Quantum-mechanical systems in traps and density functional theory

by

Jrgen Hgberget

THESIS
for the degree of
MASTER OF SCIENCE

(Master in Computational Physics)

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

June 2013

Preface

blah blah

Contents

1

Introduction

blah blah

Part I

Theory

2

Scientific Programming

The introduction of the computer around World War II had a major impact on the mathematical fields of science. Previously unsolvable problems were now solvable. The question was no longer whether or not it was possible, but rather to what precision and with which method. The computer spawned a new branch of physics, *computational physics*, breaching barriers no one could even imagine existed. 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].

2.1 Programming Languages

Writing a program, or a code, is a list of instructions for the computer. It is in many ways similar to writing human-to-human instructions. You may use different programming languages, such as C++, Python, Java, as long as the reader is able to translate it. The translator, called *compiler* or *interpreter*, translates your program from e.g. C++ code to machine code. Other languages such as Python are interpreted real-time and therefore require no compilation; it instructs as it reads. Although the latter seems like a better solution, it comes at the price of efficiency, a key concept in programming.

As a rule of thumb, efficiency is inverse proportional to the complexity of the programming language. It is therefore natural to sort languages into different subgroups depending on where they are at the efficiency-complexity scale.

2.1.1 High-level Languages

Scientific programming is more than number crunching loops. This section's subgroup of languages are often referred to as *scripting languages*. A script is a short code 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 [2].

For these types of jobs, the relief of simple rigorous syntax weighs up for the efficiency penalty. In most cases, the runtime of the program is so small that efficiency becomes irrelevant, leaving scripting languages the optimal tool for the task. These languages which prefer simplicity over efficiency are referred to as *High-level* [1]. Examples of high-level languages are Python, Ruby, Perl, Visual Basic and UNIX shells. In

[1] There are different definitions of high-level vs. low-level. You have languages such as *assembly*, which is extremely complex and close to machine code, leaving all machine-independent languages as high-level ones. However, for the purpose

this thesis, Python is the mainly used scripting language.

Python

Python is a programming language with a focus on being simple to learn and have a very clean syntax [3, 2]. To mention a few of the entries in the *Zen of Python*[2], "Beautiful is better than ugly. Simple is better than complex. Readability counts. If the implementation is hard to explain, it's a bad idea."

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 #Sum100Python.py
2 print sum(range(101))
```

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

2.1.2 Low-level Languages

A huge part of 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 $A\vec{x} = B$, 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 that we 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 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. If you e.g. try to access an element outside the bounds of an array, Python would tell you a detailed error message with proper traceback, whereas the compiled C++ code would crash runtime leaving nothing but a "segmentation fault" for the user. However, when the optimized program ends up running for days, the extra time spent in development pays off. In addition you have several options to optimize your compiled machine code by having the compiler rearrange the way instructions are sent to the processor[3] (without ruining it of course), which interpreted languages does not have.

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. [4]. The following code is a C++ implementation of the sum in Eq. 2.1.1:

---

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

[2]Retrieved by typing "import this" in your Python shell.

[3]I will not go more into details on this topic. For more information research topics such as *CPU cache*, *Memory bus latency* and *CPU architecture* in general.

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

int main(){

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

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

        return 0;
}
```

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

As we can see in lines five and six, we need to declare S and i as integer variables, exactly as described in section 2.1.2. In comparison with the Python version, it is clear that lower level languages are more complicated, and not designed for simple jobs as calculating a single sum.

Even though this is an extremely simply example, it illustrates the difference in coding styles between high- and low-level languages: Complexity vs. simplicity, efficiency vs. readability. I will not go through all the basic details of C++, but rather focus on the more complicated parts involving object orientation in scientific programming.

2.2 Object Orientation

Object orientated programming was introduced in the language *Simula 67*, developed by the Norwegian scientists Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Research Center [4]. It quickly became the state-of-the-art in programming, and is today used throughout the world in all branches of programming. It is brilliant in the way that it ties our everyday intuition into the programming language - our brain is object oriented. It is focused around the concept of *classes*, a collection of variables and functions aimed for a specific task. In a program, instances of classes, or *objects*, are created and can be viewed as independent actors aimed for specific tasks [3, 2, 4]. However, they provide a great deal of functionality like e.g. *inheritance* and accessibility control.

2.2.1 Inheritance

Consider the abstract idea of a keyboard. All keyboards have two things in common: A board and keys (obviously). In object orientation we would say that the *superclass* of keyboards describe a board with keys. It is *abstract* in the sense that you do not need to know what the keys look like, or what function they possess, in order to define the concept of a keyboard.

However, we can have different types of keyboards, for example a computer keyboard, or a musical keyboard. They are different in design and function, but they both relate to the same concept of a keyboard described previously. They are both *subclasses* of the same superclass, inheriting the basic concepts, but expands upon them defining their own specific case.

I will present examples assuming the reader is somewhat familiar to programming concepts and basic Python. On the following page an example implementation of a Keyboard class in Python is listed.

```python
#KeyboardClassPython.py

from math import sin
from numpy import linspace, zeros

class Keyboard():

    #Set member variables board and keys
    #A subclass will override these with their own representation
    keys = None
    numberOfKeys = 0
    board = None

    #Constructor sets the number of keys
    def __init__(self, nKeys):
        self.nKeys = nKeys

    def setupKeys(self):
        raise NotImplementedError("This function is pure virtual. Override me!")

    def pressKey(self, key):
        raise NotImplementedError("This function is pure virtual. Override me!")


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

    def __init__(self, language, nKeys):

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

        self.language = language
        self.setupKeys()

    def setupKeys(self):
        if self.language == "Norwegian":
            "Set up norwegian keyboard style"


        elif self.languange == "English":
            "Set up the english one"

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


class MusicalKeyboard(Keyboard):

    def __init__(self, nKeys, noteLength, amplitude):
        super(ComputerKeyboard, self).__init__(nKeys)

        self.amplitude, self.noteLength = amplitude, noteLength

        self.keys = zeros(nKeys)
        self.setupKeys()


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

    def pressKey(self, key):

        t = linspace(0, self.noteLength, 100)
        pi = 3.141592

        #returns harmonic wave with frequency and amplitude
        #representing key pressed and volume level.
        soundOutput = self.amplitude*sin(2*pi*self.keys[key]*t)
        return soundOutput
```

As we can see, the only thing differentiating the two keyboard types are how the keys are set up, and what happens when we press one of them. A superclass function designed to be overridden is referred to as *virtual*. If the function is not even implemented, they are referred to as *pure virtual* in the sense that they should be overwritten by any subclass. More on this in the next section.

2.2.2 Pointers, Virtual Functions and Types

A pointer is a hexadecimal number representing a memory address where some *type* of object is stored, e.g. a `int` at `0x7fff0882306c`. Higher level languages like Python handles all the pointers and typesetting by themselves. In low-level languages like C++, however, you need to control everything. If you pass a pointer to an object, e.g. `Keyboard* myKeyboard`, as an argument to a function, whenever that function makes changes to the object, the object is changed globally, since the memory address is directly accessed. If you instead choose to send the object without a pointer declaration, e.g. `Keyboard myKeyboard`, changing the value will not change the object globally. What happens instead is that you change a local copy of the object. However, once you crack the code, pointers are dear friends, not lethal enemies.

Virtual functions are functions designed for overriding; `virtual` is a flag telling the compiler to search for the deepest implementation of the specific function (in terms of subclassing) no matter the original type. `setupKeys` and `pressKey` are examples of this, however, they are in a sense more than virtual, since they are not even implemented, namely pure virtual; they have to be overridden in order to work.

In python, we never come into trouble with virtual functions, since you don't manually control the object type. In C++ however, we have to specify whether or not a function is virtual in the declaration in order to achieve the desired functionality. This because an instance of `ComputerKeyboard` could either be of type `ComputerKeyboard` or `Keyboard`. The next example will illustrate this.

```
#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
```

Typecasting and Polymorphism

In order to understand these concepts, consider the previous example. In the first call, the object is declared as a `superClass*` type, however, it is still initialized to be a `subClass` pointer, which results in the subclass' functions overriding the corresponding ones of the superclass, given they are virtual. In programming terms we say that *any subclass is type-compatible with a pointer to it's superclass.*

In the second call, the same thing happens, even though it is set as a `subclass*` type. This is because the function is instructed to receive a `superClass*` input. If it receives anything else, it simply attempts to convert it, or *cast* it, to the specified type; *typecasting*[4]. As discussed in the previous paragraph casting from a subclass to its superclass is allowed. The third call, outside the function, demonstrates that if we do not typecast the object, the object's functions consists solely of its own, virtual or not.

This all boils down to one powerful concept in object orientated programming, namely *polymorphism.* Polymorphism is the scenario where e.g. a function is declared to receive a superclass pointer (like `testFunc`), yet when it's called, it's called with subclass implementations of the superclass (the second call described above). The function is instructed to call member functions of the subclass, however, we can override these by declaring them as virtual functions. In other words, the code can be written extremely organized and versatile given proper use of polymorphism. To further illustrate this, consider the following example from the Quantum Monte Carlo code developed in this thesis, more spesificly

```
1  class Potential {
2  protected:
3      int n_p;
4      int dim;
5
6  public:
7      Potential(int n_p, int dim);
8      Potential();
9
10     virtual double get_pot_E(const Walker* walker) const = 0;
11
12 };
13
14 class Coulomb : public Potential {
15 public:
16
17     Coulomb(GeneralParams &);
18
19     virtual double get_pot_E(const Walker* walker) const;
```

[4]The standard example of typecasting is converting a double to an integer, resulting in the stripping of all the decimal bits (flooring).

```
20
21 };
22
23 class Harmonic_osc : public Potential {
24 protected:
25     double w;
26
27 public:
28
29     Harmonic_osc(GeneralParams &);
30
31     double get_pot_E(const Walker* walker) const;
32
33 };
```

```
1  double Coulomb::get_pot_E(const Walker* walker) const {
2
3      double e_coulomb = 0;
4
5      for (int i = 0; i < n_p - 1; i++) {
6          for (int j = i + 1; j < n_p; j++) {
7              e_coulomb += 1 / walker->r_rel(i,j);
8          }
9      }
10
11     return e_coulomb;
12 }
13
14 double Harmonic_osc::get_pot_E(const Walker* walker) const {
15
16     double e_potential = 0;
17
18     for (int i = 0; i < n_p; i++) {
19         e_potential += 0.5 * w * w * walker->get_r_i2(i);
20     }
21
22     return e_potential;
23 }
```

With this subclass hierarchy of potentials, we can instruct the system to access objects of type `Potential*`, and simply call the objects function `get_pot_E` with the current walker as input. Or, even more powerfully, we can assign any number of `Potential*` objects to the system, and simply iterate through them and accumulate their energy contributions:

```
1  class System {
2  protected:
3      ...
4
5      std::vector<Potential*> potentials;
6
7      ...
8  };
9
10 double System::get_potential_energy(const Walker* walker) {
11     double potE = 0;
12
13     //Iterate through the potential objects in the potentials list. (*pot) extracts the
           pointer from the iterator.
14     for (std::vector<Potential*>::iterator pot = potentials.begin(); pot != potentials.
           end(); ++pot) {
15         potE += (*pot)->get_pot_E(walker);
16     }
17
18     return potE;
19 }
```

The objects in the list can be any subclass implementation of the `Potential` class, all the compiler needs

to know is that it has a method `get_pot_E` that it can call on runtime. Polymorphism creates a port for versatile code structures. It does not matter whether seven different potentials are loaded or none at all. Versatile, generalized, yet beautiful.

2.2.3 Const Correctness

In the `Potential` code example above, function declarations with `const` are used. If an object is declared with `const` on input, e.g. `void f(const x)`, the function itself cannot alter the value of `x`. It is a safeguard that nothing will happen to `x` as it passes through `f`. This is practical in situations where major bugs will arise if anything happens to an object, yet you do not want to copy it on input.

