This is demonstrated by calculating the logarithm

$$\begin{aligned} \log R_G^{\mathrm{FP}}(i \to j) &= \log\big(G_{\mathrm{Diff}}^{\mathrm{FP}}(j \to i)/G_{\mathrm{Diff}}^{\mathrm{FP}}(i \to j)\big) \\ &= \frac{1}{2}\big(F(x_j) + F(x_i)\big)\big(\frac{1}{2}D\delta t(F(x_j) - F(x_i)) + x_i - x_i\big) \end{aligned} \tag{2.35}$$

$$A(i \to j) = \min\{\exp\big(\log R_G^{\mathrm{FP}}(i \to j)\big)\, R_\psi(i \to j)^2, 1\} \tag{2.36}$$

Derived from detailed balance, the Metroplis Algorithm is as a must-have when it comes to Markov Chain Monte Carlo. Besides Quantum Monte Carlo, methods such as the *Ising Model* solvers greatly benefit from these rules [12].

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 simulations are finite, and the trial positions of walkers are random. A chart flow describing the implementation of the Metropolis algorithm and the diffusion process in general is given in Fig. 2.1.

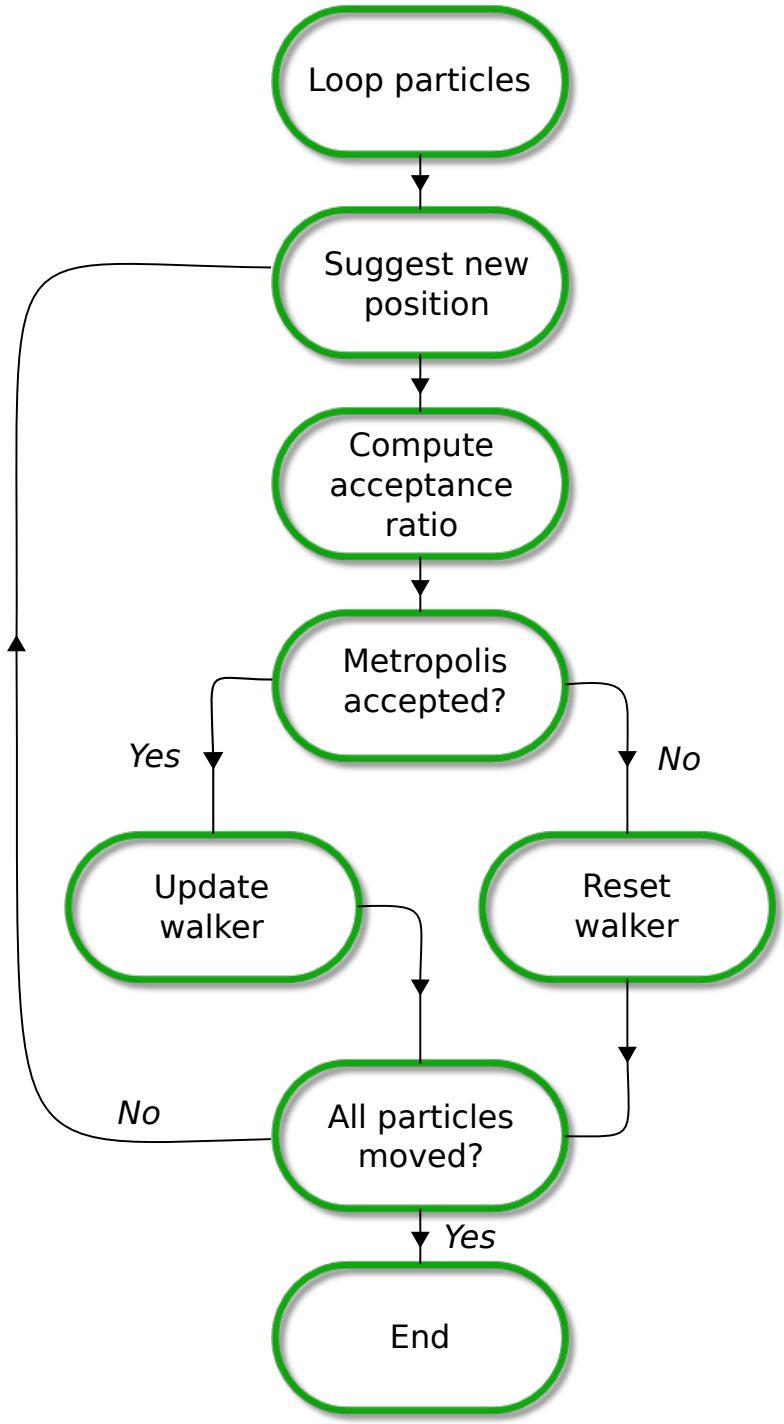

Figure 2.1: Flow chart for iterating a walker through a single time step, i.e. simulation the application of the Green's function from Eq. (2.15) using the Metropolis algorithm. New positions are suggested according to the chosen diffusion model.

2.5 The Process of Branching

In the previous section it became clear that the Metropolis test will guide the walkers to span the trial wave function. This implies that without further action, no changes to the distribution can be made, and the point of modeling the projection operator from Eq. (2.5) is rendered useless. The important fact to include is that the branching Green's function from Eq. (2.38) and (2.37) distribute weights to the walkers, effectively altering the spanned distribution.

The process of branching in Quantum Monte-Carlo is simulated by the creation and destruction of walkers with probability equal to that of the branching Green's function [8]. The explicit shapes in case of isotropic diffusion (ISO) and Anisotropic (FP) is

$$G_B^{\mathrm{ISO}}(i \to j) = e^{-\left(\frac{1}{2}[V(x_i)+V(x_j)]-E_T\right)\delta\tau} \tag{2.37}$$

$$G_B^{\mathrm{FP}}(i \to j) = e^{-\left(\frac{1}{2}[E_L(x_i)+E_L(x_j)]-E_T\right)\delta\tau}, \tag{2.38}$$

where $E_L(x_i)$ is the energy evaluated in configuration x_i (see Section 2.6.4 for details). The three different scenarios which arise is

- $G_B = 1$: No branching, proceed main loop.
- $G_B = 0$: The current walker is to be removed from the current ensemble.
- $G_B > 1$: Make on average $G_B - 1$ replicas of the current walker.

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

$$\overline{G}_B = \mathrm{floor}\left(G_B + a\right), \tag{2.39}$$

where a is a uniformly distributed number on $[0, 1)$. The chance 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.

The process of branching is demonstrated in Fig. 2.2.

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 actual code reference at ref. [6]. Isotropic diffusion is in practice never used with branching due to the singularities in the Coulomb interaction (see Eq. (2.37)). This singularity may lead to large fluctuations in the walker population, the exact opposite to the optimal behavior.

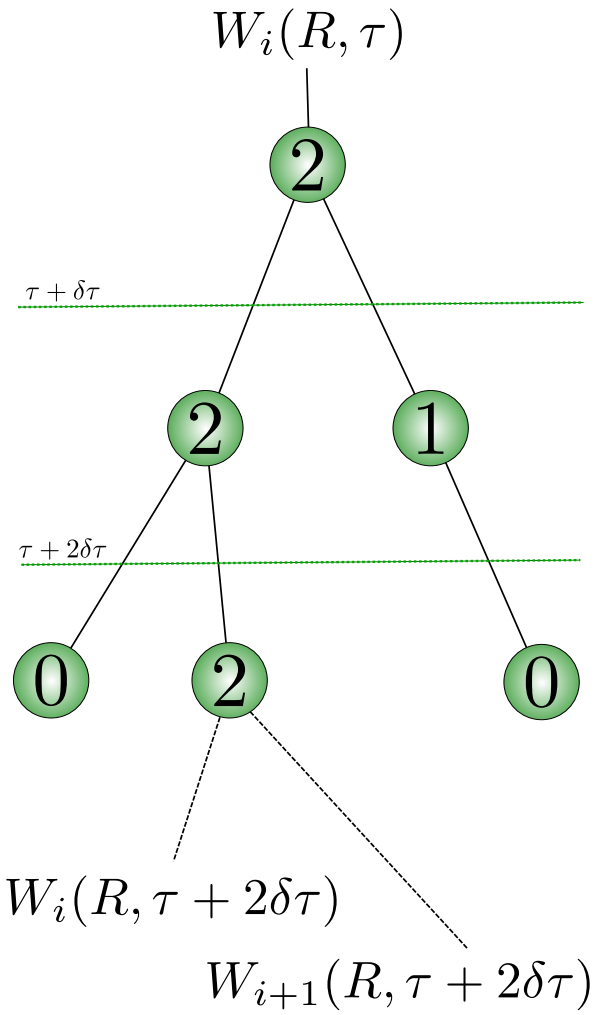

Figure 2.2: Branching illustrated. The initial walker $W_i(R,\tau)$ is branched according to the rules of Section 2.5. The numerical value inside the nodes represents $\overline{G}_B$ from Eq. (2.39). Each horizontal dashed line represent a diffusion step, i.e. a transition in time. Two lines exiting the same node represent identical walkers. After moving through the diffusion process, not two walkers should ever be equal (given that not all of the steps was rejected).

2.6 The Trial Wave Function

Recall Eq. (2.5) and (2.6). The initial condition, $\Phi(\mathbf{r}, t_0) \equiv \Psi_T(\mathbf{r})$, is in Quantum Monte-Carlo referred to as the trial wave function. Mathematically, we may choose any normalizable wave function, whose overlap with the exact ground state wave function, $\Psi_0(x)$, is non-zero. If the overlap is zero, that is, $C_0 = 0$ in Eq. (2.7), 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 $\Psi_0(x)$ will become the dominant contribution to our distribution.

In other words: The trial wave function should be chosen in such a manner 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 educational guesses and by exploiting known requirements put on the exact ground state. 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 basic Quantum Mechanics, I suggest reading in Ref. [7]. For mathematically rigid derivations of concepts, see ref. [13]. More details regarding many-body theory can be found in Ref. [14].

2.6.1 Many-body Wave Functions

Many-body theory arise from the fact that we have *many-body interactions*, e.g. the Coulomb interaction between two particles. Nature operates with 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, i.e. 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 (Eq. (2.2)) for the lowest energy eigenvalue

$$\widehat{\mathbf{H}}\Psi_0(\mathbf{r}) = E_0\Psi_0(\mathbf{r}), \tag{2.40}$$

where $\mathbf{r} \equiv \{\mathbf{r}_1, \mathbf{r}_2, ..., \mathbf{r}_N\}$ represents every particle's position. Exact solutions to realistic many-body systems rarely exist, however, like in Section 2.1.1, expanding the solution in a known basis $\Phi_k(\mathbf{r})$ is always legal, which reduces the problem into that of a *coefficient hunt*

$$\Psi_0(\mathbf{r}) = \sum_{k=0}^{\infty} C'_k\Phi_k(\mathbf{r}), \tag{2.41}$$

where the primed coefficients should not to be confused with the previous coefficients expanding an arbitrary state in the Ψ_i basis (see Eq. (2.6)). Different many-body methods, e.g. *Hartree Fock*[5] and genetic algorithms, give rise to different ways of estimating these coefficients, however, certain concepts are necessarily common, for instance truncating the basis at some level, K:

$$\Psi_0(\mathbf{r}) = \sum_{k=0}^{K} \tilde{C}'_k\Phi_k(\mathbf{r}), \tag{2.42}$$

[5]Hartree-Fock is roughly a basis change from the non-interacting case into a basis which is orthogonal to one-particle excitations. The exact ground state wave function should be orthogonal to all excited states, so it's a fair approximation depending on the dominance of one-particle excitations in the given system.

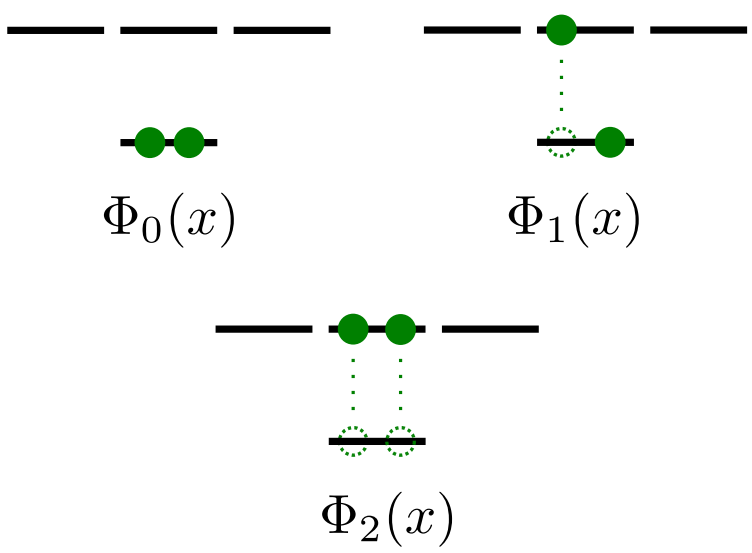

Figure 2.3: Three different electron configurations in an shell structure making up three different $\Phi_k(\mathbf{r})$, i.e. constituents of the many-body basis described in Eq. (2.42). An electron (solid dot) is represented by e.g. the orbital $\phi_{1s}(\mathbf{r}_1)$.

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 (given a reasonable ordering, more on this shortly).

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) $\phi_n(\mathbf{r}_i)$, combined in different ways. The process of calculating basis elements often boils down to a combinatoric exercise involving combinations of orbitals.

Imagine electrons surrounding a nucleus, i.e an atom. A single electron occupying a state with quantum numbers n at a position $\mathbf{r}_i$ is then described by the orbital $\phi_n(\mathbf{r}_i)$. Each unique[6] 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 Fig. 2.3 for a demonstration of this. The ordering of the terms in Eq. (2.42) 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:

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. (2.42), but is seldom necessary to perform explicitly; expressions

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

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 a N-particle system is

$$\widehat{\mathbf{H}} = \widehat{\mathbf{H}}_0 + \widehat{\mathbf{H}}_{\mathrm{I}}, \tag{2.43}$$

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

$$\begin{aligned} \widehat{\mathbf{H}}_0 &= \sum_{i=1}^{N} \widehat{\mathbf{h}}_0(\mathbf{r}_i) && (2.44)\\ &= \sum_{i=1}^{N} \widehat{\mathbf{t}}(\mathbf{r}_i) + \widehat{\mathbf{u}}_{\mathrm{ext}}(\mathbf{r}_i) \\ \widehat{\mathbf{H}}_{\mathrm{I}} &\simeq \sum_{i<j=1}^{N} \widehat{\mathbf{v}}(r_{ij}) && (2.45)\\ &= \sum_{i<j=1}^{N} \frac{1}{r_{ij}}, \end{aligned}$$

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

In order to optimize the overlap C_0, the single particle orbitals are commonly chosen to be the solutions of the non-interacting case (given that they exist)

$$\widehat{\mathbf{h}}_0(\mathbf{r}_i)\phi_n(\mathbf{r}_i) = \epsilon_n\phi_n(\mathbf{r}_i). \tag{2.46}$$

If no such choice can be made, choosing e.g. free-particle solutions, Laguerre polynomials etc. is the general strategy. However, in this case, the expansion truncation K needs to be higher in order to achieve a satisfying overlap.

Step two in detail

In the case of *Fermions*, i.e. half-integer spin particles like electrons, protons, etc., $\Phi_k(\mathbf{r})$ is an anti-symmetric function[7] on the form of a determinant: The *Slater determinant*. The shape of the determinant is given in Eq. (2.47). 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 (see Eq. (2.48)), which in many ways are easier to deal with because of the lack of an exclusion principle. In order to keep the terminology less abstract and confusing, from here on, the focus will be on systems of fermions.