If you declare a member function itself with `const` on the right hand side, it safeguards the function from changing any of the class variables. If you e.g. have a variable representing the electron charge, you do not want this changed by the Coulomb class member function. This should only happen through specific functions whose sole purpose is changing the charge, and taking care of any following consequences.

In other words: `const` works as a safeguard for changing values which should remain unchanged. A change in such a variable is then followed by a compiler error instead of infecting your code with bugs, resulting in unforeseen consequences.

2.2.4 Accessibility levels and Friend classes

`const` is a direct way to avoid any change what so ever. However, sometimes we want to keep the ability to alter variables, but only in certain situations, as e.g. internally in the class. As an example, from the main file, you should not have access to `QMC` member functions such as `dump_output`, since it does not make sense, or is directly dangerous, to do out of a context. However, you obviously want access to the `run_method` function.

The solution to this problem is to set accessibility levels. Declaring a variable under the `public` part of a class sets its accessibility level to *public*, meaning that anything, anywhere can access it as long as it has access to the object. Declarations beneath the `private` part stops all other classes than instances of itself from reaching it, even subclass instances. If you want private variables inherited, the `protected` accessibility level should be used, this ensures that the members are hidden for everyone except the class itself and its subclasses.

There is one exception to the rule of protected and private variables, namely *friend* classes. In the QMC code, there is a output class called `OutputHandler`. This class needs access to protected variables, since the user should be able to output anything he wants. If we `friend` the output class with `QMC`, we get exactly this behavior:

```
1
2  class QMC {
3  protected:
4
5      ...
6
7      std::stringstream s;
8
9      ...
10
11     int n_c;
12
13     ...
14
15     int cycle;
16
```

```
    ...

public:

    ...

};

class VMC : public QMC {
protected:

    ...

    Walker* original_walker;
    Walker* trial_walker;

    ...

public:

    ...

    friend class Distribution;
    ...

};
```

```
void Distribution::dump() {

    //if we are more than half way we save a snapshot of the walker every 100'th step
    if ((vmc->cycle >= vmc->n_c / 2) && (vmc->cycle % 100 == 0)) {

        //s: unique file name for the current snapshot of the diffusing walker
        s << path << "walker_positions/" << filename << node <<"_"<< i <<".arma";

        //saves the position matrix to file
        vmc->original_walker->r.save(s.str());

        //clears the stringstream and iterates identifyer 'i'
        s.str(std::string());
        i++;
    }
}
```

Without going into details, we can see that `Distribution` has full access to protected, or hidden, members `VMC`. Friend classes are saviors in those very specific cases when you really need full access to protected members of another class, but setting full public access would ruin the code. It is true that you could code your entire code without `const` and with solely public members, but in that case, it is very easy to put together a very disorganized code, with pointers pointing everywhere and functions being called in all sorts of contexts. Clever use of accessibility levels will make your code easier to develop in an organized, intuitive way - you will be forced to implement things in an organized fashion.

2.2.5 Example: PotionGame

To end the section I would like to demonstrate the versatile power of object orientation with polymorphism by introducing a simple turn based game. Consider the following codes:

```
#potionClass.py

class Potion:

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

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

```
1 #playerClass.py
2
3 class Player:
4
5     #Initialize the player at full health with no potions
6     def __init__(self, name):
7         self.health = 100
8         self.energy = 100
9         self.name = name
10
11        self.dead = False
12
13        self.potions = []
14
15    def addPotion(self, potion):
16        self.potions.append(potion)
17
18    #Selects the given potion and consumes it. The potion needs to know
19    #the player it should affect, hence we send 'self' as an argument.
20    def usePotion(self, potionIndex):
21
22        print "%s consumes %s." % (self.name, self.potions[potionIndex].name)
23
24        self.potions[potionIndex].applyPotion(self)
25        self.potions.pop(potionIndex)
26
27
28
29
30    def changeHealth(self, amount):
31        self.health += amount
32
33        #Cap health at [0,100].
34        if self.health > 100:
35            self.health = 100
36        elif self.health <= 0:
37            self.health = 0
38            self.dead = True;
39
40
41    def changeEnergy(self, amount):
42        self.energy += amount
```

```
43
44          #Cap energy at [0,100].
45          if self.energy > 100:
46              self.energy = 100
47          elif self.energy < 0:
48              self.energy = 0
49
50      #lists the potions to the user
51      def displayPotions(self):
52
53          if len(self.potions) == 0:
54              print "No potions aviable"
55
56          for potion in self.potions:
57              print potion.name
58
59      def attack(self, player):
60
61          energyCost = 50
62          damage = 40
63
64          player.changeHealth(-damage)
65          self.changeEnergy(-energyCost)
66
67          print "\n%s hit %s for %s using %s energy" % (self.name, player.name,
68                                                        damage, energyCost)
```

We have a `Player` class keeping track of a players energy and health level, and which potions the player is carrying, initiates attacks etc. The `Potion` class described all potions, that is, an object of type `Potion` with the ability to affect a player in some way. The subclasses define specifically which effect is to be applied, e.g. `HealthPotion` changes the health level of the player by a certain amount. User output is automated within the class members. This code does nothing by itself, but let us use it in an example where two players fight each other:

```
1  #potionGameMain.py
2
3  from potionClass import *
4  from playerClass import *
5
6  def roundOutput(players, n):
7          header= "\nRound %d: " % n
8          print header.replace('0','start')
9          for player in players:
10                 print "%s (hp/e=%d/%d):" % (player.name, player.health, player.energy)
11                 player.displayPotions()
12                 print
13
14
15 player1 = Player('john');
16 player1.addPotion(HealthPotion(10)); player1.addPotion(EnergyPotion(30))
17
18 player2 = Player('james')
19 player2.addPotion(EnergyPotion(20)); player2.addPotion(EnergyPotion(20))
20
21 #Initial output
22 roundOutput([player1, player2], 0)
23
24 #Round one: Each player gets an attack after which both can consume potions
25 player1.attack(player2)
26 player1.usePotion(1); player2.usePotion(0)
27
28 player2.attack(player1)
29 player1.usePotion(0); player2.usePotion(0)
30
31 roundOutput([player1, player2], 1)
32 #Round one end.
33 #...
```

```
~$ python potionGameMain.py

Round start:
john (hp/e=100/100):
Health Potion (10)
Energy Potion (30)

james (hp/e=100/100):
Energy Potion (20)
Energy Potion (20)


john hit james for 40 using 50 energy
john consumes Energy Potion (30).
james consumes Energy Potion (20).

james hit john for 40 using 50 energy
john consumes Health Potion (10).
james consumes Energy Potion (20).

Round 1:
john (hp/e=70/80):
No potions aviable

james (hp/e=60/70):
No potions aviable
```

The readability of this code is pretty good. Imagine if we had no objects, but just a lot of parameters per player juggled around in variables such as `Player1health` etc. Increasing the number of players then requires a total rewriting of the entire program, where as in this object oriented style, it is just a matter of adding another player object. Object orientation is truly brilliant when it comes to developing codes.

In this section I have not focused too much on scientific computing, but rather on the use of object orientation in general. When the physical methods are discussed in section **Insert physics section**, I will get back to a more specific description of scientific programming.

As an introduction to the next section, take a look at the way the classes and the main file are separated in this section's example. In a small code like this there's really no point of doing so, but when the class structures span thousands of lines, having a good structure and the right editor is crucial to the development process and the code's readability.

2.3 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. (2.1.1), 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. integration, code structuring becomes very important. For details about the structuring of the code used in this thesis, see Section 4.1.

The case of using object orientation with focus on code structure is covered in Section 2.2.

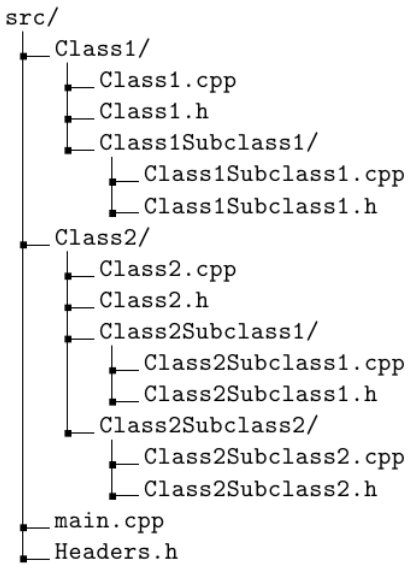

Figure 2.1: An illustration of a standard way to organize source code. The file endings represent C++ code.

2.3.1 File structures

Developing codes in scientific scenarios often involves enormous frameworks. For instance, when doing Molecular Dynamics, several collision models, force models etc. is perhaps 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 subclasses etc. gets large.

When codes consists of several independent class structures (sampling rules, potentials, etc.), it is common to gather the implementations of the different classes in separate files (see Section 2.2.5 for an example). The alternative, as mentioned, is a single file consisting of thousands of lines; navigating through it is a mess. This would be purely a cosmetic issue if the process of writing the code was linear, however, empirical evidence suggests otherwise; at least half the time is spent debugging functions, going back and forth through files. If you are using tools such as Makefile, you will also avoid recompiling unchanged parts of the code.

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 2.1 shows an example setup for the framework of an object oriented code.

2.3.2 Consistent Code Styles

KEEP TO ONE STANDARD!

2.3.3 Version Control

USE E.G. GIT!

2.3.4 Using GUI Tools

stand-alone openfile dialoges are awesome!

3

Quantum Monte-Carlo

Quantum Monte-Carlo is the method of solving Schrödinger's equation using statistical simulations, i.e. Monte-Carlo simulations. The statistical nature of Quantum Mechanics makes Monte-Carlo methods the perfect tool for accurate simulations. As we will see, the Schrödinger equation resembles a *diffusion equation* in complex time. It is therefore from a mix of diffusion theory and Quantum Mechanics that the Quantum Monte-Carlo methods originate.

In this chapter *Dirac Notation* will be used. See Appendix A for an introduction.

3.1 Modeling Diffusion

Like any phenomena involving a density function, or distribution, Quantum Mechanics can be modeled by diffusion. In Quantum Mechanics, the distribution is given by $|\psi(\vec{r},t)|^2$, 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 represents a position is space (and time for time-dependent studies). Averaging values over the paths of the ensemble will yield average values corresponding to the probability distribution governing the movement of individual walkers.

Such random movement is 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 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.

3.1.1 Stating the Schrödinger Equation as a Diffusion Problem

Consider the time-dependent Schrödinger Equation for an abstract many-body wave function[1] $\Phi(x,t)$ with an arbitrary energy shift E'

[1] See Section 3.5.1 for details about many-body wave functions.

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

The formal solution given a time-independent Hamiltonian is given by separation of variables in $\Phi(x,t)$ (see ref. [5] for more details regarding assumptions etc.). This corresponds to using the standard time-evolution operator. Further expanding $\Phi(x, t = t_0)$ in the eigenstates of $\widehat{\mathbf{H}}$, $\Psi_k(x)$, yields the following solution

$$\begin{aligned}
\Phi(x,t) &= e^{-i(\widehat{\mathbf{H}}-E')(t-t_0)}\Phi(x,t=t_0) && (3.2)\\
&= \sum_{k=0}^{\infty} C_k\Psi_k(x)e^{-i(E_k-E')(t-t_0)} && (3.3)\\
C_k &= \langle\Psi_k(x)|\Phi(x,t=0)\rangle
\end{aligned}$$

The real form of this equation is obtained through a Wick rotation in time ($it \to \tau$), which basically means that the time-evolution operation is substituted by a *projection operation*. To illustrate this, look at the solution of the real equation with E' chosen to be the ground state energy of $\widehat{\mathbf{H}}$, and t_0 chosen to be zero

$$\begin{aligned}
\Phi(x,\tau) &= \sum_{k=0}^{\infty} C_k\Psi_k(x)e^{-(E_k-E_0)\tau} && (3.4)\\
&= C_0\Psi_0(x) + \sum_{k=1}^{\infty} C_k\Psi_k(x)e^{-\delta E_k\tau}, && (3.5)
\end{aligned}$$

where $\delta E_k = E_k - E_0 > 0$ for $k \geq 1$. Observe that as τ increase, the excited states of $\widehat{\mathbf{H}}$ will be exponentially dampened, hence the name projection operation. Note however, that no real-time solutions can be achieved by solving this equation; only in the limit $\tau \to \infty$ will the solution match that of the Schrödinger Equation.

The approach of Quantum Monte Carlo is to split the evolution of τ into small sequential steps $\delta\tau$, and use the latest calculated energy, the *trial energy*, E_T, as an approximation to the ground state energy. The error introduced by this approximation will be discussed in Section 3.8. Either way, restating Eq. (3.4) in terms of sequential steps yields

$$\begin{aligned}
\Phi(x,n\delta\tau) &= \left[\prod_{i=1}^{n} e^{-(\widehat{\mathbf{H}}-E_T)\delta\tau}\right]\Phi(x,t=0) && (3.6)\\
&= \sum_{k=0}^{\infty} C_k\Psi_k(x)e^{-(E_k-E_T)n\delta\tau} && (3.7)
\end{aligned}$$

which remains exact for an infinite amount of infinitely small time steps. In practice, a balance is sought between the *number of cycles*, n, and the time step. This restatement might seem redundant, however, it is the key which opens up the possibility to model the solution by discretized diffusion of particles in Quantum Mechanical potentials.

In order to simulate the effect of the exponential operator in Eq. (3.6) on our initial state, consider the following rewriting

$$\widehat{\mathbf{P}}(n,\delta\tau) \equiv \left[\prod_{i=1}^{n} e^{-(\widehat{\mathbf{H}}-E_T)\delta\tau}\right] \tag{3.8}$$

$$= e^{-(\widehat{\mathbf{H}}-E_T)\delta\tau} e^{-(\widehat{\mathbf{H}}-E_T)\delta\tau} \cdots e^{-(\widehat{\mathbf{H}}-E_T)\delta\tau} \tag{3.9}$$

$$= \int_{x_1}\int_{x_2}\cdots\int_{x_n} |x_n\rangle \langle x_n| e^{-(\widehat{\mathbf{H}}-E_T)\delta\tau} |x_{n-1}\rangle \langle x_{n-1}| e^{-(\widehat{\mathbf{H}}-E_T)\delta\tau} |x_{n-2}\rangle$$
$$\times \cdots \langle x_2| e^{-(\widehat{\mathbf{H}}-E_T)\delta\tau} |x_1\rangle \langle x_1| \mathrm{d}x_1 \cdots \mathrm{d}x_n \tag{3.10}$$

where complete sets of position basis states are introduced (see appendix A). The state from Eq. (3.6) are then given by

$$\Phi(x, n\delta\tau) = \langle x| \widehat{\mathbf{P}}(n,\delta\tau) |\Phi(t=0)\rangle \tag{3.11}$$

$$= \int_{x_1}\int_{x_2}\cdots\int_{x_{n-1}} \langle x| e^{-(\widehat{\mathbf{H}}-E_T)\delta\tau} |x_{n-1}\rangle \langle x_{n-1}| e^{-(\widehat{\mathbf{H}}-E_T)\delta\tau} |x_{n-2}\rangle$$
$$\times \cdots \langle x_2| e^{-(\widehat{\mathbf{H}}-E_T)\delta\tau} |x_1\rangle \Phi(x_1, t=0)\mathrm{d}x_1 \cdots \mathrm{d}x_{n-1} \tag{3.12}$$

The final state x_n are replaced by x due to the fact that $\langle x|x_n\rangle = \delta(x = x_n)$, which makes the x_n integral vanish. The final state is achieved by summing through all possible *paths* represented by integrals over the position variables x_i. This is the path integral formalism of Quantum Mechanics. More information about this formalism can be found in ref. [6]. Integrating out a degree of freedom, that is, *evolving* from x_i to x_{i+1} is given by

$$\Phi(x_{i+1}, i\delta\tau + \delta\tau) = \int_{x_i} \langle x_{i+1}| e^{-(\widehat{\mathbf{H}}-E_T)\delta\tau} |x_i\rangle \Phi(x_i, i\delta\tau)\mathrm{d}x_i \tag{3.13}$$

$$\equiv \int G(x_{i+1}, x_i; \delta\tau)\Phi(x_i, i\delta\tau)\mathrm{d}x_i \tag{3.14}$$

where the *Green's Function*, $G(x_{i+1}, x_i; \delta\tau)$ serves as the transition probability.

This is still not a practical way of solving the equations. By finding the Green's function we might as well find the solution directly. However, if we split the Hamiltonian into the kinetic, $\widehat{\mathbf{T}}$ and the potential part, $\widehat{\mathbf{V}}$, we can split the Green's function into two well known processes. This splitting is known as the *short time approximation*.

$$\langle x| e^{-(\widehat{\mathbf{H}}-E_0)\delta\tau} |y\rangle = \langle x| e^{-(\widehat{\mathbf{T}}+\widehat{\mathbf{V}}-E_T)\delta\tau} |y\rangle$$
$$= \langle x| e^{-\widehat{\mathbf{T}}\delta\tau} e^{-(\widehat{\mathbf{V}}-E_T)\delta\tau} |y\rangle + \frac{1}{2}[\widehat{\mathbf{V}}, \widehat{\mathbf{T}}]\delta\tau^2 + \mathcal{O}(\delta\tau^3)$$