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

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

$$= \begin{vmatrix} \phi_1(\mathbf{r}_1) & \phi_2(\mathbf{r}_1) & \cdots & \phi_N(\mathbf{r}_1) \\ \phi_1(\mathbf{r}_2) & \phi_2(\mathbf{r}_2) & \cdots & \phi_N(\mathbf{r}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\mathbf{r}_N) & \phi_2(\mathbf{r}_N) & \cdots & \phi_N(\mathbf{r}_N) \end{vmatrix} \tag{2.47}$$

$$\Phi_0^{\mathrm{S}}(\mathbf{r}_1, \mathbf{r}_2, ..., \mathbf{r}_N) \propto \sum_{\mathrm{P}} \widehat{\mathbf{P}} \phi_1(\mathbf{r}_1)\phi_2(\mathbf{r}_2)...\phi_N(\mathbf{r}_N) \tag{2.48}$$

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. (2.41) for $k > 0$ are referred to as *correlation* terms. Given that the orbital wave functions are chosen by Eq. (2.46), the existence of the correlation terms, i.e. $C'_k \neq 0$ for $k > 0$, follows as a direct consequence of the many-body interaction, H_{I}, 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 we perform the calculation using the exact wave function, the diverging terms will cancel out; the energy is independent of the position.

In other words, incorporating the correct correlated wave function will result in a cancellation in the diverging term as we approach singularities in the many-body interactions, a property of exact ground states which can be enforced upon the approximation to further optimize the overlap.

These criteria are called *Cusp Conditions*, and serve as a powerful guide when it comes to selecting a trial wave function.

2.6.2 Choice of 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 eigenfunctions of the non-interacting case (given that they exist). Starting from Eq. (2.42), 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

$$\Psi_T(\mathbf{r}_1, ..., \mathbf{r}_N) = \Big[\sum_{k=0}^{K} C_k \Phi_k(\mathbf{r}_1, ..., \mathbf{r}_N) \Big] \prod_{i<j}^{N} f(r_{ij}). \tag{2.49}$$

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. [8] demonstrates this nicely: Hylleraas presented the following correlation function

$$f(r_{ij})_{\text{Hylleraas}} = e^{-\frac{1}{2}(r_i+r_j)} \sum_k d_k (r_{ij})^{a_k} (r_i + r_j)^{b_k} (r_i - r_j)^{e_k}, \tag{2.50}$$

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 almost 1078 terms. For the purpose of Quantum Monte-Carlo this is beyond overkill: Three decimal precision is obtained using a single free parameter (see the results for Atoms in Table **??**).

A commonly used correlation function in studies involving few variational parameters is the *Padé Jastrow* function (skipping some redundant indices)

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

For systems where the correlations are relatively well behaving, it is custom to drop the second term, and keep only the $k = 1$ term, which yields

$$f(r_{ij}; \beta) = \exp\left(\frac{a_{ij} r_{ij}}{1 + \beta r_{ij}} \right), \tag{2.51}$$

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[8]. For two dimensions, the values are $a_{ij} = 1/3$ (parallell) or $a_{ij} = 1$ (anti-parallel)[15]. This is the correlation function used for all systems this thesis.

Shifting the focus back to the spatial wave function, in the case of a fermionic system, the evaluation of a $N \times N$ Slater determinant severely limits the efficiency of many-particle simulations. However, assuming we have a spin-independent Hamiltonian, we can (in QMC) split the spatial wave function in two: One part for each spin eigenvalue. A detailed derivation of this is given in the appendix of Ref. [16]. Assuming spin-half particles we get

$$\Psi_T(\mathbf{r}; \beta) = \Big[\sum_{k=0}^{K} C'_k \tilde{\Phi}_k(\mathbf{r}_1, ..., \mathbf{r}_{\frac{N}{2}}) \tilde{\Phi}_k(\mathbf{r}_{\frac{N}{2}-1}, ..., \mathbf{r}_N) \Big] \prod_{i<j}^{N} f(r_{ij}; \beta). \tag{2.52}$$

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

$D^{\uparrow}$, and the spin down one $D^{\downarrow}$. Stitching everything together yields the following explicit shape for a spin-independent Hamiltonian using a one-parameter Padé Jastrow function

$$\Psi_T(\mathbf{r}_1, ..., \mathbf{r}_N; \beta) = \sum_{k=0}^{K} C'_k D_k^{\uparrow} D_k^{\downarrow} \prod_{i<j}^{N} f(r_{ij}; \beta). \tag{2.53}$$

This shape is referred to as a *Multi-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. (2.6).

Because of this fact, limitations due to the trial wave function in full[8] 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 means that we are in a position to trade CPU-time per walker for convergence time. For systems of many particles, CPU-time per walker needs to be as low as possible in order to get the computation done in a reasonable amount of time, i.e., 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, increasing the number of particles in the system may cause the generic trial wave function to be relatively worse than in the case with fewer particles. This is demonstrated in Ref. [17] where calculations for F_2 (18 particles) needs 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 function

In the case of well-behaving systems, a single determinant (with the Jastrow factor) is a reasonable approximation. This simplicity opens up the possibility of simulating large systems efficiently (more details regarding this is given in Section 3.3).

In order to further optimize the overlap with the exact wave function, a variational parameter α is introduced in the spatial part (in addition to $\beta$from Eq. (2.51))

$$\Psi_T(\mathbf{r}_1, ..., \mathbf{r}_N; \alpha, \beta) = D^{\uparrow}(\alpha) D^{\downarrow}(\alpha) \prod_{i<j}^{N} f(r_{ij}; \beta). \tag{2.54}$$

Determining optimal values for the variational parameters will be discussed in Section 2.6.3. If the introduction of the variational parameter was redundant, optimizations would simply yield $\alpha = 1$.

[8] "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.

2.6.3 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. (2.6)

$$\begin{aligned} E_0 &= \langle\Psi_0|\,\widehat{\mathbf{H}}\,|\Psi_0\rangle \\ E &= \langle\Psi_T(\alpha,\beta)|\,\widehat{\mathbf{H}}\,|\Psi_T(\alpha,\beta)\rangle \\ &= \sum_{kl} C_k^* C_l \underbrace{\langle\Psi_k|\,\widehat{\mathbf{H}}\,|\Psi_l\rangle}_{E_k\delta_{kl}} \\ &= \sum_k |C_k|^2 E_k \end{aligned}$$

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 result 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 (assuming no maxima exist for finite values of the parameters)

$$\frac{\partial\langle E\rangle}{\partial\alpha_i} = \frac{\partial}{\partial\alpha_i}\langle\Psi_T(\alpha_i)|\,\widehat{\mathbf{H}}\,|\Psi_T(\alpha_i)\rangle = 0 \tag{2.55}$$

In order to work with Eq. (2.55) in practice, it needs to be rewritten it in terms of known values. Since our wave function is dependent on the variational parameter, the normalization factor needs to be included in the expression of the expectation value

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

where the product and division rules for derivatives has been applied. The Hamiltonian does not depend on the variational parameters, hence both terms in the first expansion is equal. Cleaning up the expression yields

$$\begin{aligned}\frac{\partial\langle E\rangle}{\partial\alpha_i} &= 2\left(\frac{\langle\Psi_T(\alpha_i)|\,\widehat{\mathbf{H}}\frac{\partial}{\partial\alpha_i}\,|\Psi_T(\alpha_i)\rangle}{\langle\Psi_T(\alpha_i)|\Psi_T(\alpha_i)\rangle} - \langle E\rangle\frac{\langle\Psi_T(\alpha_i)|\,\frac{\partial}{\partial\alpha_i}\,|\Psi_T(\alpha_i)\rangle}{\langle\Psi_T(\alpha_i)|\Psi_T(\alpha_i)\rangle}\right)\\ &= 2\left(\left\langle E\frac{\partial\Psi_T}{\partial\alpha_i}\right\rangle - \langle E\rangle\left\langle\frac{\partial\Psi_T}{\partial\alpha_i}\right\rangle\right)\end{aligned} \tag{2.56}$$

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

$$\frac{\partial\Psi_T(\mathbf{r};\alpha_i)}{\partial\alpha_i} = \sum_{p=1}^{N}\sum_{q=0}^{N/2}\phi_q(\mathbf{r}_p;\alpha_i)\left[\frac{\partial\phi_q(\mathbf{r}_p;\alpha_i)}{\partial\alpha_i}\right]D^{-1}_{qp}, \tag{2.57}$$

where D^{-1}_{qi} is the inverse of the Slater matrix, which will be introduced in more detail in Section 3.4.1.

Using these expressiosn for the *variational energy gradient*, the derivatives can be calculated exactly the same way as for e.g. the energy. The gradient can then be used to move in the direction of the variational minimum in Eq. (2.55).

This strategy give rise to numerous ways of finding the optimal parameters, such as using the well known Newton's method, conjugate gradient methods [18], 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 2.7.

2.6.4 Calculating Expectation Values

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

$$\begin{aligned}\langle\Psi_T|\,\widehat{\mathbf{O}}\,|\Psi_T\rangle &= \int\Psi_T(x)^*\widehat{\mathbf{O}}\Psi_T(x)\mathrm{d}x\\ &= \int|\Psi_T|^2\left(\frac{1}{\Psi_T(x)}\widehat{\mathbf{O}}\Psi_T(x)\right)\mathrm{d}x\\ &= \int|\Psi_T|^2O_L(x)\mathrm{d}x \qquad (2.58)\\ O_L(x) &= \frac{1}{\Psi_T(x)}\widehat{\mathbf{O}}\Psi_T(x) \qquad (2.59)\end{aligned}$$

Discretizing the integral yields

$$\langle\Psi_T|\,\widehat{\mathbf{O}}\,|\Psi_T\rangle \equiv \langle O\rangle \simeq \frac{1}{n}\sum_{i=1}^{n}O_L(x_i) \equiv \overline{O}, \tag{2.60}$$

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 $n\to\infty$, i.e.

$$\langle O\rangle = \lim_{n\to\infty}\overline{O} = \lim_{n\to\infty}\frac{1}{n}\sum_{i=1}^{n}O_L(x_i) \tag{2.61}$$

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

$$\langle E \rangle \simeq \frac{1}{n} \sum_{i=1}^{n} \left(\frac{1}{\Psi_T(x_i)} \left(-\frac{1}{2} \nabla^2 \right) \Psi_T(x_i) + V(x_i) \right) \tag{2.62}$$

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

2.6.5 Normalization

Every explicit calculation using the trial wave function in Quantum Monte-Carlo involves taking ratios. Calculating ratios implies a cancellation in the normalization factors. Eq. (2.32) from the Metropolis section, the Quantum Force in the Fokker-Planck equation, and the sampling of local values describes in the previous section demonstrates exactly this; everything involves ratios.

Not having to normalize our wave functions saves us a lot of CPU-time, but it also relieves us of complicated normalization factors in our single particle basis expressions; any constants multiplying $\phi_n(x_i)$ in Eq. (2.47) and Eq. (2.48), can be taken outside the sum over permutations, and will hence 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. In the case of multi-determinants trial wave functions, the normalization factors from the single particle basis elements will be absorbed by the respective determinant's coefficient C_k, and is hence obsolete in this case as well.

2.7 Gradient Descent Methods

The direction of a gradient serves as a guide to extremal values. Gradient descent, also called steepest descent[9], 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.

2.7.1 General Gradient Descent

Seeking maxima or minima is simply a question of whether the positive or 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

$$\nabla f(x_m) = 0 \tag{2.63}$$
$$\nabla f(x_m - \mathrm{d}x) < 0 \tag{2.64}$$
$$\nabla f(x_m + \mathrm{d}x) > 0 \tag{2.65}$$

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 (backwards) a number of steps. From Fig. 2.4 and the previous equations, it is clear that

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

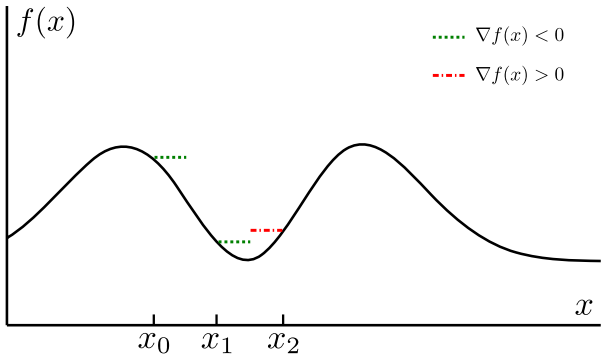

Figure 2.4: Two steps of a one dimensional Gradient Descent process. Steps are taken in the direction of the negative gradient (indicated by dotted lines).

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 a extreme amount of very small steps in order to achieve good precision.

The difference equation describing the steps from the previous paragraph would be

$$x_{i+1} = x_i - \delta \frac{\nabla f(x_i)}{|\nabla f(x_i)|} \tag{2.66}$$

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 Fig. 2.4. To counter this, a changing step length δ_i is introduced

$$x_{i+1} = x_i - \delta_i \nabla f(x_i) \tag{2.67}$$

All gradient/steepest descent methods are in principle described by Eq. (2.67)[10]. 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.

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

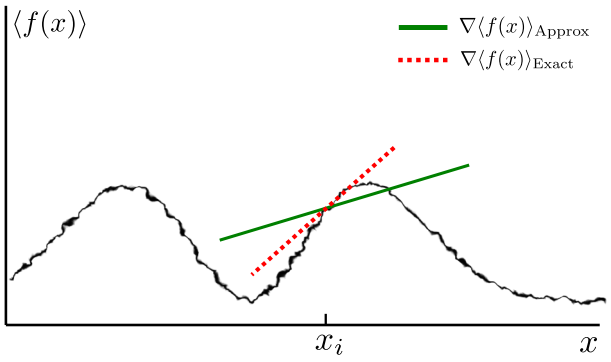

Figure 2.5: A one dimensional plot of an expectation valued function. Smeared lines are representing uncertainties due to rough sampling. The direction of the local gradient (solid green line) at a point x_i is not necessarily a good estimate of the actual analytic gradient (dashed red line).

2.7.2 Stochastic Gradient Descent

Minimizing stochastic quantities such as the variance and expectation values adds another layer of complications on top of the methods described 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. (2.56) is an example of such a process.

A full precise sampling of the stochastic quantities are 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 are dangerous, as uncertainties in the values will cause the gradient to become unstable when the variations are (expectedly) low close to the minimum. This is illustrated in Fig. 2.5.

2.7.3 Adaptive Stochastic Gradient Descent

Adaptive Stochastic Gradient Descent (ASGD) has it's roots in the mathematics of automated control theory [19]. 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

$$X_i \equiv -\nabla_i \cdot \nabla_{i-1} \tag{2.68}$$

The step length from Eq. (2.67) is modeled in the following manner in ASGD

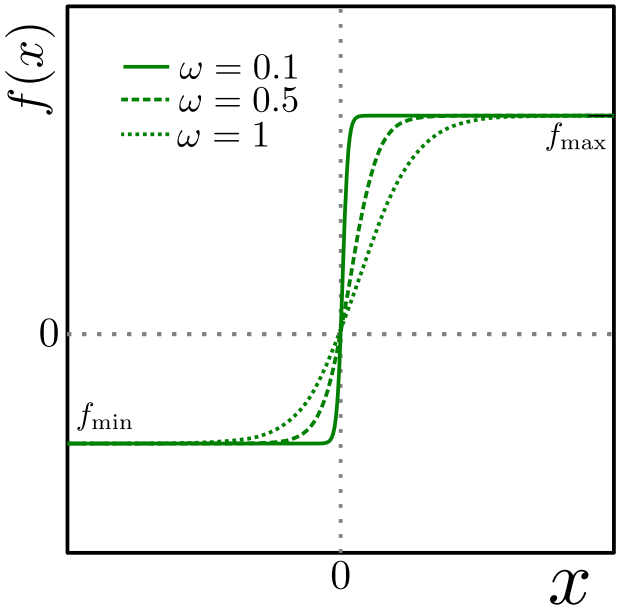

Figure 2.6: Examples of $f(X_i)$ as published in ref. [20]. As $\omega \to 0$, $f(x)$ approaches a step function.

$$\delta_i = \gamma(t_i) \tag{2.69}$$
$$\gamma(t) = a/(t+A) \tag{2.70}$$
$$t_{i+1} = \max(t_i + f(X_i), 0) \tag{2.71}$$
$$f(x) = f_{\min} + \frac{f_{\max} - f_{\min}}{1 - (f_{\max}/f_{\min})e^{-x/\omega}} \tag{2.72}$$

with $f_{\max} > 0$, $f_{\min} < 0$, and $\omega > 0$. Free parameters are a, A and t_0, however, Ref. [20] 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. If we are 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. (2.68): The value of X_i will be positive.
- Eq. (2.72): $f(X_i)$ will return a value in $[0, f_{\max}]$ depending on the magnitude of X_i.
- Eq. (2.71): The value of t will increase, i.e. $t_{i+1} > t_i$.
- Eq. (2.70): The step length will decrease.