$$\simeq \langle x| e^{-\widehat{\mathbf{T}}\delta\tau} e^{-(\widehat{\mathbf{V}}-E_T)\delta\tau} |y\rangle \tag{3.15}$$

, that is, we split the Green's Function into two processes which are well known, respectively diffusion and branching

$$G_{\mathrm{Diff}} = e^{-\widehat{\mathbf{T}}\delta\tau} \tag{3.16}$$

$$G_{\mathrm{B}} = e^{-(\widehat{\mathbf{V}}-E_T)\delta\tau} \tag{3.17}$$

Once a diffusion model is selected, for each cycle, the Green's functions are used to propagate the ensemble of walkers making up our distribution 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 we simulate long enough. These constraints will be covered in more detail later.

Incorporating only the effect of Eq.(3.16) results in a method called *Variational Monte Carlo*. Including the branching term as well results in *Diffusion Monte Carlo*. These methods will be discussed in sections 3.7 and 3.8. In either of these methods, diffusion is a key process. In practice, we choose a diffusion model where closed form expressions for the Green's Functions exists. Two examples of this will be presented in the following section.

3.1.2 Solving the Diffusion Problem

Different diffusion models may be applied to the problem introduced in the previous section. All models should in theory be able to produce the same result, however, systematic errors of specific implementations may vary depending on the choice of diffusion model. Two of the most common choices in Quantum Monte-Carlo is *Isotropic diffusion* and *Fokker-Planck Anisotropic Diffusion*.

3.1.3 Isotropic Diffusion

Isotropic diffusion is a process in which diffusing particles sees all possible directions as an equally probable path. Eq. (3.18) is an example of this. This is the simplest form of a diffusion equation, the case with a linear *diffusion constant*, D, and no drift terms.

$$\frac{\partial P(\vec{r},t)}{\partial t} = D\nabla^2 P(\vec{r},t) \tag{3.18}$$

In the Quantum Monte Carlo case, the value of the diffusion constant is $D = \frac{1}{2}$, that is, the term scaling the Laplacian in the Schrödinger Equation. The closed form expression for the Green's function is a Gaussian distribution with variance $2D\delta t$ [7]

$$G^{\text{ISO}}_{\text{Diff}}(i \to j) \propto e^{-(x_i - x_j)^2/4D\delta\tau}. \tag{3.19}$$

These equations describe the diffusion process theoretically, however, in order to achieve specific sampling rules for our walkers, we need a connection between the time-dependence of the total distribution and the time-dependence of an individual walker's components in configuration space. 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 the system. For the simple isotropic diffusion described previously, solving the Langevin equation using a Forward Euler approximation for the time derivative results in the following relation:

$$x_{i+1} = x_i + \xi, \qquad \begin{aligned} \text{Var}(\xi) &= 2D\delta t, \\ \langle \xi \rangle &= x_i, \end{aligned} \tag{3.20}$$

where ξ is a normal distributed number whose variance match that of the Green's function in Eq. (3.19). This relation is in agreement with the isotropy of Eq. (3.18) in the sense that the displacement is symmetric around the current position.

3.1.4 Anisotropic Diffusion: Fokker-Planck

Anisotropic diffusion, in contrast to isotropic diffusion, does not count all directions as equally probable. An example of this is diffusion according to the *Fokker-Planck Equation*, that is, diffusion with a drift term, $\vec{F}(\vec{r},t)$, responsible for pushing the walkers in the direction of configurations with higher probabilities.

$$\frac{\partial P(\vec{r},t)}{\partial t} = D\nabla\cdot\Big[\Big(\nabla - \vec{F}(\vec{r},t)\Big)P(\vec{r},t)\Big] \tag{3.21}$$

The remarkable thing is that simple isotropic diffusion processes obey this relation [7]. This means that Quantum Mechanical distributions can be modeled by the Fokker-Planck Equation, leading to a more optimized way of sampling in practical situations.

In Quantum Monte Carlo we want convergence to a stationary state. We can use this criteria to deduce expression for the drift term given our Quantum Mechanical distribution. A stationary state is obtained when the left hand side of Eq. (3.21) is zero:

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

The next thing we want to achieve is cancellation in the rest of the terms. In order to obtain a Laplacian term on the right hand side to potentially cancel out the one on the left, the drift term needs to be on the form $F(\vec{r},t) = g(\vec{r},t)\nabla P(\vec{r},t)$. Inserting this yields

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

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

$$\begin{aligned}\vec{F}(\vec{r},t) &= \frac{1}{P(\vec{r},t)}\nabla P(\vec{r},t)\\ &= \frac{2}{|\psi(\vec{r},t)|}\nabla|\psi(\vec{r},t)|\end{aligned} \tag{3.22}$$

In Quantum Monte Carlo, the drift term is called *The Quantum Force*, since 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 3.3. As expected, it is no longer symmetric

$$G^{\mathrm{FP}}_{\mathrm{Diff}}(i\ \to j) \propto e^{-(x_i - x_j - D\delta\tau F(x_i))^2/4D\delta\tau}. \tag{3.23}$$

The Langevin Equation for the Fokker-Planck Equation

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

$$\frac{\partial x_i}{\partial t} = DF(x_i) + \eta, \tag{3.24}$$

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

$$x_{i+1} = x_i + \xi + DF(x_i)\delta t, \tag{3.25}$$

where ξ is the same as for the isotropic case. We observe that if the drift term is set to zero, we are back in the isotropic case, just as required. For more details regarding the Fokker-Planck Equation and the Langevin equation, see ref. [8], [9] and [10].

Notes on importance sampling

To this point, it might seem far-fetched that switching diffusion model to a Fokker-Planck distribution does not violate the original equation, i.e. the Schrödinger equation. Introducing the distribution function $f(x,t) = \Phi(x,t)\Phi(x,t=0) = \Phi(x,t)\Psi_T(x)$, we can restate the (imaginary time) Schrödinger equation as

$$\begin{aligned} -\frac{\partial}{\partial t}f(x,t) \;=\; \Psi_T(x)\Big[-\frac{\partial}{\partial t}\Phi(x,t)\Big] &= \Psi_T(x)\left(\widehat{\mathbf{H}} - E_T\right)\Phi(x,t) \\ &= \Psi_T(x)\left(\widehat{\mathbf{H}} - E_T\right)\Psi_T(x)^{-1}f(x,t) \\ &= -\frac{1}{2}\Psi_T(x)\nabla^2\left(\Psi_T(x)^{-1}f(x,t)\right) + \widehat{\mathbf{V}}f(x,t) - E_T f(x,t) \end{aligned} \tag{3.26}$$

Expanding the Laplacian term further reveals

$$\begin{aligned} K(x,t) &\equiv -\frac{1}{2}\Psi_T(x)\nabla^2\left(\Psi_T(x)^{-1}f(x,t)\right) \\ &= -\frac{1}{2}\Psi_T(x)\nabla\cdot\left(\nabla\left[\Psi_T(x)^{-1}f(x,t)\right]\right) \qquad (3.27) \\ \nabla\left[\Psi_T(x)^{-1}f(x,t)\right] &= -\Psi_T(x)^{-2}\nabla\Psi_T(x)f(x,t) + \Psi_T(x)^{-1}\nabla f(x,t) \qquad (3.28) \end{aligned}$$

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

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

In order to clean up the messy calculations, we introduce the following identity

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

which inserted into the expression for $K(x,t)$ reveals

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

Inserting the expression for the Quantum Force $\vec{F}(x) = 2\Psi_T(x)^{-1}\nabla\Psi_T(x)$ and the local kinetic energy $K_L(x) = -\frac{1}{2}\Psi_T(x)^{-1}\nabla^2\Psi_T(x)$ simplifies the expression dramatically

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

Inserting everything back into Eq. (3.26) yields

$$\begin{aligned} -\frac{\partial}{\partial t}f(x,t) &= -\frac{1}{2}\nabla\cdot\left[\left(\nabla-\vec{F}(x)\right)f(x,t)\right] + K_L(x)f(x,t) + \widehat{\mathbf{V}}f(x,t) - E_T f(x,t) \\ \frac{\partial}{\partial t}f(x,t) &= \frac{1}{2}\nabla\cdot\left[\left(\nabla-\vec{F}(x)\right)f(x,t)\right] - \left(E_L(x)-E_T\right)f(x,t), \end{aligned} \tag{3.29}$$

which is a Fokker-Planck diffusion equation (Eq. (3.21)) with constant shift representing the shape of the branching in the case Fokker-Planck QMC (see Eq. (3.39).

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, we switch sampling distribution to one which are similar to the original integrand, i.e. smoothing out the sampled function, where as for Quantum Monte-Carlo, we use a function which is constructed with the sole purpose of imitating the exact ground state to suggest moves more efficiently. It is therefore reasonable to call the use of Fokker-Planck diffusion *Importance sampled Quantum Monte-Carlo*.

Introducing diffusion by the Fokker-Planck equation does not violate the Schrödinger equation, it simply alters the physical interpretation of the distribution of walkers. This new function, $\Psi_T(x)$, is referred to as a *trial wave function*, and will be discussed in more detail in Section 3.5.

3.2 Diffusive Equilibrium Constraints

Upon convergence of a Markov process, the system at hand will reach its most likely state. This is exactly the behaviour of a real system of diffusing particles described by statistical mechanics: It will *thermalize*, that is, be in its most likely state given a surrounding temperature. Markov processes are hence beloved by physicists; they are simple, yet realistic.

Once thermalization is reached, average 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. We can introduce rules of acceptance and rejection of transitions purposed by following the solutions of the Langevin Equation (Eq. (3.20) and Eq. (3.25)), but not without constraints. If any of the following conditions break, we have no guarantee that the system will thermalize properly:

3.2.1 Detailed Balance

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

$$P_i W(i \to j) = P_j W(j \to i), \tag{3.30}$$

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