The second step regarding $f(X_i)$ can be visualized in Fig. 2.6.

Assumptions

These assumptions are selected direct citations from Ref. [20]. 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 estimated averages involving very few samples.

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

This is shown in ref. to be true; 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 (2.70), 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 Fig. 2.6.

Implementation

A flow chart of the implementation is given in Fig. 2.8. For specific details regarding the implementation, see the code [6]. An example of minimization using ASGD is given in Fig. 2.7.

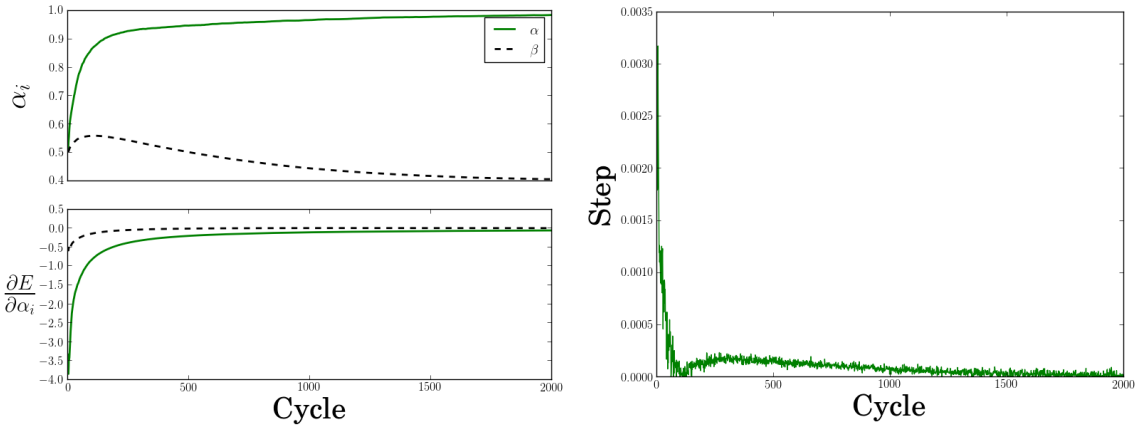

Figure 2.7: Results of Adaptive Stochastic Gradient Descent used on a two-particle quantum dot with unit oscillator frequency using 400 cycles pr. gradient sampling and 40 independent walkers. The right figure shows the evolution of the time step. The left figure shows the evolution of the variational parameters α and β introduced in Section 2.6 on top, and the evolution of the gradients on the bottom. The gradients are cycle averaged to reveal the pattern underlying the noise. We clearly see that they tend to zero, β somewhat before α. The step rushes to zero; we get a small rebound in the step after forcing it zero as it attempts to cross to negative values.

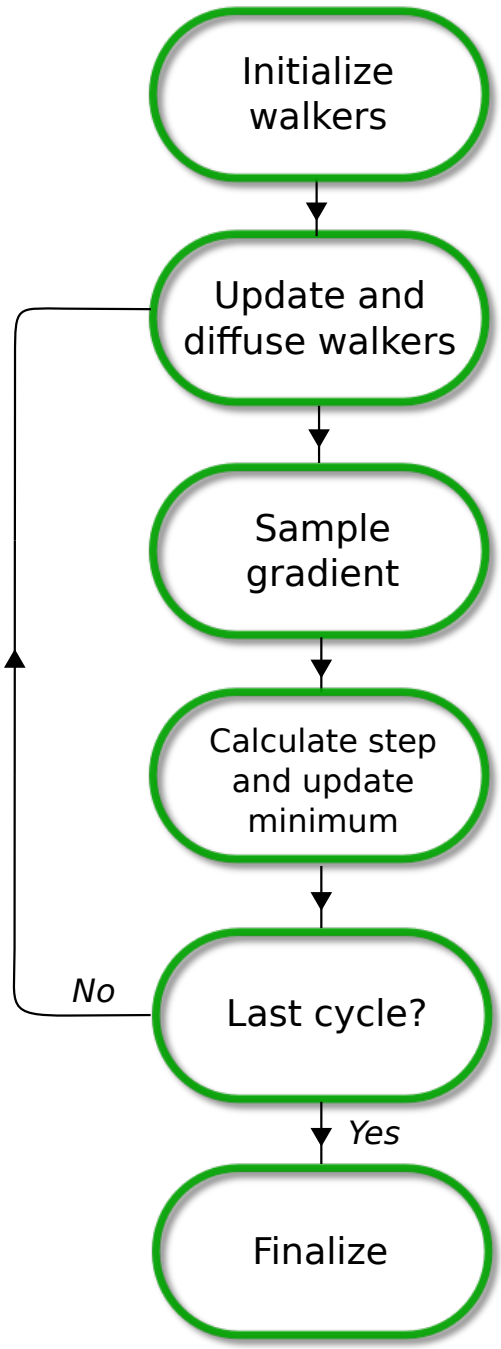

Figure 2.8: Chart flow of ASGD algorithm. Diffusing a walker is done as described in Fig. 2.1. Updating the walkers involves recalculating any values afflicted by updating the minimum. The step is calculated by Eq. (2.70). In case of Quantum Monte-Carlo, the gradient is sampled by Eq. (2.56).

2.8 Variational Monte-Carlo

As briefly mentioned in the derivation of the Quantum Monte-Carlo projection process, neglecting the branching term, i.e. set $G_{\mathrm{B}} = 1$, leaves us with a method called Variational Monte-Carlo (VMC). The naming is due to the fact that the method is variational; it supplies as upper bound for the exact ground state energy (see Section 2.6.3): The better the trial wave function, the closer the answer is to the exact ground state energy.

Without the branching term, the optimal converged state of the Markov Chain is to span that of the trial wave function. From the flow chart of the VMC algorithm in Fig. 2.9, it is clear that VMC corresponds to nothing but a standard Monte-Carlo integration of the local energy, distributing the points according to the trial wave function (ensured by the Metropolis algorithm and the Langevin equation).

2.8.1 Motivating the use of Diffusion Theory

The question becomes: Why bother with all the diffusion theory 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, is *wave function dependent* $\Psi_T(\mathbf{r})$. Eq. (2.73 implies that the evaluation of the local energy in an arbitrary distribution $P(\mathbf{r})$ is *undefined* at the zeros of $\Psi_T(\mathbf{r})$, a "0/0" expression:

$$\begin{aligned} E_{\mathrm{VMC}} &= \int P(\mathbf{r}) \frac{1}{\Psi_T(\mathbf{r})} \widehat{\mathbf{H}} \Psi_T(\mathbf{r}) \mathrm{d}\mathbf{r} \\ &\simeq \frac{1}{n} \sum_{i=1}^{n} \frac{1}{\Psi_T(\mathbf{r}_i)} \widehat{\mathbf{H}} \Psi_T(\mathbf{r}_i), \end{aligned} \tag{2.73}$$

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

The important fact is that the integral *needs* to importance sampled with the distribution $P(\mathbf{r}) = |\Psi_T(\mathbf{r})|^2$ in order to solve the integral in a satisfying manner, that is, avoid sampling close to the roots of $\Psi_T(\mathbf{r})$ without "cheating". Introducing importance sampling is done by simply switching distributions (since the calculated value of interest is an expectation value, there is no need to scale the sampled function as is done for standard arbitrary integrals).

Suggesting new positions (diffusion) boils down to be analogous to calling a *random number generator* corresponding to the trial wave function squared distribution. The problem was the roots of $\Psi_T(\mathbf{r})$, however, the distribution of points now share these roots, i.e. the probability of sampling a point where the local energy is undefined equals zero.

$$\Psi_T(x_m) = 0 \quad \Longrightarrow \quad P(x_m) = 0 \tag{2.74}$$

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.

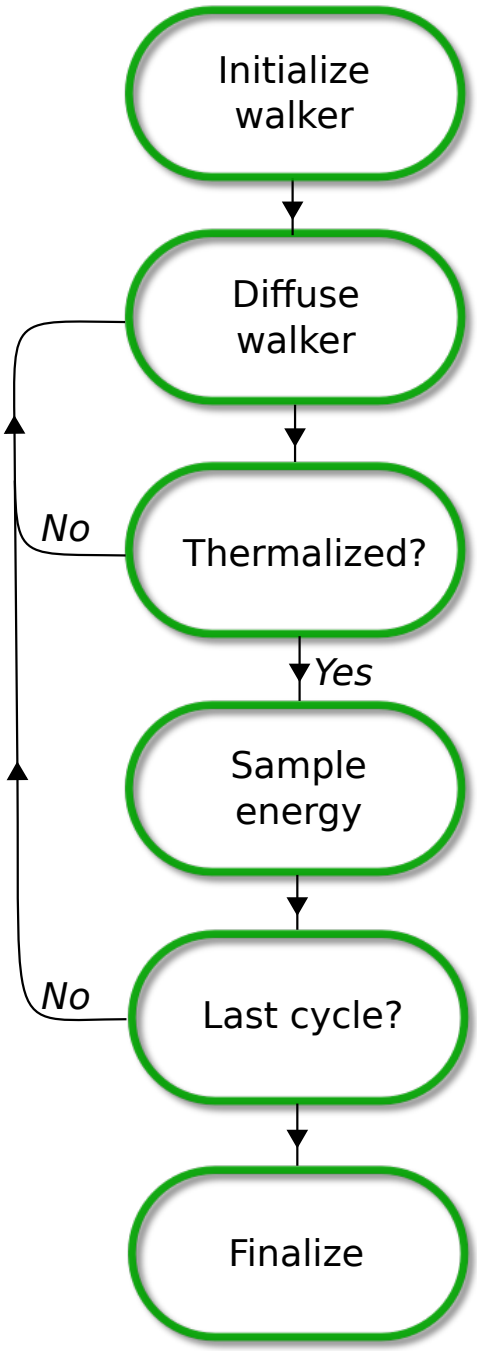

Figure 2.9: Chart flow of the Variational Monte-Carlo algorithm. The second step, *Diffuse Walker*, is the process described in Fig. 2.1. Energies are sampled as described in Section 2.6.4. Thermalization is usually set to a fixed number of cycles.

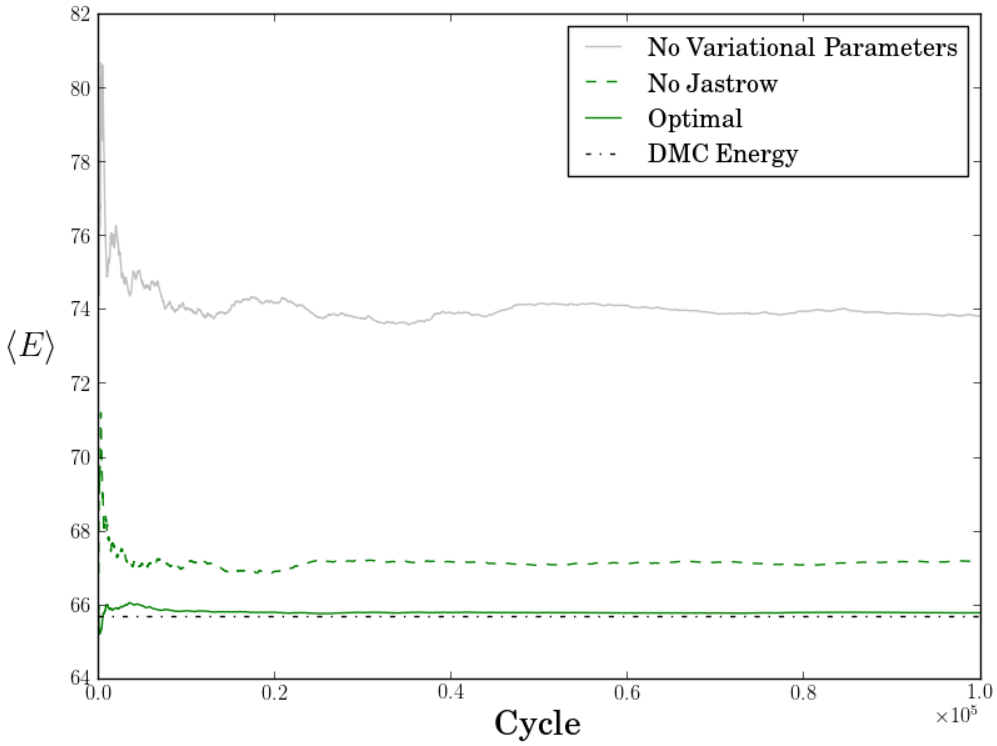

Figure 2.10: Comparison of the VMC energy using different trial wave functions. The DMC energy is believed to be very close to the exact ground state. It is apparent that adding a variational parameter to the trial wave function lowers the energy substantially, however, when adding the Jastrow factor (denoted *Optimal*) described in Section 2.6.2 the VMC energy gets very close to the "exact" answer. Lower energy means a better energy when dealing with variational methods. In this example, a 12 particle quantum dot with unit frequency is used.

2.8.2 Implementation

Variational Monte-Carlo does not benefit much from an increase of samples (beyond a given point, that is). It is much more important that the system is thermalized, than that the number of walkers are high, or the final amount of samples are huge. It is therefore sufficient to use a single walker pr. VMC process.

The single walker is initialized in a normal distributed manner (with variance $2D\delta\tau$), and released to diffuse according to the process in Fig. 2.1. A flow chart of the VMC algorithm is given in Fig. 2.9. Finalizing the sampling involves scaling energies, calculating the variance, etc.

For more details regarding the specific implementation, see the code in ref. [6].

2.8.3 Limitations

The only limitation in VMC is the choice of trial wave function. This makes VMC extremely robust; it will *always* produce a result. As the overlap C_0 in Eq. (2.6) approach unity, the VMC energy approaches the exact ground state energy as a monotone decreasing function. Fig. 2.10 described this effect.

2.9 Diffusion Monte-Carlo

Applying the full diffusion framework introduced in the previous sections results in a method known as Diffusion Monte-Carlo (DMC)[11]. Diffusion Monte-Carlo results are often referred to as *exact*, in the sense that it is overall one of the most precise many-body methods. However, just as any many-body method, DMC also has its limitations. These will be discussed in Section 2.9.3.

Where other many-body methods run into the *curse of dimensionality* (CPU-time exponentially increasing with number of particles, precision, etc.), DMC with it's 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.

Details regarding the exactness of DMC will be covered in the section on limitations.

2.9.1 Implementation

DMC is as mentioned a precise method; more statistics is like water in a desert - there is never enough of it. In addition, branching is a major part of the algorithm. In other words: DMC 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 distributed according to the trial wave function at $\tau = 0$.

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. This loop continues until the current walker is either dead ($G_B = 0$), or 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 Fig. 2.11.

2.9.2 Sampling the Energy

Unlike VMC, DMC does not weigh all walkers equally. It is therefore necessary to weigh each walker's contribution to a cumulative energy sampling accordingly, that is, with 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 our system at time-step k, $\tilde{n}_{b,i}$ be the number of blocks walker i survives, and let $W_i(\vec{r}, \tau)$ represent walker i. The relation is then

$$E_k = \frac{1}{n_w} \sum_{i=1}^{n_w} \frac{1}{\tilde{n}_{b,i}} \sum_{l=1}^{\tilde{n}_{b,i}} G_{\mathrm{B}}\Big(W_i(\vec{r}, \tau_k + l\delta\tau)\Big) E_L\Big(W_i(\vec{r}, \tau_k + l\delta\tau)\Big) \tag{2.75}$$

As the formalism required, setting $G_B = n_w = n_b = 1$ reproduce the VMC algorithm.

The new trial energy (remember Eq. (2.13)) is set to be equal to the previous cycle's average energy

$$E_T = E_k \tag{2.76}$$

[11] In literature, DMC is also known as *Projection Monte-Carlo*, for reasons described in Section 2.1.1.

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

$$E_{\text{DMC}} = \overline{E_T} = \frac{1}{n}\sum_{k=1}^{n} E_k \tag{2.77}$$

2.9.3 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, which is far more automatic.

Time Step Errors

The error introduced by the short time approximation goes as $\mathcal{O}(\delta\tau^2)$ (see Eq. (2.14)). 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. However, DMC is rarely used without importance sampling (Fokker-Planck), which, due to the Quantum Force, has an acceptance ratio ~ 1. It is therefore common to ignore this problem, and use a sufficiently low time step.

Selecting the Time-step

Studying the branching Green's function in equation 2.38 in more detail reveals that it's magnitude increase exponentially with the spread of the energies

$$G_B \propto \exp\left(\Delta E \delta\tau\right) \tag{2.78}$$

As will be shown in Section 2.11.1, 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. Due to finite computer memory, N slots in the memory are dedicated for storing walkers on every node. Too large branching factors may cause the system to max out the memory on one node before the walkers can be redistributed across all the nodes.

Setting an upper bound to the branching might seem like a good idea, however, with these fluctuations, this would imply an imbalance in the population drift, causing the walkers to slowly die out or reach a state not representing the desired projection.

The solution is to balance out the increase in ΔE by lowering the time-step accordingly. However, too low a time-step will hinder DMC to evolve walkers efficiently, especially if the positional span of the distribution is large. If the walkers are inefficient, that is, the trial energy converges slowly, the projection process will be equally slow. This has been verified by calculations in this thesis; slowly converging systems, such as low frequency quantum dots, will have densities which look far from the radially symmetric expected result if the time step or the number of walkers (the amount of statistics) is too low.

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

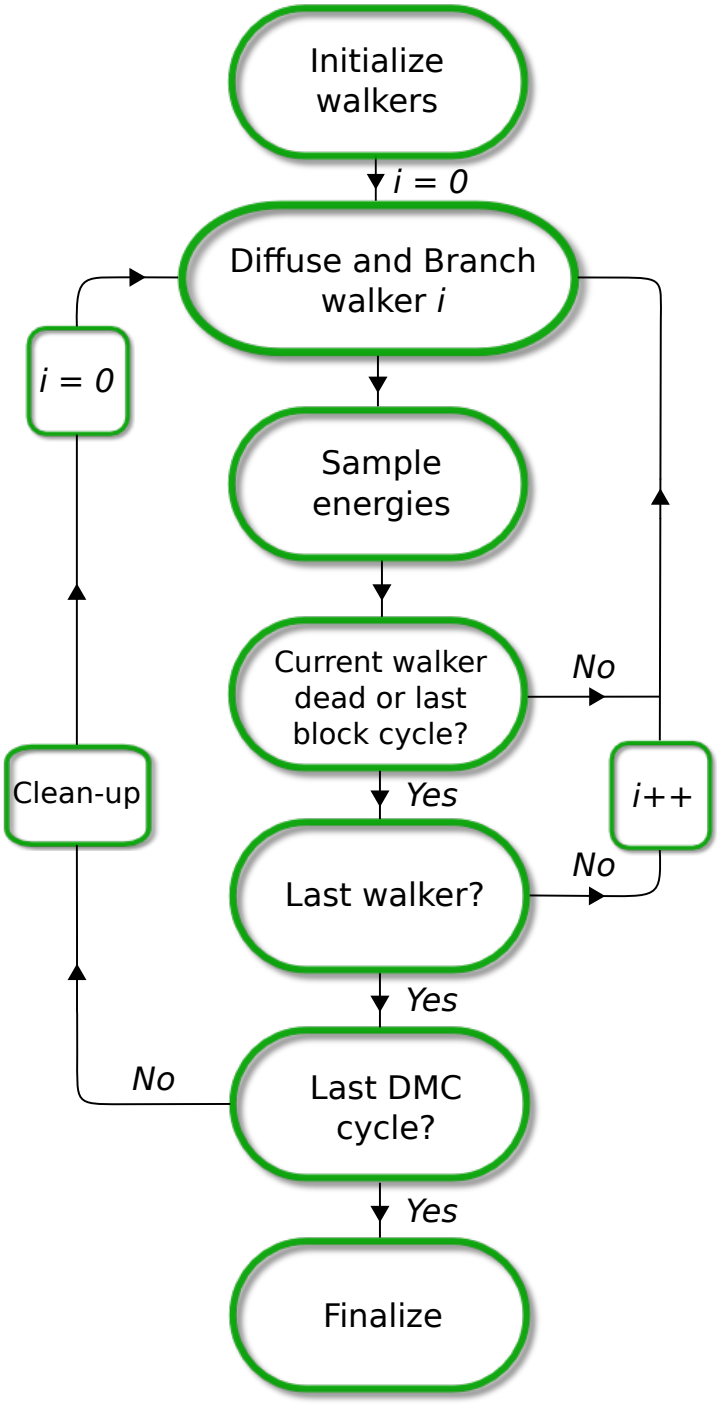

Figure 2.11: Chart flow of the Diffusion Monte-Carlo algorithm. The count variable i is the index of the walker loop. The second step, *Diffuse and Branch Walker*, is the process described in Fig. 2.1 in combination with the branching from Fig. 2.2. Energies are sampled as described in Section 2.9. Thermalization is not done in the same way as VMC (see Fig. 2.9), but rather includes the entire flow with a reduced number of DMC - and block cycles.

2.9.4 Fixed node approximation

Looking at Eq. (2.48), 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 interchange of two particles introduce 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 importance sampled DMC iterates, the density of walkers at a given time, $P(x,\tau)$, represents the projected wave function from Eq. (2.6) multiplied by the trial wave function (see the end of Section 2.2.2 for details)

$$P(\mathbf{r},\tau) = \Phi(\mathbf{r},\tau)\Psi_T(\mathbf{r}), \tag{2.79}$$

where theoretically using an exact projection operator

$$\lim_{\tau\to\infty} P(\mathbf{r},\tau) = \langle\Phi_0|\Phi_T\rangle\,\Phi_0(\mathbf{r})\Psi_T(\mathbf{r}), \tag{2.80}$$

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, i.e. the roots, 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[12]. 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:

$$\frac{\Psi_T(\mathbf{r}_i)}{\Psi_T(\mathbf{r}_j)} < 0 \quad \Longrightarrow \quad A(i \to j) = 0 \tag{2.81}$$

where $A(i \to j)$ is the probability of accepting the move, as described in Section 2.4. An illustrative example is attempted in Fig. 2.12.

2.10 Estimating One-body Densities

The one-body density is defined as

$$\rho(\mathbf{r}_1) = \iint_{\mathbf{r}_2\mathbf{r}_3} \ldots \int_{\mathbf{r}_N} |\Phi(\mathbf{r}_1\mathbf{r}_2\ldots\mathbf{r}_N)|^2 \,\mathrm{d}\mathbf{r}_2\ldots\mathrm{d}\mathbf{r}_N. \tag{2.82}$$

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 to find *any* of the systems 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 N, and not unity.

[12]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 infinity position space, this requires an enormous amount of walkers to succeed.

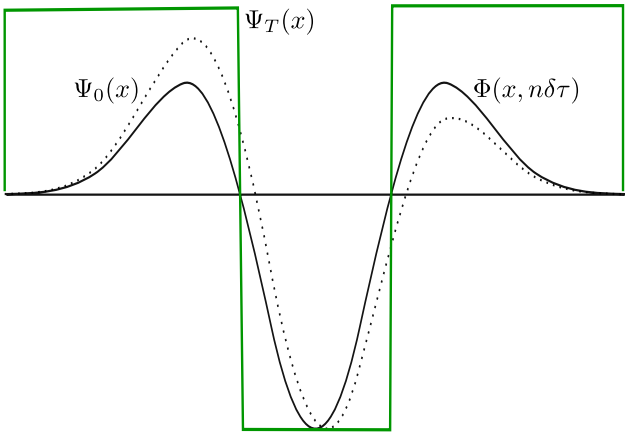

Figure 2.12: A one-dimensional illustration of the fixed node approximation. The dotted line is the exact ground state $\Psi_0(x)$. The distribution of walkers at cycle n, $\Phi(x, n\delta\tau)$, similar in shape with $\Psi_0(x)$, however, sharing nodes with the trial wave function $\Psi_T(x)$ (box-like function for illustration purposes), and thus making it impossible to match the true ground state exactly.

The one-body density integral corresponds to projecting every degree of freedom into one. In a Monte-Carlo simulation, estimating this quantity is done by collecting snapshots of the walkers' positions. These snapshots serve as samples to a histogram where each set of Cartesian coordinates (independent of particle number) give rise to one histogram-count ($\mathrm{d}\mathbf{r}_1$ is approximated by a finite value).

2.10.1 Estimating the Exact Ground State Density

The one-body density of the trial wave functions is in other words trivial; create a histogram of sampled VMC positions. The challenge is estimating the one-body density of the exact wave function, given a set of DMC data. As described in Section 2.2.2, the distribution of walkers by design spanning 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.

This implies the need of a method to transform the one-body density of $f(\mathbf{r}, \tau)$) into the one of $|\Phi(\mathbf{r}, \tau)|^2$ (from here on referred to as the *pure density*). This is done by using a third estimate obtained from VMC, the *variational density*, which corresponds to the one-body density of $|\Psi_T(\mathbf{r})|^2$. To achieve this, a relation shown in in Ref. [8] by Taylor expanding $\langle\Phi_0|\,\widehat{\mathbf{A}}\,|\Phi_0\rangle\,/\,\langle\Phi_0|\Phi_0\rangle$ around $\Delta \equiv \Psi_T(\mathbf{r}) - \Phi(\mathbf{r}, \tau)$ serves as a good starting ground

$$\begin{aligned}\langle A\rangle_0 = \frac{\langle\Phi_0|\,\widehat{\mathbf{A}}\,|\Phi_0\rangle}{\langle\Phi_0|\Phi_0\rangle} &\simeq 2\frac{\langle\Phi_0|\,\widehat{\mathbf{A}}\,|\Psi_T\rangle}{\langle\Phi_0|\Psi_T\rangle} - \frac{\langle\Psi_T|\,\widehat{\mathbf{A}}\,|\Psi_T\rangle}{\langle\Psi_T|\Psi_T\rangle} + \mathcal{O}(\Delta^2) \\ &= 2\langle A\rangle_{\mathrm{DMC}} - \langle A\rangle_{\mathrm{VMC}} + \mathcal{O}(\Delta^2) \end{aligned} \tag{2.83}$$

Expressed in terms of *density operators* (not one-body densities), the expectation values become

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

where tr denotes the *trace* (sum of eigenvalues[21]). Inserting these equation into Eq. (2.83) yields

$$\begin{aligned}\mathrm{tr}(\widehat{\rho}_0\widehat{\mathbf{A}}) &\simeq 2\mathrm{tr}(\widehat{\rho}_{\mathrm{DMC}}\widehat{\mathbf{A}}) - \mathrm{tr}(\widehat{\rho}_{\mathrm{VMC}}\widehat{\mathbf{A}}) + \mathcal{O}(\Delta^2) \\ &\simeq \mathrm{tr}\left((2\widehat{\rho}_{\mathrm{DMC}} - \widehat{\rho}_{\mathrm{VMC}})\,\widehat{\mathbf{A}}\right) + \mathcal{O}(\Delta^2)\end{aligned} \tag{2.84}$$

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

$$\widehat{\rho}_0 \simeq 2\widehat{\rho}_{\mathrm{DMC}} - \widehat{\rho}_{\mathrm{VMC}} \tag{2.85}$$

The transformation of densities into one-body densities can be applied to both sides of this equation, implying that combining the one-body densities of $f(\mathbf{r},\tau)$ from DMC and the trial wave function one from VMC according to Eq. (2.85) serves as a good approximation to the one-body density of the iterated wave function $\Phi(\mathbf{r},\tau)$ given that it is close to the exact wave function.

2.10.2 Radial Densities

In the case of symmetric wave functions, like in the case for atoms and quantum dots, the radial one-body density is obtained by integrating out the angular dependencies of the one-body density. This corresponds to weighing all angular directions equally in a brute for average and can only be applied to radially symmetric systems without loosing information.

Calculating the integral in two and three dimensions yields

$$\begin{aligned}I_{3\mathrm{D}} &= \iint \rho(\mathbf{r}_1,\theta_1,\phi_1) r_1^2 \sin\theta_1 \mathrm{d}\theta_1 \mathrm{d}\phi_1 \\ &\propto r_1^2 \rho(\mathbf{r}_1) \qquad (2.86) \\ I_{2\mathrm{D}} &= \int \rho(\mathbf{r}_1,\phi_1) r_1 \mathrm{d}\phi_1 \\ &\propto r_1 \rho(\mathbf{r}_1) \qquad (2.87)\end{aligned}$$

In practice, this integral is calculated by creating a histogram $H(r)$ of all sampled radii. Transforming this histogram into the radial one-body density $\rho(r_1)$ is according to Eq. (2.86) and Eq. (2.87) done in the following manner

$$\rho(\mathbf{r}_1) = \frac{H(\mathbf{r}_1)}{r_1^{(d-1)}} \tag{2.88}$$

where d denotes the number of dimensions. An example radial one-body density is given in Fig. 2.13

It should be noted that in the case of atoms, the one-body density does not reveal particularly interesting shapes (this will be presented in Figure **??**), however, simply showing the distribution of radii, that is, not dividing by $\mathbf{r}_1^2$ reveal interesting shapes which can be used to discuss the physics of different atoms.

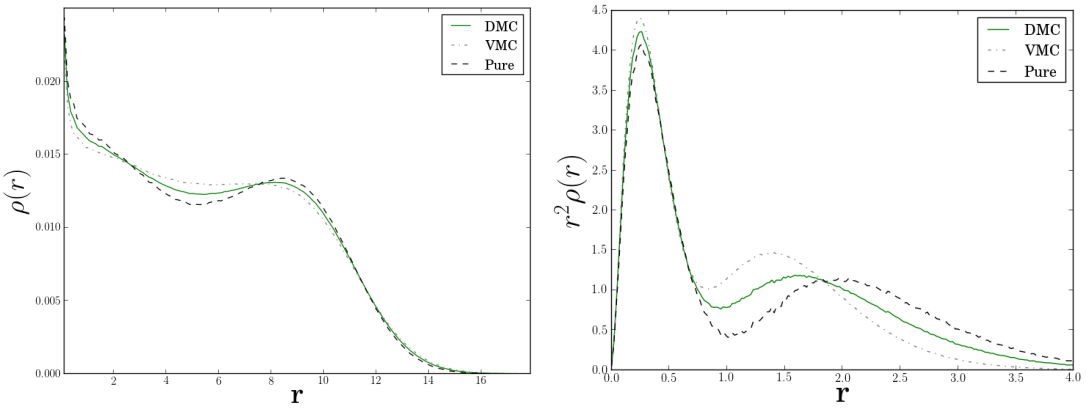

Figure 2.13: Two examples of radial one-body densities for both VMC, DMC (from $f(\mathbf{r}, \tau)$), and the pure density from Eq. (2.85). On the left: A 12-particle quantum dot with $\omega = 0.1$. The density diverges close to zero due to a "$\frac{0}{0}$" expression (see Eq. (2.88)). On the right: Unscaled radial density for the beryllium atom, i.e. $r_1^2\rho(\mathbf{r}_1)$. These results will be discussed in the results section.

2.11 Estimating the Statistical Error

As with any statistical result, the statistical errors needs to be supplied in order for it to be taken seriously. Systematic errors, that is, errors introduced due to limitations in the model, is discussed in each method's respective section, and will not be related to the statistical error.

Statistical errors, that is, the deviation from the true ensemble average due to the fact that the equality in Eq. (2.61) can never be fulfilled, 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. the samples are independent of each other.

2.11.1 The Variance and Standard Deviates

Given a set of samplings, e.g. local energies, the variance is a measure of their spread from the true mean value

$$\begin{aligned}
\mathrm{Var}(E) &= \left\langle (E - \langle E \rangle)^2 \right\rangle \\
&= \left\langle E^2 \right\rangle - 2\underbrace{\langle E \langle E \rangle \rangle}_{\langle E \rangle \langle E \rangle} + \langle E \rangle^2 \\
&= \left\langle E^2 \right\rangle - \langle E \rangle^2 \qquad (2.89) \\
&\simeq \overline{E^2} - \overline{E}^2 \qquad (2.90)
\end{aligned}$$

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 good of a fit different trial wave functions are to the system. 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 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. Beryllium simulations.