3.2.2 Ergodicity

Another requirement is that the sampling must be ergodic, that is, the Markov chain (or simpler: The walker) needs to be able to be able to reach any configuration state 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 hence breaks the ergodicity requirement.

3.3 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. I will not go into details about transport theory in this section, but rather start the derivation from the criteria of detailed balance, Eq. (3.30), modeling the transition probability as two-part: $g(i \to j)$, the probability of selecting configuration j given configuration i, times a probability of accepting the selected move, $A(i \to j)$:

$$\begin{aligned} P_i W(i \to j) &= P_j W(j \to i) \\ P_i g(i \to j) A(i \to j) &= P_j g(j \to i) A(j \to i) \end{aligned} \tag{3.31}$$

Inserting the probability distribution as the Wave function squared, and the selection probability as our Green's function, then gives us a simple expression:

$$\begin{aligned} |\psi_i|^2 G(i \to j) A(i \to j) &= |\psi_j|^2 G(j \to i) A(j \to i) \\ \frac{A(j \to i)}{A(i \to j)} &= \frac{G(i \to j)}{G(j \to i)} \frac{|\psi_i|^2}{|\psi_j|^2} \equiv R_G(j \to i) R_\psi(j \to i)^2 \end{aligned} \tag{3.32}$$

Assume now that configuration i has a higher overall probability than configuration j, that is, $A(i \to j) = 1$. A more effective thermalization is obtained by accepting all these moves. What saves us from breaking the criteria of ergodicity is the fact that we do not reject the move otherwise. Instead, we insert $A(i \to j) = 1$ into Eq. (3.32), and thus ensuring detailed balance as well as ergodicity. 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:

$$A(i \to j) = \begin{cases} R_G(i \to j) R_\psi(i \to j)^2 & R_G(i \to j) R_\psi(i \to j)^2 < 1 \\ 1 & \text{else} \end{cases} \tag{3.33}$$

Or more simplistic:

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

For the isotropic case, we have a cancellation of the Greens function due to symmetry, $R_G(i \to j) = 1$, leaving us with the standard Metropolis algorithm:

$$A(i \to j) = \min\{R_\psi(i \to j)^2, 1\} \tag{3.35}$$

If we on the other hand use the Fokker-Planck equation, we will not get a cancellation. Inserting Eq. (3.23) into Eq. (3.33) results in the *Metropolis Hastings algorithm*. The ratio of Green's function can be evaluated efficiently by simply subtracting the exponents of the exponentials

$$\begin{aligned} R_G^{\mathrm{FP}}(i \to j) &= G_{\mathrm{Diff}}^{\mathrm{FP}}(j \to i)/G_{\mathrm{Diff}}^{\mathrm{FP}}(i \to j) \\ &= \exp\Big[\frac{1}{2}\big(F(x_j)+F(x_i)\big)\big(\frac{1}{2}D\delta t(F(x_j)-F(x_i))+x_i-x_i\big)\Big] \end{aligned} \tag{3.36}$$

$$A(i \to j) = \min\{R_G^{\mathrm{FP}}(i \to j)R_\psi(i \to j)^2,\, 1\} \tag{3.37}$$

Derived from detailed balance, the Metroplis Algorithm is as a must-have when it comes to Markov Chain Monte Carlo. Besides Quantum Monte Carlo, we have methods such as the *Ising Model* which greatly benefit from these rules [11].

In practice, without the Metropolis sampling, our 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 position of the walkers are random. A chart flow describing the implementation of the Metropolis algorithm and the diffusion process in general is given in Fig. 3.1.

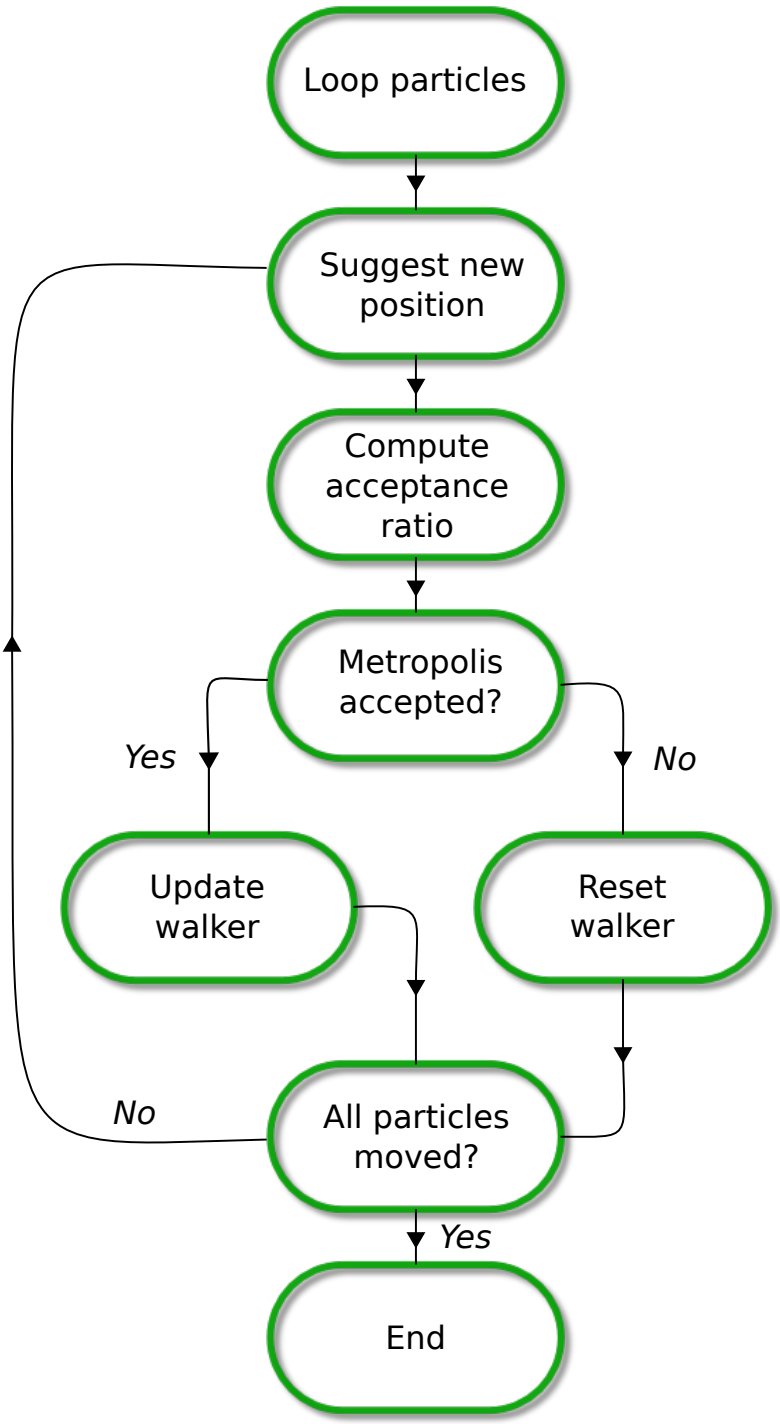

Figure 3.1: Flow chart for iterating a walker through a single time step, i.e. simulation the application of the Green's function from Eq. (3.16) using the Metropolis algorithm. New positions are suggested according to the chosen diffusion model.

3.4 The Process of Branching

The process of branching of Quantum Monte-Carlo is simulated by the creation or destruction of walkers with probability equal to that of the Green's function in Eq. (3.17) [7]. The explicit shapes in case of isotropic diffusion (ISO) and Anisotropic (FP) is

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

$$G_B^{\mathrm{FP}}(i \to j) = e^{-\left(\frac{1}{2}[E_L(x_i)+E_L(x_j)]-E_T\right)\delta\tau}, \tag{3.39}$$

where $E_L(x_i)$ is the energy evaluated in configuration x_i (see Section 3.5.3 for details).

The Green's function represents a weight of the walker in a current position. When the final estimates are calculated, this weight is factored in to the distribution function. This however, is not a practical way of doing calculations; we cannot calculate a factor for every possible transition. What is done instead is to introduce a birth/death process. If the value is less than one, we have "deaths", i.e. the walker is removed from the ensemble. "Births" occur when the Green's function is greater than one; the walker is replicated. As for all Green's functions, $G_B(i \to j) = 1$ implies no change.

The practical implementation becomes simply creating $[\mathrm{floor}(G_B) - 1]$ replicas of the current walker, with $[G_B - \mathrm{floor}(G_B)]$ probability of adding one extra on top (I skip the transition indices for now). As an example, a Green's function of $G_B = 3.3$ will have guaranteed three replicas, however, there is a $3.3 - 3 = 0.3$ chance that one additional replica is made. The efficient comparison to do is to define

$$\overline{G}_B = \mathrm{floor}\left(G_B + a\right), \tag{3.40}$$

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 required. The three different scenarios which arise is

- $\overline{G}_B = 1$: No branching, proceed main loop.
- $\overline{G}_B = 0$: The current walker is to be removed from the current ensemble.
- $\overline{G}_B > 1$: Make $\overline{G}_B - 1$ replicas of the current walker.

This process is demonstrated in Fig. 3.2.

There are some programming challenges due to the fact that the number of walkers is not conserved, such as cleaning up dead walkers and stabilizing the population across different nodes. For details regarding this, see the actual code reference at ref. [12]. Isotropic diffusion is in practice never used with branching. From Eq. (3.38) we see that in case of the Coulomb interaction, the branching Green's function would have singularities, leading to large fluctuations in the walker population. This is exactly opposite of the optimal behaviour of the system.

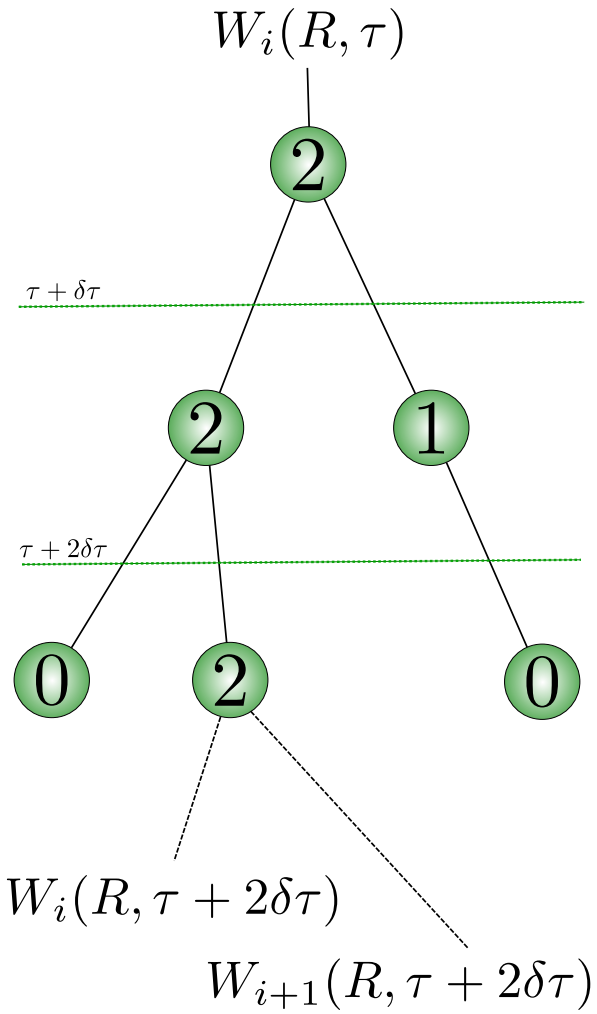

Figure 3.2: Branching illustrated. The initial walker $W_i(R,\tau)$ is branched according to the rules of Section 3.4. The numerical value inside the nodes represents $\overline{G}_B$ from Eq. (3.40). Each horizontal dashed line represent a diffusion step, i.e. a transition in time. Two lines exiting 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).

3.5 The Trial Wave Function

Recall Eq. (3.2)-(3.5). The initial condition, $\Phi(\vec{r}, t = 0)$, 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. (3.5), the entire diffusion 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.

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 ref. [5]. For mathematically rigid derivations of concepts, see ref. [13]. More details regarding many-body theory can be found in ref. [14].

3.5.1 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. If this was not the case, i.e. for non-interacting particles, the full system would decouple into N single particle systems.

Finding the ground state is not surprisingly equivalent to solving the time-independent Schrödinger Equation

$$\widehat{\mathbf{H}}\Psi_0(x) = E_0\Psi_0(x), \tag{3.41}$$

where $x \equiv \{x_1, x_2, ..., x_N\}$. Exact solutions to realistic many-body systems rarely exist, however, like in Section 3.1.1, expanding the solution in a known basis is always legal, which reduces the problem into that of a *coefficient hunt*

$$\Psi_0(x) = \sum_{k=0}^{\infty} C'_k\Phi_k(x). \tag{3.42}$$

Different many-body methods, e.g. *Hartree Fock*[2] and genetic algorithms, give rise to different ways of hunting down these coefficients, however, certain concepts are necessarily common, for instance truncating the basis at some level, K:

$$\Psi_0(x) = \sum_{k=0}^{K} \tilde{C}'_k\Phi_k(x). \tag{3.43}$$

The many-body basis elements $\Phi_k(x)$ are constructed using N elements from a basis of single particle wave functions (or *orbitals* for short), $\phi_n(x_i)$, 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 state n at a position x_i is then described by the orbital $\phi_n(x_i)$. Each unique[3] configuration of electrons will give rise to one

[2] 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.

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

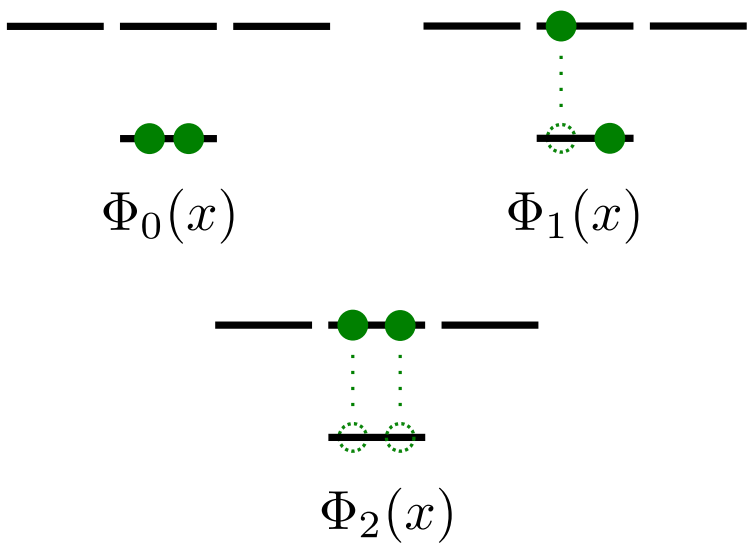

Figure 3.3: Three different electron configurations in an shell structure making up three different $\Phi_k(x)$, i.e. constituents of the many-body basis described in Eq. (3.43). An electron (solid dot) is represented by e.g. the orbital $\phi_{1s}(x_1)$.

unique $\Phi_k(x)$. In other words, the complete basis of $\Phi_k(x)$ is described by the collection of all possible excited states. $\Phi_0(x)$ is the ground state of the atom, $\Phi_1(x)$ has one electron exited to a higher shell, $\Phi_2(x)$ has another, and so on. See Fig. 3.3 for a demonstration of this.

In other words, constructing a single solution of a many-body problem involves three steps:

Step one	Choose a set of orbitals $\phi_n(x_i)$.
Step two	Construct $\Phi_k(x)$ from $N\times$ $\phi_n(x_i)$.
Step three	Construct $\Psi_k(x)$ from $K\times$ $\phi_k(x)$.

The last step is well described by Eq. (3.43), but is seldom necessary to perform explicitly; expressions based on the calculated wave functions is given in terms of the constituents and their weights.

Step one in detail

The Hamiltonian for a N-particle system is

$$\widehat{\mathbf{H}} = \widehat{\mathbf{H}}_0 + \widehat{\mathbf{H}}_{\mathrm{I}}, \tag{3.44}$$

where $\widehat{\mathbf{H}}_0$ and $\widehat{\mathbf{H}}_{\mathrm{I}}$ are respectively the one-body and the many-body term. As an approximation, we truncate the many-body interactions at the Coulomb level. The one-body term consist of the external

potential and the kinetic terms for all particles.

$$\begin{aligned}\widehat{\mathbf{H}}_0 &= \sum_{i=1}^{N} \widehat{\mathbf{h}}_0(x_i) \qquad (3.45)\\ &= \sum_{i=1}^{N} \widehat{\mathbf{t}}(x_i) + \widehat{\mathbf{u}}_{\text{ext}}(x_i)\\ \widehat{\mathbf{H}}_{\text{I}} &\simeq \sum_{i<j=1}^{N} \widehat{\mathbf{v}}(r_{ij}) \qquad (3.46)\\ &= \sum_{i<j=1}^{N} \frac{1}{r_{ij}}\end{aligned}$$

The single particle orbitals are chosen to be the solutions of the non-interacting case (given that they exist)

$$\widehat{\mathbf{h}}_0(x_i)\phi_n(x_i) = \epsilon_n \phi_n(x_i). \qquad (3.47)$$

If no such choice can be made, choosing an generally suited basis, e.g. free-particle solutions, is the general strategy.

Step two in detail

In the case of *Fermions*, i.e. half-integer spin particles like electrons, protons, etc., $\Phi_k(x)$ is an anti-symmetric function[4] on the form of a determinant: The *Slater determinant*. The shape of the determinant is given in Eq. (3.48). 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. (3.49)), 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.

$$\begin{aligned}\Phi_0^{\text{AS}}(x_1, x_2, ..., x_N) &\propto \sum_{\text{P}} (-)^{\text{P}} \widehat{\mathbf{P}} \phi_1(x_1)\phi_2(x_2)...\phi_N(x_N)\\ &= \begin{vmatrix} \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_N(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \cdots & \phi_N(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(x_N) & \phi_2(x_N) & \cdots & \phi_N(x_N) \end{vmatrix} \qquad (3.48)\end{aligned}$$

$$\Phi_0^{\text{S}}(x_1, x_2, ..., x_N) \propto \sum_{\text{P}} \widehat{\mathbf{P}} \phi_1(x_1)\phi_2(x_2)...\phi_N(x_N) \qquad (3.49)$$

Comment to Eq. (3.48) and Eq. (3.49): 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.

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

Dealing with correlations

The contributions to the sum on the right-hand side in Eq. (3.42) for $k > 0$ are often referred to as *correlation* terms. Given that the orbital wave functions are chosen by Eq. (3.47), 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}.

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 position independent.

In other words, incorporating the correct correlated wave function will result in a cancellation in the diverging term as we approach singularities, a property which brings us closer to the exact wave function.

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

3.5.2 Choice of Trial Wave function

Choosing the trial wave function boils down to optimizing the overlap described in the introduction 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. (3.43), 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 *anzats* for the trial wave function becomes

$$\Psi_T(x_1, ..., x_N) = \Big[\sum_{k=0}^{K} C_k \Phi_k(x_1, ..., x_N)\Big] \prod_{i<j}^{N} f(r_{ij}) \tag{3.50}$$

Explicit shapes

Several models for the correlation function exist, however, some are less practical than others. An example given in ref. [7] demonstrates this nicely: Hylleraas presented the following correlation function

$$f(r_{ij})_{\mathrm{Hylleraas}} = e^{-\frac{1}{2}(r_i + r_j)} \sum_k d_k (r_{ij})^{a_k} (r_i + r_j)^{b_k} (r_i - r_j)^{e_k}, \tag{3.51}$$

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.

A more well suited correlation function 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 well behaving, it is custom to drop the second term, and keep only the $k = 1$ term, i.e.

$$f(r_{ij};\beta) = \exp\left(\frac{a_{ij}r_{ij}}{1+\beta r_{ij}}\right), \tag{3.52}$$

where β is a variational parameter, and a_{ij} is a spin-dependent constant tuned to obey the cusp condition.

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 is holding back the effectiveness of many-particle simulations. However, assuming we have a spin-independent Hamiltonian, we can split the spatial wave function is two; one part for each spin eigenvalue. A detailed derivation of this is given in the appendix of ref. [15]. Assuming spin-half particles we get

$$\Psi_T(x_1,...,x_N;\beta) = \Big[\sum_{k=0}^{K} C_k\tilde{\Phi}_k(x_1,...,x_{\frac{N}{2}})\tilde{\Phi}_k(x_{\frac{N}{2}-1},...,x_N)\Big]\prod_{i<j}^{N} f(r_{ij};\beta). \tag{3.53}$$

Since the particles are identical, we are free to say that the first half are spin up, and the second spin down, hence the splitting as above. For simplicity, 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

$$\Psi_T(x_1,...,x_N;\beta) = \sum_{k=0}^{K} C_k D_k^{\uparrow} D_k^{\downarrow}\prod_{i<j}^{N} f(r_{ij};\beta). \tag{3.54}$$

This shape is referred to as a *Multi-determinant* trial wave function.

Limitations

Depending on the complexity of the system at hand, we need more complicated trial wave functions. 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 tweak 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 ($t = 0$) into a distribution closer to the exact wave function by Eq. (3.5).

Because of this fact, our limitations due to the trial wave function is far less than what is the case of standard Monte-Carlo integration. A heavier 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.

Single Determinant Trial Wave function

In the case of well-behaving systems, a single determinant converges rapidly enough. This simplicity opens up the possibility of simulating large systems efficiently. This will be referred to as a *single determinant* trial wave function, and serve as a very simple approximation. In order to optimize the overlap with the exact wave function, a variational parameter α is introduced in the spatial part (from Eq. (3.52), we already have β)

$$\Psi_T(x_1, ..., x_N; \alpha, \beta) = D^{\uparrow}(\alpha) D^{\downarrow}(\alpha) \prod_{i<j}^{N} f(r_{ij}; \beta). \tag{3.55}$$

Determining optimal values for the variational parameters will be discussed in Section 3.5.5.

3.5.3 Calculating Expectation Values

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

$$\begin{aligned}
\langle \Psi_T | \, \widehat{\mathbf{O}} \, | \Psi_T \rangle &= \int \Psi_T(x)^* \widehat{\mathbf{O}} \Psi_T(x) \mathrm{d}x \\
&= \int |\Psi_T|^2 \left(\frac{1}{\Psi_T(x)} \widehat{\mathbf{O}} \Psi_T(x) \right) \mathrm{d}x \\
&= \int |\Psi_T|^2 O_L(x) \mathrm{d}x \qquad (3.56) \\
O_L(x) &= \frac{1}{\Psi_T(x)} \widehat{\mathbf{O}} \Psi_T(x) \qquad (3.57)
\end{aligned}$$

Discretizing the integral (and thus introducing an error) yields

$$\langle \Psi_T | \, \widehat{\mathbf{O}} \, | \Psi_T \rangle \equiv \langle O \rangle \simeq \frac{1}{n} \sum_{i=1}^{n} O_L(x_i) \equiv \overline{O}, \tag{3.58}$$

where x_i is a random variable following the 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.

$$\langle O \rangle = \lim_{n \to \infty} \overline{O} = \lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} O_L(x_i) \tag{3.59}$$

In the case of the energy estimation, this means that once our walkers hit equilibrium, we can start sampling local values based on their configurations x_i. In the case of energies, we get

$$\langle E \rangle \simeq \frac{1}{n} \sum_{i=1}^{n} \left(\frac{1}{\Psi_T(x_i)} \left(-\frac{1}{2} \nabla^2 \right) \Psi_T(x_i) + V(x_i) \right) \tag{3.60}$$

3.5.4 Normalization

Every explicit calculation using the trial wave function in Quantum Monte-Carlo involves taking ratios. Ratios implies cancellation of the normalization factors. Eq. (3.33) 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. (3.48) and Eq. (3.49), 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: Single determinant 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.

3.5.5 Selecting Optimal Variational Parameters

All practical ways of determining the optimal values of the variational parameters originate from the same powerful principle: *The Variational Principle*. The easiest way of demonstrating the principle is to evaluate the expectation value of the energy, using an approach similar to what used in Eq. (3.5)

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

Using further that E_0 is the lowest energy eigenvalue, we can, just as done with the time propagator, introduce $E_k = E_0 + \delta E_k$ to 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 stunning: No matter how we choose our trial wave function, we will never undershoot the exact ground state energy. This basically means that the problem of choosing variational parameters boils down to a minimization problem in the parameters space (we assume no maximum exist for finite parameter values)

$$\frac{\partial\langle E\rangle}{\partial\alpha_i} = \frac{\partial}{\partial\alpha_i}\langle\Psi_T(\alpha_i)|\,\widehat{\mathbf{H}}\,|\Psi_T(\alpha_i)\rangle = 0 \tag{3.61}$$

In order to work with Eq. (3.61) in practice, we need to rewrite it in terms of known values. Since our wave function is dependent on the variational parameter, we need to include the expression for the normalization factor before we can expand the expression

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

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

$$\begin{aligned}\frac{\partial\langle E\rangle}{\partial\alpha_i} &= 2\left(\frac{\langle\Psi_T(\alpha_i)|\,\widehat{\mathbf{H}}\frac{\partial}{\partial\alpha_i}\,|\Psi_T(\alpha_i)\rangle}{\langle\Psi_T(\alpha_i)|\Psi_T(\alpha_i)\rangle} - \langle E\rangle\frac{\langle\Psi_T(\alpha_i)|\,\frac{\partial}{\partial\alpha_i}\,|\Psi_T(\alpha_i)\rangle}{\langle\Psi_T(\alpha_i)|\Psi_T(\alpha_i)\rangle}\right) \\ &= 2\left(\left\langle E\frac{\partial\Psi_T}{\partial\alpha_i}\right\rangle - \langle E\rangle\left\langle\frac{\partial\Psi_T}{\partial\alpha_i}\right\rangle\right) \qquad (3.62)\end{aligned}$$