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,

$$\sigma^2(x) = \mathrm{Var}(x), \tag{2.91}$$

and has hence 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 $\langle E\rangle$.

2.11.2 The Covariance and correlated samples

It was briefly mentioned in the introduction that certain error estimation techniques was too naive in case of correlated samples. Two samples, x, y, are said to be correlated if their *covariance*, $\mathrm{Cov}(x,y)$, is non-zero

$$\begin{aligned}\mathrm{Cov}(x,y) &\equiv \langle(x-\langle x\rangle)(y-\langle y\rangle)\rangle \\ &= \langle xy - x\langle y\rangle - \langle x\rangle y + \langle x\rangle\langle y\rangle\rangle \\ &= \langle xy\rangle - \langle x\langle y\rangle\rangle \underbrace{- \langle y\langle x\rangle\rangle + \langle\langle x\rangle\langle y\rangle\rangle}_{0} \\ &= \langle xy\rangle - \langle x\rangle\langle y\rangle\,.\end{aligned} \tag{2.92}$$

Notice that $\mathrm{Cov}(x,x) = \mathrm{Var}(x)$. Using this definition, whether or not we have correlated samples boils down to whether or not $\langle xy\rangle = \langle x\rangle\langle y\rangle$.

The consequence of ignoring the correlations is a resulting error estimate which is generally less than the true error; correlated samplings are more clustered, i.e. less spread, due to previous samplings' influence on the value of the current sample[13]. Denoting the true standard deviation as σ_c, the above discussion can be distilled to

$$\sigma_c(x) \geq \sigma(x), \tag{2.93}$$

where $\sigma(x)$ is the deviation from Eq. (2.91).

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

2.11.3 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:

$$\sigma(\overline{x}) \neq \sigma(x). \tag{2.94}$$

Imagine doing a number of simulations, each resulting in a unique $\overline{x}$, the quantity of interest is not the deviation within a single simulation, but the deviation between the results of all the simulations.

$$\begin{aligned} m &\equiv \overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i && (2.95)\\ \sigma^2(m) &= \left\langle m^2 \right\rangle - \langle m \rangle^2 && (2.96) \end{aligned}$$

Combining the above equations yields

$$\begin{aligned} \sigma^2(m) &= \left\langle \frac{1}{n^2}\left[\sum_{i=1}^{n} x_i\right]^2 \right\rangle - \left\langle \frac{1}{n}\sum_{i=1}^{n} x_i \right\rangle^2 \\ &= \frac{1}{n^2}\left(\left\langle \sum_{i=1}^{n} x_i \sum_{j=1}^{n} x_j \right\rangle - \left\langle \sum_{i=1}^{n} x_i \right\rangle \left\langle \sum_{j=1}^{n} x_j \right\rangle \right) \\ &= \frac{1}{n^2}\sum_{i,j=1}^{n} \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle \\ &= \frac{1}{n^2}\sum_{i,j=1}^{n} \mathrm{Cov}(x_i, x_j) && (2.97) \end{aligned}$$

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. (2.92), it is apparent that in order to calculate the covariance as in Eq. (2.97), the true mean $\langle x_i \rangle$ needs to be known. Using $m = \overline{x}$ as a necessary approximation to the true mean yields

$$\begin{aligned} \mathrm{Cov}(x_i, x_j) &\equiv \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \\ &\simeq \langle (x_i - m)(x_j - m) \rangle \\ &\simeq \frac{1}{n^2}\sum_{k,l=1}^{n} (x_k - m)(x_l - m) && (2.98)\\ &\equiv \frac{1}{n}\mathrm{Cov}(x) && (2.99) \end{aligned}$$

Inserting this relation into Eq. (2.97) yields

$$\begin{aligned}\sigma^2(m) &= \frac{1}{n^2}\sum_{i,j=1}^{n}\mathrm{Cov}(x_i,x_j)\\ &\simeq \frac{1}{n^2}\sum_{i,j=1}^{n}\frac{1}{n}\mathrm{Cov}(x)\\ &= \frac{1}{n^3}\mathrm{Cov}(x)\underbrace{\sum_{i,j=1}^{n}}_{n^2}\\ &= \frac{1}{n}\mathrm{Cov}(x), \end{aligned} \tag{2.100}$$

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*[14], 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 $\overline{x}$

$$\overline{x} = \frac{1}{n}\sum_{k=0}^{n} x_{k\cdot\tau} \tag{2.101}$$

This implies that $n\tau$ samples are needed in order to get the same magnitude of samples to the average as in Eq. (2.95); the *effective sample size* becomes $n_{\text{eff}} = n_{\text{tot}}/\tau$. In the cases where $\tau = 1$, the sample set is uncorrelated. For details regarding the derivations of τ based on the covariance, see Refs. [22] and [12].

2.11.4 Blocking

Introducing correlation lengths in the system solver is 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. [22], however, details aside, the idea itself is quite intuitive: Given a data set of N samples from a single Monte-Carlo simulation, imagine dividing the dataset into *blocks* of n samples, that is, blocks of size $n_b = N/n$. The error σ_n in each block will naturally increase as n decrease, (see Eq. (2.100))

$$\sigma_n \propto \frac{1}{\sqrt{n}} \tag{2.102}$$

However, treating each block as an individual simulation, n_b averages m_n can be used to calculate the total error from Eq. (2.96), that is, estimate the covariance

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

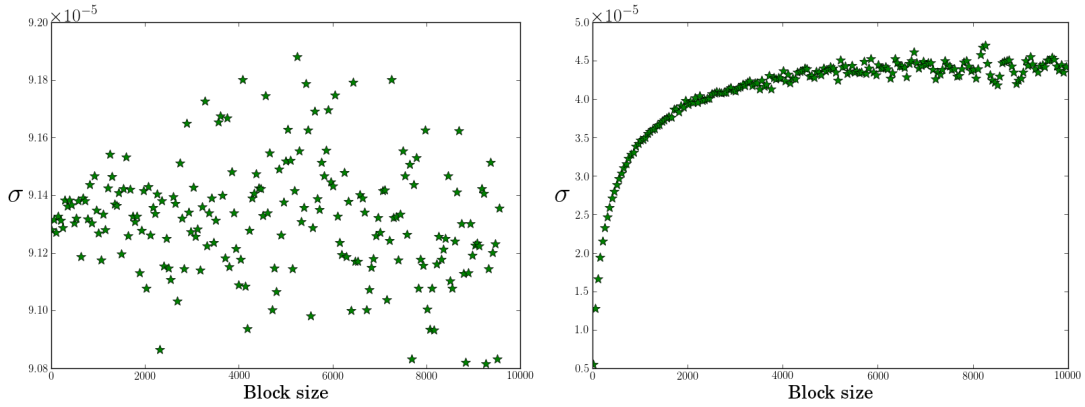

Figure 2.14: Left hand side: Blocking result of (approximately) uncorrelated data generated from a uniform Monte-Carlo integration of $\int_1^2 2x\mathrm{d}x$ resulting in 3.00003 (exact is 3.0). This is in excellent agreement with the magnitude of the error $\sim 9 \cdot 10^{-5}$. There is no sign of 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 $\omega = 0.1$ quantum dot. The plateau is strongly present, implying correlated data. The resulting total error is $\sim 4.5 \cdot 10^{-5}$.

$$\overline{m_n^r} \equiv \frac{1}{n_b}\sum_{k=1}^{n_b} m_k^r \tag{2.103}$$

$$\begin{aligned}\sigma^2(m) &= \left\langle m^2 \right\rangle - \left\langle m \right\rangle^2 \\ &\simeq \overline{m_n^2} - (\overline{m_n})^2 \end{aligned} \tag{2.104}$$

The approximation 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. (2.104) to reasonable precision.

The result of a blocking analysis is therefore a series of $(n,\, \sigma(m))$ 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 (where Eq. (2.104) is fulfilled). This plateau will then serve as a reasonable approximation to the covariance, that is, the true average. See Fig. 2.14 for a demonstration of resulting blocking plots.

2.11.5 Variance Estimators

The standard intuitive variance estimator

$$\sigma^2(x) \simeq \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2 = \left(\frac{1}{n}\sum_{i=1}^{n} x_i^2\right) - \overline{x}^2, \tag{2.105}$$

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

$$\sigma^2(x) \simeq \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2 = \left(\frac{1}{n-1}\sum_{i=1}^{n} x_i^2\right) - \frac{n}{n-1}\overline{x}^2, \tag{2.106}$$

which is only noticeably different from Eq. (2.105) when the sample size gets small, as it does in blocking analysis. It is therefore standard to use Eq. (2.106) for blocking errors.

3

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 solver involves several layers of complexity, such as support for different potentials, single particle bases, sampling models, etc., which may easily lead to a combinatoric disaster if the planning is not done right.

This chapter will begin by covering the underlaying assumptions regarding the modeled systems in Section 3.1.1 before listing the generalization goals in Section 3.1.2 and the optimization goals in Section 3.1.3. The strategies used to reach the generalization goals based on the assumptions will then be covered in Section 3.2, followed by several optimization schemes, some of which are a direct consequence of the initial assumptions, and some of which are more general.

3.1 Underlying Assumptions and Goals

Thousands of lines of code should be written once and for all. In industrial computational projects it is custom to plan every single part of the program before the coding begins, simply due to the fact that large scale object oriented frame works demands it. Brute-force coding massive frameworks almost exclusively result in unforeseen consequences, rendering the code difficult to expand, disorganized, if not completely defect. The code used in this thesis has been completely restructured four times. This section will cover the assumptions made and the goals set in the planning stages preliminary to the coding process.

3.1.1 Assumptions

The code scheme was laid down 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 modeled by all particles being in the same assumed single particle ground state.

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. These topics has been covered in Chapter 2.

3.1.2 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 solvers:

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

The challenge is, despite the vast array of combinations, to preserve simplicity and structure as layers of complexity are added. If-testing inside the solvers in order to achieve generalization is considered less optimal, and is only used if no other solution is apparent/exists.

3.1.3 Optimization Goals

Designed for the CPU, runtime optimizations are 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 if-tests in repeated function calls, but rather through polymorphism (see Section 1.2.4).

(iii) Linear scaling of runtime vs. the number of CPUs for large simulations.

3.2 Specifics Regarding Generalization

Generalization is achieved through the use of deep object orientation, i.e. polymorphism, rather than if-testing. These concepts are described in Section 1.2. The assumptions listed in Section 3.1.1 are applied if not otherwise is stated.

For details regarding the implementation of methods, see the code in [6].

3.2.1 Generalization Goals (i)-(vii)

As discussed in Section 2.6.1, the mathematical difference between fermions and bosons (of importance to QMC) is how the many-body wave functions are constructed from the single-particle basis. In the case of fermions, the expression is given in terms of two Slater determinants, while for bosons, it is simply the product of all states.

```
double Fermions::get_spatial_wf(const Walker* walker) {
    using namespace arma;

    //Spin up times Spin down (determinants)
    return det(walker->phi(span(0, n2 - 1), span())) * det(walker->phi(span(n2, n_p - 1),
         span()));
}
```

```
double Bosons::get_spatial_wf(const Walker* walker) {

    double wf = 1;

    //Using the phi matrix as a vector in the case of bosons.
    //Assuming all particles to occupy the same single particle state (neglecting
        permutations).
    for (int i = 0; i < n_p; i++){
      wf *= walker->phi(i);
    }

    return wf;
}
```

Fermion/Boson overloaded pure virtual methods exist for all methods involving evaluation of the many body wave function, e.g. the spatial ratio and the Laplacian sum. 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.

This way of splitting the system class also takes care of optimization goal (ii) in Section 3.1.3 regarding no use of repeating if-tests.

Similar polymorphic splitting is introduced in the following classes:

- `Orbitals` Hydrogen orbitals, harmonic oscillator, etc.
- `BasisFunctions` Stand-alone single particle wave functions. Initialized by `Orbitals`.
- `Sampling` Brute force - or importance sampling.
- `Diffusion` Isotropic or Fokker-Planck. Automatically selected by `Sampling`.
- `ErrorEstimator` Simple or Blocking.
- `Jastrow` Padé Jastrow - or no Jastrow factor.
- `QMC` VMC or DMC.
- `Minimizer` ASGD. Support for adding additional minimizers.

Implementing e.g. 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 1.2. The splitting done in `QMC` is done to avoid rewriting a lot of general QMC code.

A detailed description of the generalization of potentials, i.e. generalization goal (vii), is given in Section 1.2.4.

3.2.2 Generalization Goal (vi) and Expanded bases

An expanded single particle basis is implemented as a subclass of the `Orbitals` superclass. It is designed as a wrapper to an `Orbitals` subclass, e.g. harmonic oscillator, containing basis elements $\phi_\alpha(r_j)$. In addition to these elements, the class has a set of expansion coefficients $C_{\gamma\alpha}$ in which the new basis elements are constructed

$$\psi_\gamma^{\text{Exp.}}(r_j) = \sum_{\alpha=0}^{B-1} C_{\gamma\alpha}\phi_\alpha(r_j) \tag{3.1}$$

where B is the size of the expanded basis.

```
class ExpandedBasis : public Orbitals {

...

protected:

    int basis_size;
    arma::mat coeffs;
    Orbitals* basis;

    //Hartree-Fock
    void calculate_coefficients();

};
```

In order to calculate the expansion coefficients, a *Hartree-Fock* solver has been implemented. For a given single particle basis loaded in the `Orbitals* basis` member, everything that is needed in order to obtain an expanded basis, i.e. calculate the coefficients, is expressions for the one-body - and two-body interaction elements.

The implementation of Eq. (3.1) into the expanded basis class is achieved by overloading the original `Orbitals::phi` virtual member function

```
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;

}
```

The Hartree-Fock implementation at the present time is bugged, and has not been a focus for the thesis. The implementation has merely been done to lay the foundation in case future Master students are to expand upon the code. Hence Hartree-Fock theory will not be covered in detail, however, introductory theory can be found in Refs. [12, 23].

3.2.3 Generalization Goal (viii)

Functionality 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, `Orbitals::dell_phi`, which returns a single particle derivative, is virtual, and can be overloaded to call the numerical derivative implementation `Orbitals::num_diff`.The same goes for the Jastrow factor and the variational derivatives in the minimizers. Similar implementations exist for the Laplacian. An alternative to numerically evaluate the single particle wave function derivatives would be to perform the derivative on the full many-body wave function, however, this would demand another layer of polymorphism and make the code all-around less intuitive.

The implemented numerical derivatives are finite difference schemes with error proportional to the squared step length.

3.3 Optimizations due to a Single two-level Determinant

Assumption (iii) unlocks the possibility to optimize the expressions involving the Slater-determinant dramatically. Similar optimizations for bosons due to assumption (iv) are considered trivial and will not be covered in detail. See the code in [6] for details regarding bosons.

The full trial wave function Ψ_T is given by Eq. (2.54). The function arguments will be skipped in order to clean up the expressions. Written in terms of a spatial function $|D|$, which is split into a spin up -and spin down part by Eq. (2.52), and a Jastrow function J, the trial wave function reads

$$\Psi_T = |D^{\uparrow}||D^{\downarrow}|J \tag{3.2}$$

The Quantum Force becomes

$$\begin{aligned}
\mathbf{F}_i &= 2\frac{\nabla_i \left(|D^\uparrow||D^\downarrow|J\right)}{|D^\uparrow||D^\downarrow|J} \\
&= 2\left(\frac{\nabla_i|D^\uparrow|}{|D^\uparrow|} + \frac{\nabla_i|D^\downarrow|}{|D^\downarrow|} + \frac{\nabla_i J}{J}\right) \\
&= 2\left(\frac{\nabla_i|D^\alpha|}{|D^\alpha|} + \frac{\nabla_i J}{J}\right) \qquad (3.3)
\end{aligned}$$

where α is the spin configuration of particle i. The counterpart to α is independent of particle i and will be zero in the expression above.

Equally for the Laplacian used in the local energy, the result becomes

$$E_L = \sum_i \frac{1}{\Psi_T}\nabla_i^2\Psi_T + V \qquad (3.4)$$

$$\begin{aligned}
\frac{1}{\Psi_T}\nabla_i^2\Psi_T &= \frac{1}{|D^\uparrow||D^\downarrow|J}\nabla_i^2|D^\uparrow||D^\downarrow|J \\
&= \frac{\nabla_i^2|D^\uparrow|}{|D^\uparrow|} + \frac{\nabla_i^2|D^\downarrow|}{|D^\downarrow|} + \frac{\nabla_i^2 J}{J} \\
&\quad + 2\frac{\left(\nabla_i|D^\uparrow|\right)\left(\nabla_i|D^\downarrow|\right)}{|D^\uparrow||D^\downarrow|} + 2\frac{\left(\nabla_i|D^\uparrow|\right)\left(\nabla_i J\right)}{|D^\uparrow|J} + 2\frac{\left(\nabla_i|D^\downarrow|\right)\left(\nabla_i J\right)}{|D^\downarrow|J} \\
&= \frac{\nabla_i^2|D^\alpha|}{|D^\alpha|} + \frac{\nabla_i^2 J}{J} + 2\frac{\nabla_i|D^\alpha|}{|D^\alpha|}\frac{\nabla_i J}{J} \qquad (3.5)
\end{aligned}$$

Finally, the expression for the R_ψ ratio for Metropolis is

$$\begin{aligned}
R_\psi &= \frac{\Psi_T^{\text{new}}}{\Psi_T^{\text{old}}} \\
&= \frac{|D^\uparrow|^{\text{new}}|D^\downarrow|^{\text{new}}J^{\text{new}}}{|D^\uparrow|^{\text{old}}|D^\downarrow|^{\text{old}}J^{\text{old}}} \\
&= \frac{|D^\alpha|^{\text{new}}}{|D^\alpha|^{\text{old}}}\frac{J^{\text{new}}}{J^{\text{old}}} \qquad (3.6)
\end{aligned}$$

where either the spin up or the spin down determinant is unchanged by the step, i.e. $|D^{\overline{\alpha}}|^{\text{new}} = |D^{\overline{\alpha}}|^{\text{old}}$, where $\overline{\alpha}$ denotes the opposite spin of α.

From these expressions it is clear that the dimensionality of the calculations is halved by splitting the Slater determinants. Calculation the determinant of a $N \times N$ matrix is $\mathcal{O}(N^2)$ floating point operations (flops). This implies a speedup of four in estimating the determinants alone.

3.4 Optimizations due to Single-particle Moves

Moving one particle at the time (see the diffusion algorithm in Fig. 2.1), means changing only a single row in the Slater-determinant at the time. Changes to a single row implies that many *co-factors* remain unchanged. Since all of the expressions deduced in the previous section contains ratios of the spatial wave functions, expressing these determinants in terms of their co-factors should reveal a cancellation of terms.

3.4.1 Optimizing the Slater ratios

The inverse of the Slater-matrix is given in terms of its *adjugate* by the following relation [24] (spin-configuration parameter α will be skipped for now)

$$D^{-1} = \frac{1}{|D|}\text{adj}D$$

The adjugate of a matrix is the transpose of the cofactor matrix C. The expression in terms of matrix element equations reads

$$\begin{aligned} D^{-1}_{ij} &= \frac{C_{ji}}{|D|} && (3.7)\\ D_{ji} &= \phi_i(r_j) && (3.8) \end{aligned}$$

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

$$|D| = \sum_i D_{ji}C_{ji}. \tag{3.9}$$

The spatial part of the R_ψ ratio is obtained by inserting Eq. (3.9) into Eq. (3.6)

$$R_S = \frac{\sum_i D^{\text{new}}_{ji}C^{\text{new}}_{ji}}{\sum_i D^{\text{old}}_{ji}C^{\text{old}}_{ji}} \tag{3.10}$$

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

$$C^{\text{new}}_{ji} = C^{\text{old}}_{ji} = (D^{\text{old}}_{ij})^{-1}|D^{\text{old}}|, \tag{3.11}$$

where the inverse relation of Eq. (3.7) has been used. Inserting this into Eq. (3.10) yields

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

The diagonal element of the identity matrix is by definition unity. Inserting this fact combined with the relation from Eq. (3.8) yields the optimized expression for the ratio

$$R_S = \sum_i \phi_i(r^{\text{new}}_j)(D^{\text{old}}_{ij})^{-1} \tag{3.12}$$

where j is the currently moved particle. The sum i spans the Slater matrix whose spin value matches that of particle j.

Similar reductions can be applied to all the Slater ratio expressions from the previous section, see refs. [8, 12]:

$$\frac{\nabla_i |D|}{|D|} = \sum_k \nabla_i \phi_k(r_i^{\text{new}})(D_{ki}^{\text{new}})^{-1} \tag{3.13}$$

$$\frac{\nabla_i^2 |D|}{|D|} = \sum_k \nabla_i^2 \phi_k(r_i^{\text{new}})(D_{ki}^{\text{new}})^{-1} \tag{3.14}$$

where k spans the Slater matrix whose spin values match that of the moved particle. Unlike for the ratio, $N/2$ of the gradients needs to be recalculated once a particle is moved (with N being the number of particles). This is due to the fact that it is the new Slater inverse that is used, and not the old.

Closed form expressions for the derivatives and Laplacians of the single particle may be implemented and accessed when calling these functions to avoid expensive numerical calculations. See Appendices B, C and D for a tabulation of closed form expressions. Appendix A presents an efficient framework for obtaining these expressions.

3.4.2 Optimizing the Inverse

One might question the efficiency of calculating inverse matrices compared to brute force estimation of the determinants. However, as for the ratio in Eq. (3.12), using co-factor expansions, an updating algorithm which dramatically decreases the cost of calculating the inverse of the new Slater matrix can be implemented.

Letting i denote the currently moved particle, the new inverse is given in terms of the old by the following expression [8, 12]

$$\tilde{I}_{ij} = \sum_l D_{il}^{\text{new}} (D_{lj}^{\text{old}})^{-1} \tag{3.15}$$

$$(D_{kj}^{\text{new}})^{-1} = (D_{kj}^{\text{old}})^{-1} - \frac{1}{R_S}(D_{ji}^{\text{old}})^{-1} \tilde{I}_{ij} \qquad j \neq i \tag{3.16}$$

$$(D_{ki}^{\text{new}})^{-1} = \frac{1}{R_S}(D_{ki}^{\text{old}})^{-1} \qquad \text{else} \tag{3.17}$$

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

Further optimization can be achieved by calculating the $\tilde{I}$ 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.

3.4.3 Optimizing the Padé Jastrow factor Ratio

As done with the Green's function ratio in Eq. (2.36), the ratio between two Jastrow factors are best calculating as exponentiation the logarithm

$$\log \frac{J^{\text{new}}}{J^{\text{old}}} = \sum_{k<j=1}^{N} \frac{a_{kj} r_{kj}^{\text{new}}}{1+\beta r_{kj}^{\text{new}}} - \frac{a_{kj} r_{kj}^{\text{old}}}{1+\beta r_{kj}^{\text{old}}} \tag{3.18}$$

$$\equiv \sum_{k<j=1}^{N} g_{kj}^{\text{new}} - g_{kj}^{\text{old}} \tag{3.19}$$

The relative distances r_{kj} behave much like the cofactors in Section 3.4.1: Changing r_i only changes r_{ij}, that is

$$r_{kj}^{\text{new}} = r_{kj}^{\text{old}} \qquad k \neq i \tag{3.20}$$

which inserted into Eq. (3.19) yields

$$\begin{aligned} \log \frac{J^{\text{new}}}{J^{\text{old}}} &= \sum_{k<j\neq i} g_{kj}^{\text{old}} - g_{kj}^{\text{old}} + \sum_{j=1}^{N} g_{ij}^{\text{new}} - g_{ij}^{\text{old}} \\ &= \sum_{j=1}^{N} a_{ij} \left(\frac{r_{ij}^{\text{new}}}{1+\beta r_{ij}^{\text{new}}} - \frac{r_{ij}^{\text{old}}}{1+\beta r_{ij}^{\text{old}}} \right) \end{aligned} \tag{3.21}$$

Exponentiating both sides reveals the final optimized ratio

$$\frac{J^{\text{new}}}{J^{\text{old}}} = \exp\left[\sum_{j=1}^{N} a_{ij} \left(\frac{r_{ij}^{\text{new}}}{1+\beta r_{ij}^{\text{new}}} - \frac{r_{ij}^{\text{old}}}{1+\beta r_{ij}^{\text{old}}} \right) \right] \tag{3.22}$$

3.5 Optimizing the Padé Jastrow Derivative Ratios

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

3.5.1 The Gradient

Using the notation of Eq. (3.19), the x component of the Padé Jastrow gradient ratio for particle i is

$$\frac{1}{J}\frac{\partial J}{\partial x_i} = \frac{1}{\prod_{k<l} \exp g_{kl}} \frac{\partial}{\partial x_i} \prod_{k<l} \exp g_{kl} \tag{3.23}$$

Using the product rule, only terms with k or l equal to i survive the differentiation. In addition, the terms independent of i will cancel the corresponding terms in the denominator. In other words,

$$\begin{aligned}\frac{1}{J}\frac{\partial J}{\partial x_i} &= \sum_{k\neq i}\frac{1}{\exp g_{ik}}\frac{\partial}{\partial x_i}\exp g_{ik} \\ &= \sum_{k\neq i}\frac{1}{\exp g_{ik}}\exp g_{ik}\frac{\partial g_{ik}}{\partial x_i} \\ &= \sum_{k\neq i}\frac{\partial g_{ik}}{\partial x_i} \\ &= \sum_{k\neq i}\frac{\partial g_{ik}}{\partial r_{ik}}\frac{\partial r_{ik}}{\partial x_i},\end{aligned} \tag{3.24}$$

where

$$\begin{aligned}\frac{\partial g_{ik}}{\partial r_{ik}} &= \frac{\partial}{\partial r_{ik}}\left(\frac{a_{ik}r_{ik}}{1+\beta r_{ik}}\right) \\ &= \frac{a_{ik}}{1+\beta r_{ik}} - \frac{a_{ik}r_{ik}}{(1+\beta r_{ik})^2}\beta \\ &= \frac{a_{ik}(1+\beta r_{ik}) - a_{ik}\beta r_{ik}}{(1+\beta r_{ik})^2} \\ &= \frac{a_{ik}}{(1+\beta r_{ik})^2},\end{aligned} \tag{3.25}$$

and

$$\begin{aligned}\frac{\partial r_{ik}}{\partial x_i} &= \frac{\partial}{\partial x_i}\sqrt{(x_i-x_k)^2+(y_i-y_k)^2+(z_i-z_k)^2} \\ &= \frac{1}{2}2(x_i-x_k)/\sqrt{(x_i-x_k)^2+(y_i-y_k)^2+(z_i-z_k)^2} \\ &= \frac{x_i-x_k}{r_{ik}}.\end{aligned} \tag{3.26}$$

Combining these expressions yields

$$\frac{1}{J}\frac{\partial J}{\partial x_i} = \sum_{k\neq i}\frac{a_{ik}}{r_{ik}}\frac{x_i-x_k}{(1+\beta r_{ik})^2} \tag{3.27}$$

When changing Cartesian variable in the differentiation, the only change to the expression is that the corresponding Cartesian variable changes in the numerator of Eq. (3.27). In other words, generalizing to the full gradient is done by substituting the Cartesian difference with the position vector difference.

$$\frac{\nabla_i J}{J} = \sum_{k\neq i=1}^{N}\frac{a_{ik}}{r_{ik}}\frac{\vec{r}_i-\vec{r}_k}{(1+\beta r_{ik})^2} \tag{3.28}$$

3.5.2 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. [12]. The expression becomes

$$\frac{\nabla_i^2 J}{J} = \left|\frac{\nabla_i J}{J}\right|^2 + \sum_{k\neq i=1}^{N}\left(\frac{d-1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}} + \frac{\partial^2 g_{ik}}{\partial r_{ik}^2}\right) \tag{3.29}$$

where d is the number of dimensions, arising due to the fact that the Laplacian, unlike the gradient, is a summation of contributions from all dimensions. A simple differentiation of Eq. (3.25) with respect to r_{ik} yields

$$\frac{\partial^2 g_{ik}}{\partial r_{ik}^2} = -\frac{2a_{ik}\beta}{(1+\beta r_{ik})^3} \tag{3.30}$$

Inserting Eq. (3.25) and Eq. (3.30) into Eq. (3.29) yields

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

which when cleaned up results in

$$\frac{\nabla_i^2 J}{J} = \left|\frac{\nabla_i J}{J}\right|^2 + \sum_{k\neq i} a_{ik}\frac{(d-3)(\beta r_{ik}+1)+2}{r_{ik}(1+\beta r_{ik})^3} \tag{3.31}$$

The local energy calculation needs the sum of the Laplacians for all particles (see Eq. (3.4)). In other words, the quantity of interest becomes

$$\sum_i \frac{\nabla_i^2 J}{J} = \sum_i \left[\left|\frac{\nabla_i J}{J}\right|^2 + \sum_{k\neq i}^{N} a_{ik}\frac{(d-3)(\beta r_{ik}+1)+2}{r_{ik}(1+\beta r_{ik})^3}\right] \tag{3.32}$$

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

$$\sum_i \frac{\nabla_i^2 J}{J} = \sum_i \left|\frac{\nabla_i J}{J}\right|^2 + 2\sum_{k>i} a_{ik}\frac{(d-3)(\beta r_{ik}+1)+2}{r_{ik}(1+\beta r_{ik})^3} \tag{3.33}$$

3.6 Tabulating Recalculated Data

The optimizations covered in this section will exclusively arise from point (i) in section 3.1.3. Avoiding recalculating expressions also include exploiting symmetries, such as was done in the Padé Jastrow Laplacian in Eq. (3.33).

Code examples are presented to narrow the gap between presented optimizations and practical implementations in order to demonstrate that most optimizations does not require much additional code.

3.6.1 The relative distance matrix

In the discussions regarding the optimization of the Jastrow ratio, it became clear that moving one particle only changed N of the relative distances. Storing these values in a matrix r_{rel}, the row and column representing the moved particle can be updated, ensuring that the relative distances are calculated once and for all.

$$r_{\text{rel}} = r_{\text{rel}}^T = \begin{pmatrix} 0 & r_{12} & r_{13} & \cdots & r_{1N} \\ & 0 & r_{23} & \cdots & r_{2N} \\ & & \ddots & \ddots & \vdots \\ & \cdots & & 0 & r_{(N-1)N} \\ & & & & 0 \end{pmatrix}. \tag{3.34}$$

```
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 }
```

Functions such as `Coulomb::get_potential_energy` and all of the Jastrow functions can then simply access these matrix elements.

Similar storage has been done for the squared distance vector in all cases and the length of the distance vector in case of atomic orbitals.

3.6.2 The Slater related matrices

Apart from the inverse, whose optimization was covered in Section 3.4.2, calculating the single particle wave functions and its gradients are the most expensive operations of the QMC algorithm.

Storing these function values in a matrices representing the Slater Matrix and its derivatives will ensure that these values never gets recalculated.

$$D \equiv \left[D^{\uparrow}\, D^{\downarrow}\right] = \begin{bmatrix} \phi_1(r_0) & \phi_1(r_1) & \cdots & \phi_1(r_N) \\ \phi_2(r_0) & \phi_2(r_1) & \cdots & \phi_2(r_N) \\ \vdots & \vdots & & \vdots \\ \phi_{N/2}(r_0) & \phi_{N/2}(r_1) & \cdots & \phi_{N/2}(r_N) \end{bmatrix} \tag{3.35}$$

$$\nabla D \equiv \left[\nabla D^{\uparrow}\, \nabla D^{\downarrow}\right] = \begin{bmatrix} \nabla\phi_1(r_0) & \nabla\phi_1(r_1) & \cdots & \nabla\phi_1(r_N) \\ \nabla\phi_2(r_0) & \nabla\phi_2(r_1) & \cdots & \nabla\phi_2(r_N) \\ \vdots & \vdots & & \vdots \\ \nabla\phi_{N/2}(r_0) & \nabla\phi_{N/2}(r_1) & \cdots & \nabla\phi_{N/2}(r_N) \end{bmatrix} \tag{3.36}$$

3.6.3 Avoiding spin tests

Since the slater determinant is split by spin eigenvalues, the same splitting occurs in the inverse, the Slater matrix etc. The brute force implementation is to perform an if-test to decide which of the matrices to access. In the case of a two-level system we get

$$\begin{array}{ll} i < N/2 & D^{\uparrow}(i) \\ i \geq N/2 & D^{\downarrow}(i - N/2) \end{array} \tag{3.37}$$

However, simply concatenating the Slater related matrices solves the entire problem. This has already been done in Eq. (3.35) and Eq. (3.36). Applying this to the inverse yields

$$D^{-1} \equiv \left[(D^{\uparrow})^{-1} \, (D^{\downarrow})^{-1} \right] \tag{3.38}$$

When the index representing the moved particle succeeds $N/2 - 1$, the values representing the opposite spin is automatically accessed, which means that no if-tests is required in order to sort the spin splitting. In e.g. the updating algorithm for the inverse, simply keeping track of whether to start at $k = 0$ or $k = N/2$ solves the problem. This "start" parameter can be calculated once every particle move, and can then be used throughout the program.