Using this expression for the *variational gradient* means we can calculate the derivatives exactly the same way we calculate the energy, and use these derivatives to move in the direction of the variational minimum in Eq. (3.61).

This strategy give rise to numerous ways of finding the optimal parameters, such as using the well known Newton's method, conjugate gradient methods [16], 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.

3.6 Gradient Descent Methods

The direction of a gradient serves as a guide to extremal values. Gradient Descent, also called Steepest Descent, is a family of minimization methods using this property of gradients in order to back trace a local minimum in the vicinity of an initial guess.

3.6.1 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

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

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

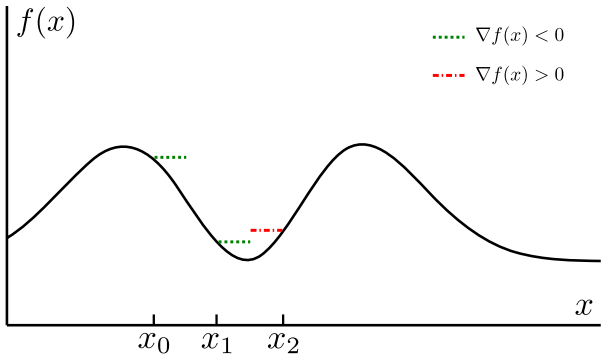

Figure 3.4: Two steps of a one dimensional Gradient Descent process. Steps are taken in the direction of the negative gradient (indicated by dotted lines).

Assume now that we start from an initial position x_0, and measure the direction of the gradient before we take a step in that direction. From Fig. 3.4 and the previous equations, we see that once we cross the minimum, the gradient changes sign. The brute force way of minimizing is to stop once the gradient changes sign, however, this would require extremely many extremely small steps in order to achieve good precision. The difference equation for the brute force case would be

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

A better algorithm is to continue iterating even though the minimum is crossed. The brute force scheme breaks down in this case, oscillating between two points, e.g. x_1 and x_2 in Fig. 3.4, because of the constant step length δ. To counter this, a changing step length δ_i is introduced

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

All Gradient/Steepest Descent methods is in principle described by Eq. (3.67). 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 $\quad \delta_i = \delta / i^2$
- 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. Calculating the local gradient is simply a finite difference calculation (assuming the analytic expression is not known).

3.6.2 Stochastic Gradient Descent

Minimizing stochastic quantities, such as the variance or expectation values, adds another layer of complications on top of the methods described in the previous section. Assuming a closed form expression for

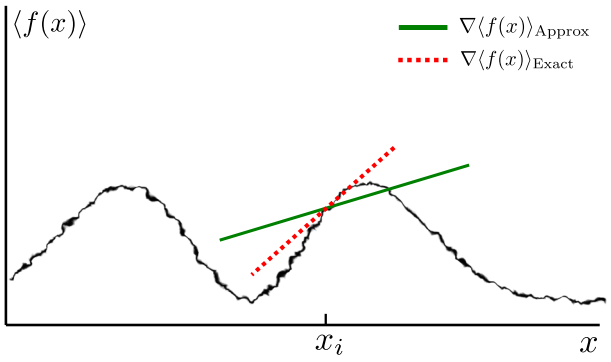

Figure 3.5: A one dimensional plot of an expectation valued function. Smeared lines are representing uncertainties due to rough sampling. The direction of the local gradient (solid green line) at a point x_i is not necessarily a good estimate of the actual analytic gradient (dashed red line).

the stochastic quantity is unobtainable, the gradient needs to be calculated by using e.g. Monte-Carlo sampling. Eq. (3.62) 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. 3.5.

3.6.3 Adaptive Stochastic Gradient Descent

Adaptive Stochastic Gradient Descent (ASGD) has it's roots in the mathematics of automated control theory [17]. 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

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

The step length from Eq. (3.67) is in ASGD modeled in the following manner

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

with $f_{\max} > 0$, $f_{\min} < 0$, and $\omega > 0$. Free parameters are a, A, t_0, however, Ref. [18] suggests $A = 20$ and $t_0 = t_1 = A$ for universal usage.

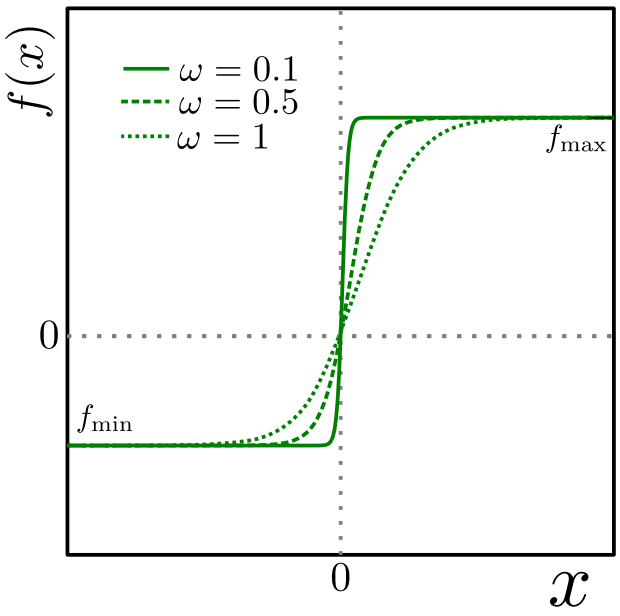

Figure 3.6: Examples of $f(X_i)$ as published in ref. [18]. As $\omega \to 0$, $f(x)$ approaches a step function.

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 of $f(x)$ is to alter 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. We know from previous discussion, that if the sign of the gradient change, we have crossed the minimum. Crossing the minimum with ASGD has the following consequence

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

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

Assumptions

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

This is shown in ref. [18] 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 $t = 0$.

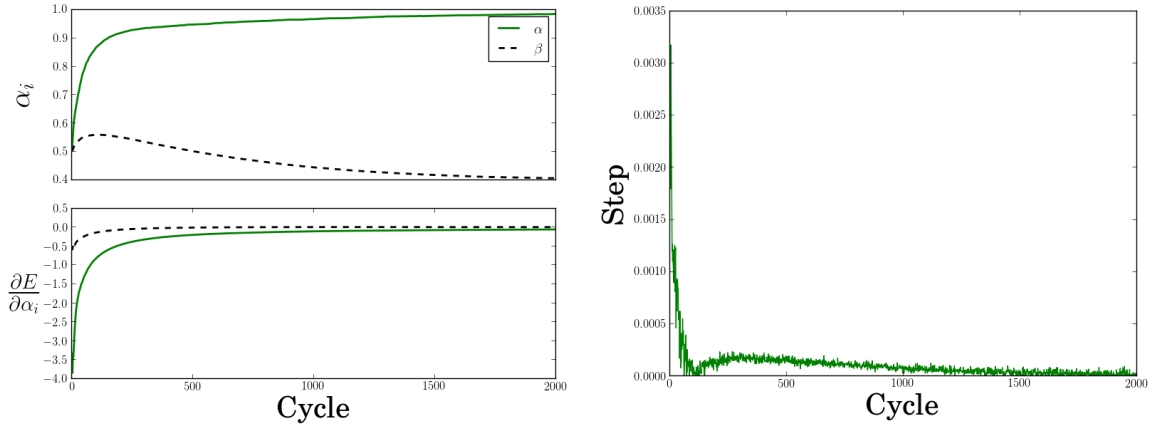

Figure 3.7: Results of Adaptive Stochastic Gradient Descent used on a two-particle Quantum Dot with unit oscillator frequency using 400 cycles pr. gradient sampling and 40 independent walkers. The right figure shows the evolution of the time step. The left figure shows the evolution of the variational parameters α and β introduced in Section 3.5 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.

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

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

This is exactly the behavior displayed in Fig. 3.6.

These assumptions and several more are described in more detail in ref. [18]. They underline that the shape and flow of the algorithm is in no way random; ASGD is optimizing minimization of stochastic quantities.

Implementation

A flow chart of the implementation is given in Fig. 3.8. For specific details regarding the implementation, i refer to the code [12]. An example of minimization using ASGD is given in Fig. 3.7.

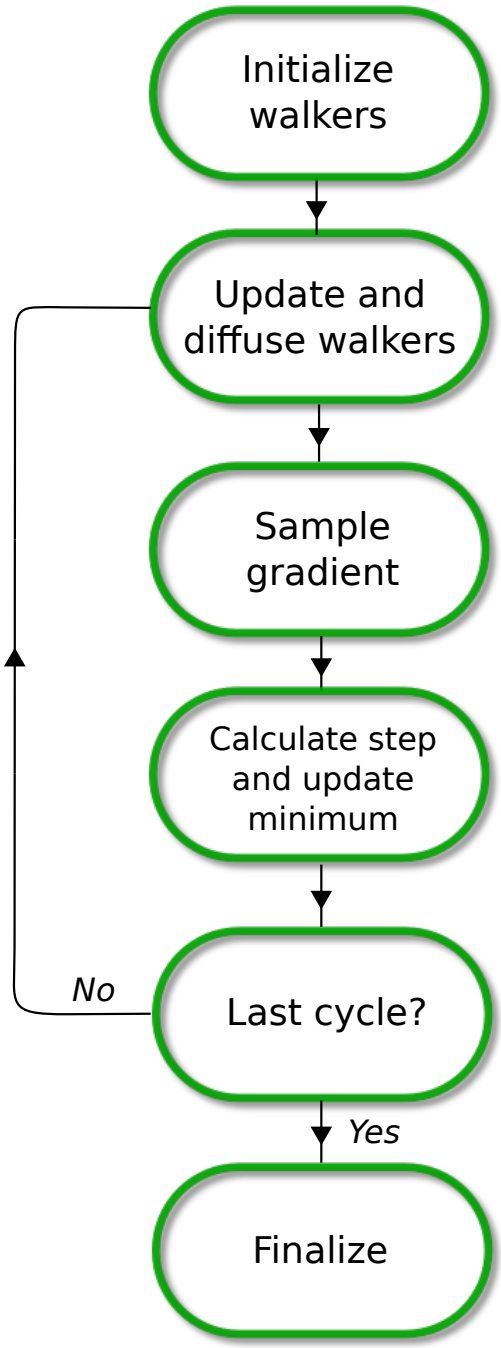

Figure 3.8: Chart flow of ASGD algorithm. Diffusing a walker is done as described in Fig. 3.1. Updating the walkers involves recalculating any values afflicted by updating the minimum. The step is calculated by Eq. (3.70). In case of Quantum Monte-Carlo, the gradient is sampled by Eq. (3.62).

3.7 Variational Monte-Carlo

As briefly mentioned in the derivation of the Quantum Monte-Carlo projection process, neglecting the branching term, i.e. force $G_{\mathrm{B}} = 1$, 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 3.5.5). The better the trial wave function, the better the answer.

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. 3.9, it is clear that VMC corresponds to nothing but a standard Monte-Carlo integration of the local energy, using the trial wave function for distributing the points.

3.7.1 Motivating the use of Diffusion Theory

The question becomes: Why bother with all the diffusion theory if the result is simply an expectation value? From statistics we know that we may use *any* distribution when calculating an expectation value. Why bother with a trial wave function, thermalization, etc.?

The reason is simple, yet not obvious. The quantity of interest, the local energy, is *dependent on a wave function* $\psi(x)$. It is not necessarily smooth, and even though the wave functions tend to zero at infinities, the local energy is *undefined*; a "0/0" expression (see Eq. (3.73). This is the case for all the roots of $\psi(x)$. Given an arbitrary distribution $P(x)$, we get

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

where the points x_i are drawn from the distribution $P(x)$.

We *need* to importance sample the integral with the distribution corresponding to that of the wave function used in the local energy in order to solve the integral in a satisfying manner, that is, avoid sampling close to the roots of $\psi(x)$ without "cheating". Introducing importance sampling is done by simply switching distributions (we do not need to scale the sampled function, we are computing an expectation value, not an arbitrary integral).

$$P(x) = |\psi(x)|^2$$

Suggesting new positions (diffusion) boils down to be analogous to calling a *random number generator* corresponding to our trial wave function squared distribution. The problem was the roots of $\psi(x)$, 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.

$$\psi(x_m) = 0 \quad \Longrightarrow \quad P(x_m) = 0 \tag{3.74}$$

In other words, the more undefined the energy is at a point, the less probable the point is. That is what we need, and it's what all the theory of Quantum Monte-Carlo safely delivers that standard Monte-Carlo can not.

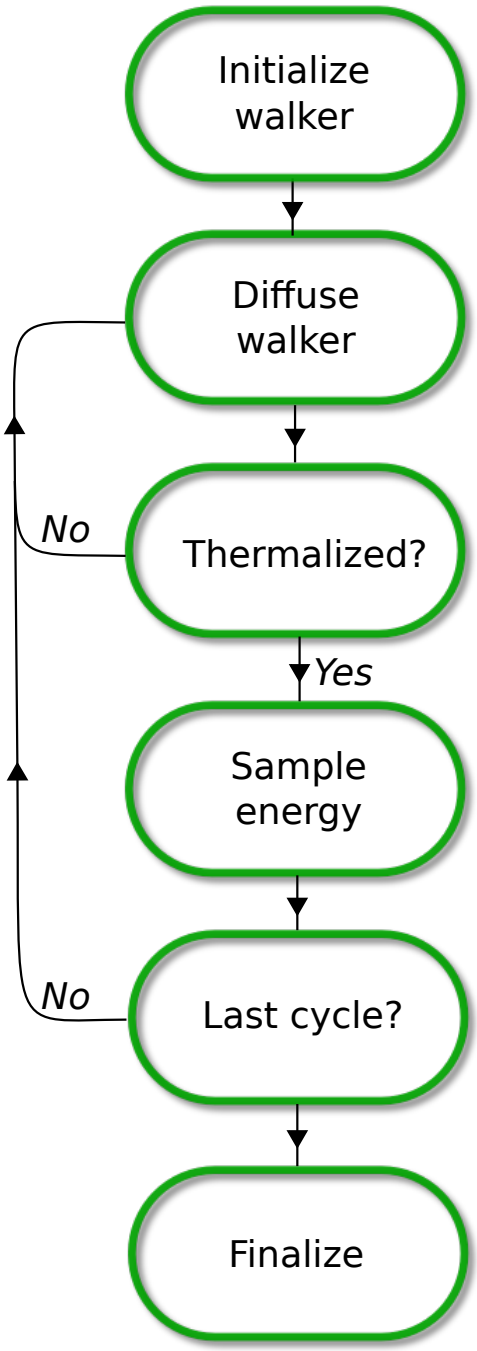

Figure 3.9: Chart flow of the Variational Monte-Carlo algorithm. The second step, *Diffuse Walker*, is the process described in Fig. 3.1. Energies are sampled as described in Section 3.5.3. Thermalization is usually set to a fixed number of cycles.

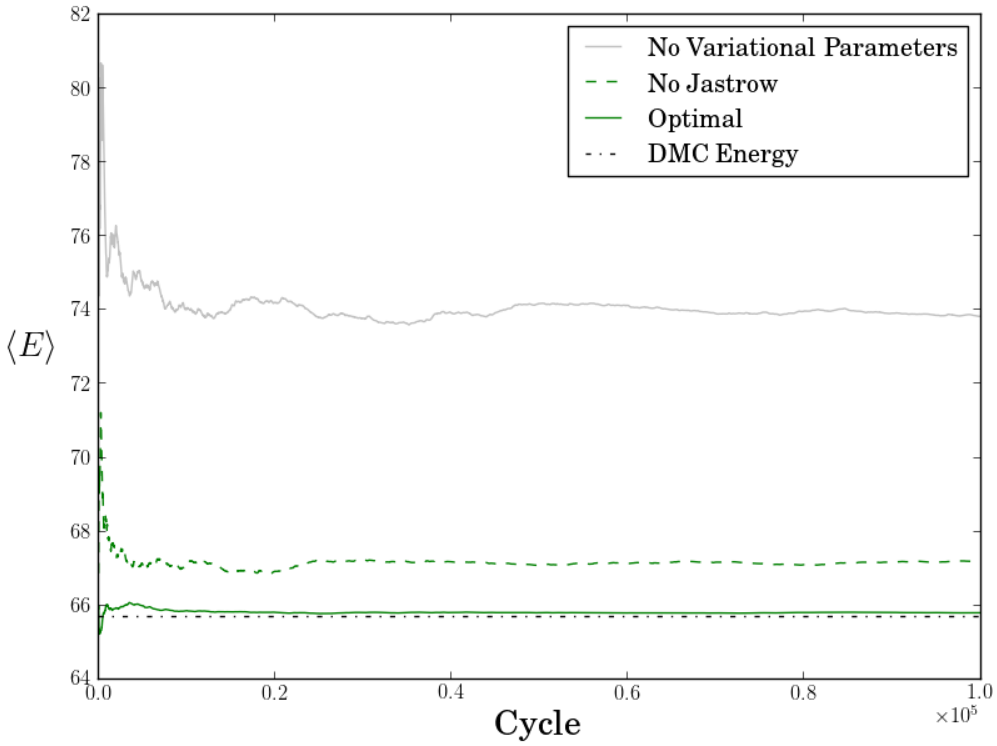

Figure 3.10: Comparison of the VMC energy using different trial wave functions. The DMC energy is believed to be very close to the exact ground state. We see that adding a variational parameter to the trial wave function lowers the energy somewhat, however, when adding the Jastrow factor described in Section 3.5.2 we get 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 is used.

3.7.2 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. 3.1. A flow chart of the VMC algorithm is given in Fig. 3.9. Finalizing the sampling involves scaling energies, calculating the variance, etc.

For details regarding specific implementations, I refer to the code in ref. [12].

3.7.3 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. (3.5) approach unity, the VMC energy approaches the exact ground state energy as a monotone decreasing function. Fig. 3.10 described this effect.

3.8 Diffusion Monte-Carlo

Applying the full diffusion framework introduced in the previous sections results in a method known as Diffusion Monte-Carlo (DMC)[5]. 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. 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, in order to reach convergence to the "exact" ground state in a satisfactory way.