3.6.4 The Padé Jastrow gradient

Consider Eq. (3.28). Just as for the Jastrow Laplacian, there are (anti)symmetries in the expression, which implies an optimized way of calculating the gradient. However, unlike the Laplacian, the gradient is split into components, which makes the exploitation of symmetries a little less straight-forward.

Defining

$$\mathrm{d}\mathbf{J}_{ik} \equiv \frac{a_{ik}}{r_{ik}} \frac{\vec{r}_i - \vec{r}_k}{(1 + \beta r_{ik})^2} = -\mathrm{d}\mathbf{J}_{ki}, \tag{3.39}$$

the gradient can be written in a more compact form

$$\frac{\nabla_i J}{J} = \sum_{k \neq i = 1}^{N} \mathrm{d}\mathbf{J}_{ik}. \tag{3.40}$$

As for the relative distances, storing the elements and exploiting the symmetry properties, only half the total elements needs to be calculated.

$$\mathrm{d}J \equiv \begin{pmatrix} 0 & \mathrm{d}\mathbf{J}_{12} & \mathrm{d}\mathbf{J}_{13} & \cdots & \mathrm{d}\mathbf{J}_{1N} \\ & 0 & \mathrm{d}\mathbf{J}_{23} & \cdots & \mathrm{d}\mathbf{J}_{2N} \\ & & \ddots & \ddots & \vdots \\ & (-) & & 0 & \mathrm{d}\mathbf{J}_{(N-1)N} \\ & & & & 0 \end{pmatrix} = -\mathrm{d}J^T. \tag{3.41}$$

```
void Pade_Jastrow::get_dJ_matrix(Walker* walker, int i) const {

    for (int j = 0; j < n_p; j++) {
        if (j == i) continue;

        b_ij = 1.0 + beta * walker->r_rel(i, j);
        factor = a(i, j) / (walker->r_rel(i, j) * b_ij * b_ij);
        for (int k = 0; k < dim; k++) {
            walker->dJ(i, j, k) = (walker->r(i, k) - walker->r(j, k)) * factor;
            walker->dJ(j, i, k) = -walker->dJ(i, j, k);
        }
    }
}
```

Calculating the gradient is now only a matter of summing the rows of the matrix in Eq. (3.41). Further optimization can be achieved by realizing that the function has access to the gradient of the previous iteration

$$\frac{\nabla_i J^{\text{old}}}{J^{\text{old}}} = \sum_{k\neq i=1}^{N} \mathrm{d}\mathbf{J}_{ik}^{\text{old}} \tag{3.42}$$

$$\frac{\nabla_i J^{\text{new}}}{J^{\text{new}}} = \sum_{k\neq i=1}^{N} \mathrm{d}\mathbf{J}_{ik}^{\text{new}} \tag{3.43}$$

By moving particle p in QMC, only a single row and column of the $\mathrm{d}J$ matrix changes. Assuming that $i \neq p$, only a single term from the new matrix is required

$$\frac{\nabla_{i\neq p} J^{\text{new}}}{J^{\text{new}}} = \sum_{k\neq i\neq p} \mathrm{d}\mathbf{J}_{ik}^{\text{old}} + \mathrm{d}\mathbf{J}_{ip}^{\text{new}} \tag{3.44}$$

$$= \left[\sum_{k\neq i\neq p} \mathrm{d}\mathbf{J}_{ik}^{\text{old}} + \mathrm{d}\mathbf{J}_{ip}^{\text{old}}\right] - \mathrm{d}\mathbf{J}_{ip}^{\text{old}} + \mathrm{d}\mathbf{J}_{ip}^{\text{new}} \tag{3.45}$$

$$= \frac{\nabla_i J^{\text{old}}}{J^{\text{old}}} - \mathrm{d}\mathbf{J}_{ip}^{\text{old}} + \mathrm{d}\mathbf{J}_{ip}^{\text{new}} \tag{3.46}$$

reducing the calculation to three flops. For the case $i = p$, the entire sum as in Eq. (3.43) must be calculated. This is demonstrated in the following code

```
void Pade_Jastrow::get_grad(const Walker* walker_pre, Walker* walker_post, int p) const {
    double sum;

    for (int i = 0; i < n_p; i++) {
        if (i == p) {

            //for i == p the entire sum needs to be calculated
            for (int k = 0; k < dim; k++) {

                sum = 0;
                for (int j = 0; j < n_p; j++) {
                    sum += walker_post->dJ(p, j, k);
                }

                walker_post->jast_grad(p, k) = sum;
            }

        } 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  }
```

Due to dJ being a 3-dimensional matrix (tensor), this optimization scales very well with increasing number of particles.

3.6.5 The single-particle Wave Functions

A single particle wave function is defined by a position in space r and a quantum number q. A QMC walker holds the position of all particles, so in the case of QMC, the parameters defining the single particle wave function is a walker, a particle number i and of course the quantum number (for the gradient we also need the dimension).

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 if-test on the quantum number, and calculate the corresponding expression inside the correct if-test.

```
1  double AlphaHarmonicOscillator::phi(const Walker* walker, int particle, int q_num) {
2
3      //Ground state of the harmonic oscillator
4      if (q_num == 0){
5        return exp(-0.5*w*walker->get_r_i2(i));
6      }
7
8      ...
9
10 }
```

This is however inefficient when the number of particles numbers become large. A more efficient implementation is to implement the single particle wave function expressions as `BasisFunctions` subclasses. `BasisFunctions` has one pure virtual function, *eval*, which then only needs the particle number i and the walker. The class itself is defined by the quantum number.

The following is an example of the harmonic oscillator single particle wave function for quantum number $q = 1$.

```
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(-0.5*w*walker->get_r_i2(i));
9
10 }
```

These objects representing single particle wave functions can be loaded into an array, such that element q corresponds to the `BasisFunctions` object representing this quantum number. The new `Orbitals` function then becomes

```
1  double Orbitals::phi(const Walker* walker, int particle, int q_num) {
2      return basis_functions[q_num]->eval(walker, particle);
3  }
```

with no if-tests required.

Other more specific optimizations can be implemented, so called *quantum number independent factors*, $\overline{Q}_i$. When moving particle p, as discussed previously, a single column in the Slater matrix from Eq. (3.35) needs to be updated. All these terms are calculated in the same position. This implies that terms independent of the quantum number will be equal. Looking at the single particle states for harmonic oscillator and hydrogen listed in the Appendix, their exponential form results in that an exponential factor independent of the quantum number is present in all terms.

$$\overline{Q}_i^{\text{H.O.}} = e^{-\frac{1}{2}\alpha\omega r_i^2} \tag{3.47}$$
$$\overline{Q}_i^{\text{Hyd.}} = e^{-\frac{1}{n}\alpha Z r_i} \tag{3.48}$$

For hydrogen, we have a dependence on the principle quantum number n, however, for e.g $n = 2$, 20 terms share this exponential factor. Calculating it once and for all saves 19 exponential calls pr. particle pr. walker pr. cycle etc.

The implementation is rather simple. Upon moving a particle, the virtual function `Orbitals.set_qnum_indie_terms` is called, updating the value of e.g. an `exp_factor` pointer shared by all the loaded `BasisFunctions` objects and the `Orbitals` class.

```
void AlphaHarmonicOscillator::set_qnum_indie_terms(const Walker * walker, int i) {

   //k2 = alpha*omega
   *exp_factor = exp(-0.5 * (*k2) * walker->get_r_i2(i));
}

void hydrogenicOrbitals::set_qnum_indie_terms(const Walker* walker, int i) {

    //k = alpha*Z
    *exp_factor_n1 = exp(-(*k) * walker->get_r_i(i));
    *exp_factor_n2 = exp(-(*k) * walker->get_r_i(i) / 2);
    ...

}
```

The `BasisFunctions` objects can then simply access this value instead of calculating the exponential

```
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_factor);

}

double lapl_hydrogenic_0::eval(const Walker* walker, int i) {

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

    psi = (*k)*((*k)*walker->get_r_i(i) - 2)/walker->get_r_i(i);
    return psi*(*exp_factor);

}
```

This optimization is an enormous speedup for many particle simulations.

All single particle expressions given are calculated using *SymPy*. See Appendix **REF SYMPY** for details. For Quantum Dots, 84 `BasisFunctions` objects are needed for a 42-particle simulation, reducing

the number of exponential calls with 83 pr. particle pr. walker pr. cycle, which for an average DMC calculation results in $24 \cdot 10^9$ saved exponential calls.

3.7 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 exist in order to work around this, however, keeping the cache in mind from the beginning of the coding process can result in a much faster code. In the case of the QMC code, the most optimal use of the cache would be to have complete walkers ready 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. This is ensured by not using pointers within the walker objects, as pointers are free to be declared in any memory address.

3.7.1 Disadvantage of Generalizing the code

The size of the matrices within the walker objects are dynamically allocated in order to handle any number of particles and dimensions. At compile-time, there is no telling how much memory each walker will demand, and thus it is harder for the compiler to optimize the cache usage.

The alternative is to statically declare, i.e. declare with a fixed size known at compile time, the matrices to be of the maximum possible simulation size. This would however waste a ton of memory, and would render most applications impossible to run at a standard computer. A second alternative would be to re-compile the code every time the system variables are changed. This is not optimal in the case of a generalized solver.

3.7.2 Consequences

The QMC code has support for both kinds of scenarios. Compiling the source code with a main file which fixes the system variables will result in a more efficient executable. For the purpose of this thesis, this was not done; all system variables are controlled via a configuration script (see `runQMC.py` in ref. [6]).

Part II

Results

Appendices

A

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

The focus on 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 optimize to have these expressions in order for the code to remain efficient.

Calculating these expressions by hand is out of the question, given that the complexity of the expressions is proportional to the magnitude of the quantum number, which again scales with the number of particles. In the case of a 42 particle Quantum Dot, the number of unique derivatives involved in the simulation is 84.

A.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.

A.1.1 Doing Symbolic Algebra

In order for SymPy to recognize e.g. `x` as a symbol, that is, a *mathematical variable*. 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 [7]: x*x+y
Out[7]: '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. 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 [7]: x2+y #x2 is printed more nicely given a describing label
Out[7]: 'x^2 + y'

In [8]: im(z) #Imaginary part cannot be assumed to be anything.
Out[8]: 'im(z)'

In [9]: im(x2) #Flagged as real, the imaginary part is zero.
Out[9]: 0
```

A.1.2 Exporting C++ and Latex Code

Exporting code is extremely simple. 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

$$\frac{x^4}{e^{x^2}}$$

A.1.3 Calculating Derivatives

As an example for this section, let's look at the $2s$ orbital from hydrogen (not normalized)

$$\phi_{2s}(\vec{r}) = (Zr - 2)e^{-\frac{1}{2}Zr} \tag{A.1}$$

$$r^2 = x^2 + y^2 + z^2 \tag{A.2}$$

Calculating the gradients and Laplacian is very simply by using the `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. 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() #Gathers all the terms nicely
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. Result:

$$-\frac{Zx\left(Zr-4\right)}{2re^{\frac{1}{2}Zr}}$$

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.

A.2 Using the auto-generation Script

The Python superclass `orbitalsGenerator` (the generator) aims to serve as a interface with the 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:

`constructor`	Takes argument `M` equal to the number of orbitals to compute, should directly be set by `self.setMax(M)`. Second argument `doInit` defaults to `True`. Should be sent to superclass constructor. If false, does not initialize the construction of expression, latex file, etc. Third argument `toCPP` defaults to `False`. Should be sent to superclass constructor. If true, C++ files will be generated.
`makeStateMap`	Setup the `self.stateMap` list, where index i represents a set of quantum numbers. Auto-generated LaTeXtables will then include quantum numbers. Ensured correct quantum number - expression pairs sent to simplification functions etc.
`setupOrbitals`	Fill the `self.orbitals` list with SymPy expressions representing the single particle wave functions. Must be functions of the x, y and z SymPy Symbols. Used to calculate the gradients and Laplacians. Element i will represent quantum number i.
`simplifyLocal`	Implement the factorization of the raw expressions. E.g. $kxe^{-x} + ke^{-x} \rightarrow k(x+1)e^{-x}$. Quantum numbers are supplied in the argument in case they are needed, e.g. in the n-dependent exponential factor of the hydrogenic orbitals.
`genericFactor`	Defaults to unity. In case all expressions share a common factor, e.g. $\exp(-\frac{1}{2}\alpha\omega r^2)$ in the case of Quantum Dots, this can be implemented as a generic factor, cleaning up the Latex table. May also dramatically clean up the simplification code.
`extraToFile`	Returns a string which will be set as an introductory text to the Latex output.
`initCPPbasis`	Set up the C++ class names, constructor arguments, member variables. See e.g. the `HO.py` file for examples.
`getCPre`	Given an expression and a its index i as input, set up the C++ pre-return calculation. E.g. `H = printing.ccode(expr/self.genericFactor(i));`
`getCreturn`	Given an expression and a its index i as input, set up the C++ return value. E.g. `return H*(*exp_factor);`
`makeOrbConstArg`	Used to set up the constructor input in the generated `Orbitals` constructor. Defaults to sending names equal to those used in the declaration, however, in e.g. the case of hydrogenic orbitals, different basis functions need different exponential factors. See `hydrogenic.py` for an example.

Implementing these functions will generate a Latex file listing the calculated expressions (see Appendix B and D), the constructor needed by the `Orbitals` subclass holding the generated orbitals and C++ header and source files containing the `BasisFunctions` implementation. In the code used in this thesis, 6000 lines of C++ code was auto-generated using SymPy.

B

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 B.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 B.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 B.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 B.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 B.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 B.6: Orbital expressions HOOrbitals : 1, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_5 \to \phi_{2,0}$	
$\phi(\vec{r})$	$2k^2x^2 - 1$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2x\left(2k^2x^2 - 5\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2y\left(2k^2x^2 - 1\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 6\right)\left(2k^2x^2 - 1\right)$

Table B.7: Orbital expressions HOOrbitals : 2, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_6 \to \phi_{0,3}$	
$\phi(\vec{r})$	$y\left(2k^2y^2 - 3\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2y^2 - 3\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-2k^4y^4 + 9k^2y^2 - 3$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2 - 8\right)\left(2k^2y^2 - 3\right)$

Table B.8: Orbital expressions HOOrbitals : 0, 3. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_7 \to \phi_{1,2}$	
$\phi(\vec{r})$	$x\left(2k^2y^2 - 1\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\left(kx - 1\right)\left(kx + 1\right)\left(2k^2y^2 - 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2y^2 - 5\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2 - 8\right)\left(2k^2y^2 - 1\right)$

Table B.9: Orbital expressions HOOrbitals : 1, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_8 \to \phi_{2,1}$	
$\phi(\vec{r})$	$y\left(2k^2x^2 - 1\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2x^2 - 5\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\left(ky - 1\right)\left(ky + 1\right)\left(2k^2x^2 - 1\right)$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2 - 8\right)\left(2k^2x^2 - 1\right)$

Table B.10: Orbital expressions HOOrbitals : 2, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_9 \to \phi_{3,0}$	
$\phi(\vec{r})$	$x\left(2k^2x^2 - 3\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-2k^4x^4 + 9k^2x^2 - 3$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2x^2 - 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2 - 8\right)\left(2k^2x^2 - 3\right)$

Table B.11: Orbital expressions HOOrbitals : 3, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{10} \rightarrow \phi_{0,4}$	
$\phi(\vec{r})$	$4k^4y^4 - 12k^2y^2 + 3$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2x\left(4k^4y^4 - 12k^2y^2 + 3\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2y\left(4k^4y^4 - 28k^2y^2 + 27\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 10\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$

Table B.12: Orbital expressions HOOrbitals : 0, 4. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{11} \rightarrow \phi_{1,3}$	
$\phi(\vec{r})$	$xy\left(2k^2y^2 - 3\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-y\left(kx - 1\right)\left(kx + 1\right)\left(2k^2y^2 - 3\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-x\left(2k^4y^4 - 9k^2y^2 + 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2xy\left(k^2r^2 - 10\right)\left(2k^2y^2 - 3\right)$

Table B.13: Orbital expressions HOOrbitals : 1, 3. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{12} \rightarrow \phi_{2,2}$	
$\phi(\vec{r})$	$\left(2k^2x^2 - 1\right)\left(2k^2y^2 - 1\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2x\left(2k^2x^2 - 5\right)\left(2k^2y^2 - 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2y\left(2k^2x^2 - 1\right)\left(2k^2y^2 - 5\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 10\right)\left(2k^2x^2 - 1\right)\left(2k^2y^2 - 1\right)$

Table B.14: Orbital expressions HOOrbitals : 2, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{13} \rightarrow \phi_{3,1}$	
$\phi(\vec{r})$	$xy\left(2k^2x^2 - 3\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-y\left(2k^4x^4 - 9k^2x^2 + 3\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-x\left(ky - 1\right)\left(ky + 1\right)\left(2k^2x^2 - 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2xy\left(k^2r^2 - 10\right)\left(2k^2x^2 - 3\right)$

Table B.15: Orbital expressions HOOrbitals : 3, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{14} \rightarrow \phi_{4,0}$	
$\phi(\vec{r})$	$4k^4x^4 - 12k^2x^2 + 3$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2x\left(4k^4x^4 - 28k^2x^2 + 27\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2y\left(4k^4x^4 - 12k^2x^2 + 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 10\right)\left(4k^4x^4 - 12k^2x^2 + 3\right)$

Table B.16: Orbital expressions HOOrbitals : 4, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{15} \rightarrow \phi_{0,5}$	
$\phi(\vec{r})$	$y\left(4k^4y^4 - 20k^2y^2 + 15\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(4k^4y^4 - 20k^2y^2 + 15\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-4k^6y^6 + 40k^4y^4 - 75k^2y^2 + 15$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2 - 12\right)\left(4k^4y^4 - 20k^2y^2 + 15\right)$

Table B.17: Orbital expressions HOOrbitals : 0, 5. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{16} \rightarrow \phi_{1,4}$	
$\phi(\vec{r})$	$x\left(4k^4y^4 - 12k^2y^2 + 3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\left(kx - 1\right)\left(kx + 1\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(4k^4y^4 - 28k^2y^2 + 27\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2 - 12\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$

Table B.18: Orbital expressions HOOrbitals : 1, 4. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{17} \rightarrow \phi_{2,3}$	
$\phi(\vec{r})$	$y\left(2k^2x^2 - 1\right)\left(2k^2y^2 - 3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(2k^2x^2 - 5\right)\left(2k^2y^2 - 3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-\left(2k^2x^2 - 1\right)\left(2k^4y^4 - 9k^2y^2 + 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2 - 12\right)\left(2k^2x^2 - 1\right)\left(2k^2y^2 - 3\right)$

Table B.19: Orbital expressions HOOrbitals : 2, 3. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{18} \rightarrow \phi_{3,2}$	
$\phi(\vec{r})$	$x\left(2k^2x^2 - 3\right)\left(2k^2y^2 - 1\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\left(2k^2y^2 - 1\right)\left(2k^4x^4 - 9k^2x^2 + 3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(2k^2x^2 - 3\right)\left(2k^2y^2 - 5\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2 - 12\right)\left(2k^2x^2 - 3\right)\left(2k^2y^2 - 1\right)$

Table B.20: Orbital expressions HOOrbitals : 3, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{19} \rightarrow \phi_{4,1}$	
$\phi(\vec{r})$	$y\left(4k^4x^4 - 12k^2x^2 + 3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(4k^4x^4 - 28k^2x^2 + 27\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-\left(ky - 1\right)\left(ky + 1\right)\left(4k^4x^4 - 12k^2x^2 + 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2 - 12\right)\left(4k^4x^4 - 12k^2x^2 + 3\right)$

Table B.21: Orbital expressions HOOrbitals : 4, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{20} \rightarrow \phi_{5,0}$	
$\phi(\vec{r})$	$x\left(4k^4x^4 - 20k^2x^2 + 15\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-4k^6x^6 + 40k^4x^4 - 75k^2x^2 + 15$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(4k^4x^4 - 20k^2x^2 + 15\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2 - 12\right)\left(4k^4x^4 - 20k^2x^2 + 15\right)$

Table B.22: Orbital expressions HOOrbitals : 5, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{21} \rightarrow \phi_{0,6}$	
$\phi(\vec{r})$	$8k^6y^6 - 60k^4y^4 + 90k^2y^2 - 15$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2x\left(8k^6y^6 - 60k^4y^4 + 90k^2y^2 - 15\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2y\left(8k^6y^6 - 108k^4y^4 + 330k^2y^2 - 195\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 14\right)\left(8k^6y^6 - 60k^4y^4 + 90k^2y^2 - 15\right)$

Table B.23: Orbital expressions HOOrbitals : 0, 6. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{22} \rightarrow \phi_{1,5}$	
$\phi(\vec{r})$	$xy\left(4k^4y^4 - 20k^2y^2 + 15\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-y\left(kx - 1\right)\left(kx + 1\right)\left(4k^4y^4 - 20k^2y^2 + 15\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-x\left(4k^6y^6 - 40k^4y^4 + 75k^2y^2 - 15\right)$
$\nabla^2\phi(\vec{r})$	$k^2xy\left(k^2r^2 - 14\right)\left(4k^4y^4 - 20k^2y^2 + 15\right)$

Table B.24: Orbital expressions HOOrbitals : 1, 5. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{23} \rightarrow \phi_{2,4}$	
$\phi(\vec{r})$	$\left(2k^2x^2 - 1\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2x\left(2k^2x^2 - 5\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2y\left(2k^2x^2 - 1\right)\left(4k^4y^4 - 28k^2y^2 + 27\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 14\right)\left(2k^2x^2 - 1\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$

Table B.25: Orbital expressions HOOrbitals : 2, 4. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{24} \rightarrow \phi_{3,3}$	
$\phi(\vec{r})$	$xy\left(2k^2x^2 - 3\right)\left(2k^2y^2 - 3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-y\left(2k^2y^2 - 3\right)\left(2k^4x^4 - 9k^2x^2 + 3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-x\left(2k^2x^2 - 3\right)\left(2k^4y^4 - 9k^2y^2 + 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2xy\left(k^2r^2 - 14\right)\left(2k^2x^2 - 3\right)\left(2k^2y^2 - 3\right)$

Table B.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 B.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 B.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 B.29: Orbital expressions HOOrbitals : 6, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

C

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 C.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 C.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 C.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 C.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 C.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 C.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 C.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 C.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 C.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 C.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 C.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 C.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 C.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 C.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 C.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 C.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 C.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 C.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 C.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 C.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 C.21: Orbital expressions HOOrbitals3D : 3, 0, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

D

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 D.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 D.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 D.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 D.4: Orbital expressions hydrogenicOrbitals : 2, 1, 1. Factor $e^{-\frac{1}{2}kr}$ is omitted.

$\phi_4 \to \phi_{2,1,-1}$	
$\phi(\vec{r})$	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 D.5: Orbital expressions hydrogenicOrbitals : 2, 1, -1. Factor $e^{-\frac{1}{2}kr}$ is omitted.

$\phi_5 \to \phi_{3,0,0}$	
$\phi(\vec{r})$	$2k^2r^2 - 18kr + 27$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kx\left(2k^2r^2-30kr+81\right)}{3r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{ky\left(2k^2r^2-30kr+81\right)}{3r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kz\left(2k^2r^2-30kr+81\right)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{k(kr-18)\left(2k^2r^2-18kr+27\right)}{9r}$

Table D.6: Orbital expressions hydrogenicOrbitals : 3, 0, 0. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_6 \to \phi_{3,1,0}$	
$\phi(\vec{r})$	$z\,(kr-6)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kxz(kr-9)}{3r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{kyz(kr-9)}{3r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$\frac{3kr^2-kz^2(kr-9)-18r}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{kz(kr-18)(kr-6)}{9r}$

Table D.7: Orbital expressions hydrogenicOrbitals : 3, 1, 0. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_7 \to \phi_{3,1,1}$	
$\phi(\vec{r})$	$x\,(kr-6)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$\frac{3kr^2-kx^2(kr-9)-18r}{3r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{kxy(kr-9)}{3r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kxz(kr-9)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{kx(kr-18)(kr-6)}{9r}$

Table D.8: Orbital expressions hydrogenicOrbitals : 3, 1, 1. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_8 \to \phi_{3,1,-1}$	
$\phi(\vec{r})$	$y\,(kr-6)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\frac{kxy(kr-9)}{3r}$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$\frac{3kr^2-ky^2(kr-9)-18r}{3r}$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-\frac{kyz(kr-9)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{ky(kr-18)(kr-6)}{9r}$

Table D.9: Orbital expressions hydrogenicOrbitals : 3, 1, -1. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_9 \to \phi_{3,2,0}$	
$\phi(\vec{r})$	$-r^2+3z^2$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\frac{x\left(k\left(-r^2+3z^2\right)+6r\right)}{3r}$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-\frac{y\left(k\left(-r^2+3z^2\right)+6r\right)}{3r}$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-\frac{z\left(k\left(-r^2+3z^2\right)-12r\right)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{k\left(-r^2+3z^2\right)(kr-18)}{9r}$

Table D.10: Orbital expressions hydrogenicOrbitals : 3, 2, 0. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_{10} \to \phi_{3,2,1}$	
$\phi(\vec{r})$	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 D.11: Orbital expressions hydrogenicOrbitals : 3, 2, 1. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_{11} \to \phi_{3,2,-1}$	
$\phi(\vec{r})$	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 D.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 D.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 D.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 D.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 D.16: Orbital expressions hydrogenicOrbitals : 4, 1, 0. Factor $e^{-\frac{1}{4}kr}$ is omitted.

$\phi_{16} \rightarrow \phi_{4,1,1}$	
$\phi(\vec{r})$	$x\left(k^2r^2 - 20kr + 80\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$\frac{4k^2r^3 - 80kr^2 - kx^2(kr-20)(kr-8) + 320r}{4r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{kxy(kr-20)(kr-8)}{4r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kxz(kr-20)(kr-8)}{4r}$
$\nabla^2\phi(\vec{r})$	$\frac{kx(kr-32)\left(k^2r^2-20kr+80\right)}{16r}$

Table D.17: Orbital expressions hydrogenicOrbitals : 4, 1, 1. Factor $e^{-\frac{1}{4}kr}$ is omitted.

$\phi_{17} \rightarrow \phi_{4,1,-1}$	
$\phi(\vec{r})$	$y\left(k^2r^2 - 20kr + 80\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kxy(kr-20)(kr-8)}{4r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$\frac{4k^2r^3 - 80kr^2 - ky^2(kr-20)(kr-8) + 320r}{4r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kyz(kr-20)(kr-8)}{4r}$
$\nabla^2\phi(\vec{r})$	$\frac{ky(kr-32)\left(k^2r^2-20kr+80\right)}{16r}$

Table D.18: Orbital expressions hydrogenicOrbitals : 4, 1, -1. Factor $e^{-\frac{1}{4}kr}$ is omitted.

Bibliography

[1] C. J. Murray, *The SUPERMEN*. New York: Wiley, 1997.

[2] H. P. Langtangen, *Python Scripting for Computational Science*, 3rd ed. Springer, 2008. [Online]. Available: http://www.bibsonomy.org/bibtex/240eb1bb4f4f80d745c3df06a8e882392/hake

[3] M. Lutz, *Programming Python - powerful object-oriented programming (3. ed.)*. O'Reilly, 2006. [Online]. Available: http://www.oreilly.de/catalog/python3/index.html;http://www.bibsonomy.org/bibtex/2b34de9c149a4962403c89fe6360cd3b7/dblp

[4] H. P. Langtangen, *A primer on scientific programming with Python*. Berlin; Heidelberg; New York: Springer, 2011. [Online]. Available: http://www.worldcat.org/search?qt=worldcat_org_all&q=9783642183652

[5] G. O'Regan, *A Brief History of Computing, Second Edition*. Springer, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-1-4471-2359-0

[6] J. Høgberget, *Git Repository: LibBorealis*, 2013. [Online]. Available: http://www.github.com/jorgehog/QMC2

[7] D. Griffiths, *Introduction to Quantum Mechanics*, 2nd ed. Pearson, 2005.

[8] B. Hammond, J. W. A. Lester, and P. J. Reynolds, *Monte Carlo Methods in Ab Initio Quantum Chemistry*, S. Lin, Ed. World Scientific Publishing Co., 1994.

[9] C. W. Gardiner, *Handbook of stochastic methods for physics, chemistry, and the natural sciences*, 3rd ed. Berlin: Springer-Verlag, 2004. [Online]. Available: http://www.loc.gov/catdir/enhancements/fy0818/2004043676-d.html

[10] H. Risken and H. Haken, *The Fokker-Planck Equation: Methods of Solution and Applications Second Edition*. Springer, 1989.

[11] 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.

[12] M. Hjorth-Jensen, "Computational Physics," 2010.

[13] J. J. Sakurai, *Modern Quantum Mechanics*, Revised ed ed. New York: Addison-Wesley, 1994.

[14] I. Shavitt and R. J. Bartlett, *Many-Body Methods in Chemistry and Physics*. Cambridge: Cambridge University Press, 2009.

[15] L. E. Lervåg, "VMC CALCULATIONS OF TWO-DIMENSIONAL QUANTUM DOTS," Master's thesis, University of Oslo, 2010.

[16] D. A. Nissenbaum, "The stochastic gradient approximation: an application to li nanoclusters," Ph.D. dissertation, Northeastern University, 2008. [Online]. Available: http://hdl.handle.net/2047/d10016466

[17] C. Filippi and C. J. Umrigar, "Multiconfiguration wave function for quantum Monte Carlo calculations of first-row diatomic molecules," *The Journal of Chemical Physics*, vol. 105, Jul. 1996.

[18] G. Golub and C. Van Loan, *Matrix computations*. Johns Hopkins Univ Press, 1996, vol. 3.

[19] A. Harju, B. Barbiellini, S. Siljamäki, R. M. Nieminen, and G. Ortiz, "Stochastic Gradient Approximation: An Efficient Method to Optimize Many-Body Wave Functions," *Physical Review Letters*, vol. 79, pp. 1173–1177, Aug. 1997.

[20] S. Klein, J. P. W. Pluim, M. Staring, and M. A. Viergever, "Adaptive Stochastic Gradient Descent Optimisation for Image Registration." *International Journal of Computer Vision*, vol. 81, no. 3, pp. 227–239, 2009. [Online]. Available: http://dx.doi.org/10.1007/s11263-008-0168-y

[21] J. M. Leinaas, "Non-Relativistic Quantum Mechanics," lecture notes FYS4110.

[22] H. Flyvbjerg and H. G. Petersen, "Error estimates on averages of correlated data," *The Journal of Chemical Physics*, vol. 91, no. 1, pp. 461–466, 1989. [Online]. Available: http://link.aip.org/link/?JCP/91/461/1

[23] I. Shavitt and R. J. Bartlett, *Many-body methods in chemistry and physics: MBPT and coupled-cluster theory*, ser. Cambridge molecular science series. Cambridge University Press, 2009. [Online]. Available: http://books.google.no/books?id=gU1eHAAACAAJ

[24] D. C. Lay, *Linear Algebra and its Applications*, 4th ed. Pearson, 2012.