3.8.1 Implementation

DMC is a precise method; more statistics is like water in a desert - you can not get 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. Normally one would only use two: The time-step and walker loops, however, introducing a third *block loop* within the walker loop will boost convergence dramatically. This loop continues until the current walker is either dead, 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. 3.11.

3.8.2 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 k, 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. Then we have

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

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

The new trial energy (remember Eq. (3.7)) is set to be

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

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

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

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

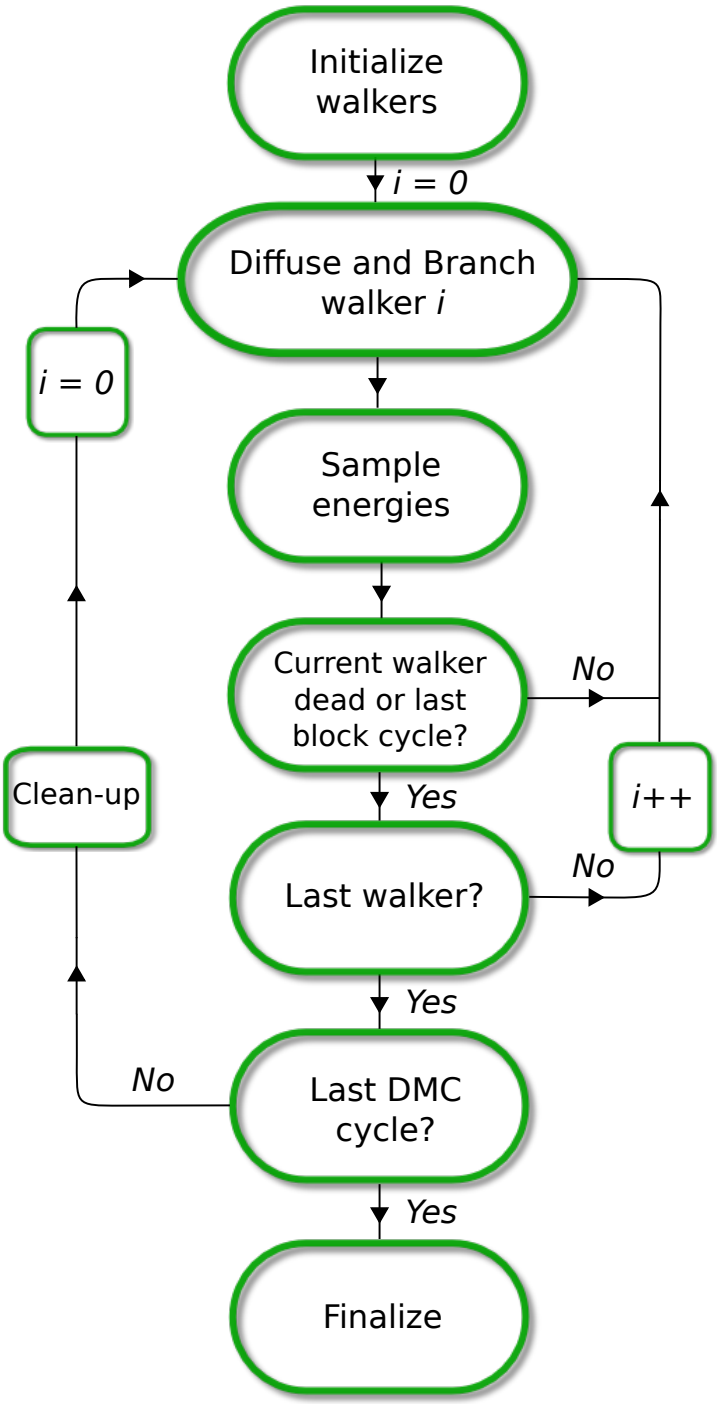

Figure 3.11: Chart flow of the Diffusion Monte-Carlo algorithm. The count variable i is the index of the walker loop. The second step, *Diffuse and Branch Walker*, is the process described in Fig. 3.1 in combination with the branching from Fig. 3.2. Energies are sampled as described in Section 3.8. Thermalization is not done in the same way as VMC (see Fig. 3.9), but rather includes the entire flow with a reduced number of DMC - and block cycles.

3.8.3 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 *Plug and Play*.

Time Step Errors

The error introduced by the short time approximation goes as $\mathcal{O}(\delta\tau^2)$ (see Eq. (3.15)). 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 more detail reveals that it's magnitude increase exponentially with the spread of the energies

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

As will be shown in Section 3.9.1, the spread in energy samples are higher the worse of an approximation to the ground state our trial wave function becomes. In addition, the magnitude of the spread scales with the magnitude of the energy. Due to finite computer memory, we have N slots 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.

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. Every time we perform a time-step without having the trial energy close to the exact energy, we introduce an error to the Quantum Monte-Carlo equations (remember the transition from Eq. (3.5) to Eq. (3.7)). It is therefore necessary to have rapid convergence down to a sufficiently low energy level, where the magnitude of the exact energy error matches those of the other systematic errors.

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

3.8.4 Fixed node approximation

Looking at Eq. (3.49), we notice that by choosing positive phases for our single particle wave functions, the bosonic many body wave function is exclusively positive. For fermions however, the sign change upon permuting a particle pair introduces the possibility that the wave function will have both negative and positive regions, independent of our choice of phases in the single particle wave functions.

As we simulate importance sampled DMC in time, the density of walkers at a given time, $P(x, \tau)$, is given by Eq. (3.5) multiplied by the trial wave function (see the end of Section 3.1.4 for details)

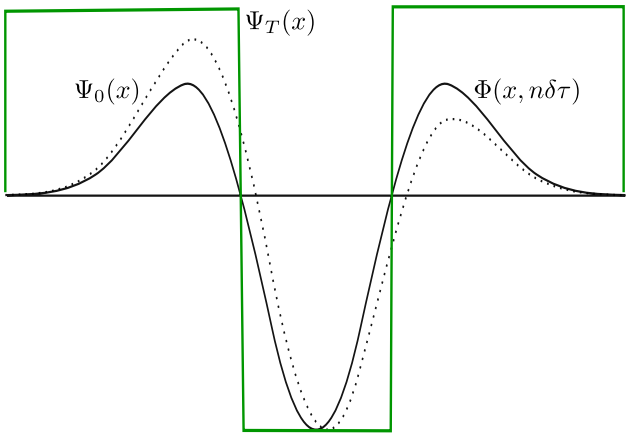

Figure 3.12: An illustration of the fixed node approximation. The dotted line is the exact ground state $\Psi_0(x)$. The distribution of walkers at cycle n, $\Phi(x, n\delta\tau)$, similar in shape with $\Psi_0(x)$, however, sharing nodes with the trial wave function $\Psi_T(x)$ (box-like function for illustration purposes), and thus making it impossible to match the true ground state exactly.

$$P(x,\tau) = \Phi(x,\tau)\Psi_T(x), \tag{3.79}$$

where

$$\lim_{\tau\to\infty} P(x,\tau) = \Phi_0(x)\Psi_T(x), \tag{3.80}$$

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, $\Phi(x,\tau)$ and $\Psi_T(x)$ have to change sign together[6]. 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:

$$\frac{\Psi_T(x_i)}{\Psi_T(x_j)} < 0 \quad \Longrightarrow \quad A(i \to j) = 0 \tag{3.81}$$

where $A(i \to j)$ is the probability of accepting the move, as described in Section 3.3. An illustrative example is attempted in Fig. 3.12.

3.9 Estimating the Statistical Error

As with any statistical result, appropriate statistical errors needs to be included for it to be taken seriously. Systematic errors, that is, errors introduced due to limitations in the model, will be discussed in each method's respective section. Statistical errors, that is, the deviation from the true ensemble average due

[6]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 region.

to the fact that we can never fulfill the equality in Eq. (3.59), can be estimated using several methods, some of which are *naive* in the sense that they assume the dataset to be completely *uncorrelated*.

3.9.1 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

$$\begin{aligned}
\mathrm{Var}(E) &= \left\langle (E - \langle E \rangle)^2 \right\rangle \\
&= \left\langle E^2 \right\rangle - 2 \underbrace{\langle E \langle E \rangle \rangle}_{\langle E \rangle \langle E \rangle} + \langle E \rangle^2 \\
&= \left\langle E^2 \right\rangle - \langle E \rangle^2 \qquad (3.82) \\
&\simeq \overline{E^2} - \overline{E}^2 \qquad (3.83)
\end{aligned}$$

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 a fit different trial wave functions are to the system. Note however, a common misconception is to use the numerical value of the variance to compare *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 fit. This is not true. The variance has units (in the case of local energies) energy squared, 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,

$$\sigma^2(x) = \mathrm{Var}(x), \qquad (3.84)$$

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 $\langle E \rangle$.

3.9.2 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

$$\begin{aligned}
\mathrm{Cov}(x,y) &\equiv \langle (x-\langle x\rangle)(y-\langle y\rangle)\rangle \\
&= \langle xy - x\langle y\rangle - \langle x\rangle y + \langle x\rangle\langle y\rangle\rangle \\
&= \langle xy\rangle - \langle x\langle y\rangle\rangle \underbrace{- \langle y\langle x\rangle\rangle + \langle\langle x\rangle\langle y\rangle\rangle}_{0} \\
&= \langle xy\rangle - \langle x\rangle\langle y\rangle\,. \qquad (3.85)
\end{aligned}$$

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

The consequence of ignoring correlations is an 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. Denoting the true standard deviation as σ_c, the above discussion can be distilled to

$$\sigma_c(x) \geq \sigma(x), \qquad (3.86)$$

where σ is the deviation from Eq. (3.84).

3.9.3 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 it's combined mean, i.e.

$$\sigma(\overline{x}) \neq \sigma(x). \qquad (3.87)$$

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 all the simulations.

$$\begin{aligned}
m &\equiv \overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad (3.88)\\
\sigma^2(m) &= \langle m^2\rangle - \langle m\rangle^2 \qquad (3.89)
\end{aligned}$$

Combining the above equations yields

$$\begin{aligned}
\sigma^2(m) &= \left\langle \frac{1}{n^2}\left[\sum_{i=1}^{n} x_i\right]^2 \right\rangle - \left\langle \frac{1}{n}\sum_{i=1}^{n} x_i \right\rangle^2 \\
&= \frac{1}{n^2}\left(\left\langle \sum_{i=1}^{n} x_i \sum_{j=1}^{n} x_j \right\rangle - \left\langle \sum_{i=1}^{n} x_i \right\rangle \left\langle \sum_{j=1}^{n} x_j \right\rangle \right) \\
&= \frac{1}{n^2}\sum_{i,j=1}^{n} \langle x_i x_j\rangle - \langle x_i\rangle\langle x_j\rangle \\
&= \frac{1}{n^2}\sum_{i,j=1}^{n} \mathrm{Cov}(x_i, x_j) \qquad (3.90)
\end{aligned}$$

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 our samples are uncorrelated. Going back to the definition of covariance in Eq. (3.85), we see that in order to calculate the covariance as in Eq. (3.90), we need to know the true mean $\langle x_i \rangle$. Using $m = \overline{x}$ as an approximation to the true mean yields

$$\begin{aligned} \mathrm{Cov}(x_i, x_j) &\equiv \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \\ &\simeq \langle (x_i - m)(x_j - m) \rangle \\ &\simeq \frac{1}{n^2} \sum_{k,l=1}^{n} (x_k - m)(x_l - m) \qquad (3.91) \\ &\equiv \frac{1}{n} \mathrm{Cov}(x) \qquad (3.92) \end{aligned}$$

Inserting this relation into Eq. (3.90) yields

$$\begin{aligned} \sigma^2(m) &= \frac{1}{n^2} \sum_{i,j=1}^{n} \mathrm{Cov}(x_i, x_j) \\ &\simeq \frac{1}{n^2} \sum_{i,j=1}^{n} \frac{1}{n} \mathrm{Cov}(x) \\ &= \frac{1}{n^3} \mathrm{Cov}(x) \underbrace{\sum_{i,j=1}^{n}}_{n^2} \\ &= \frac{1}{n} \mathrm{Cov}(x), \qquad (3.93) \end{aligned}$$

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

Explicitly computing the covariance is rarely done in Monte-Carlo simulations; if the sample size is large, it is extremely expensive. A variety of alternative methods to counter the correlations are available, the simplest of which is to define a *correlation length*, τ[7], 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 $\overline{x}$

$$\overline{x} = \frac{1}{n} \sum_{k=0}^{n} x_{k \cdot \tau} \qquad (3.94)$$

This basically means that we need $n\tau$ samples in order to get the same magnitude of samples to our average as in Eq. (3.88); the *effective sample size* becomes $n_{\mathrm{eff}} = n_{\mathrm{tot}}/\tau$. $\tau = 1$ defines the uncorrelated case. For details regarding the derivations of τ based on the covariance, see ref. [19] and ref. [11].

3.9.4 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,

[7] This parameter is often referred to as the *auto-correlation time* in the literature.

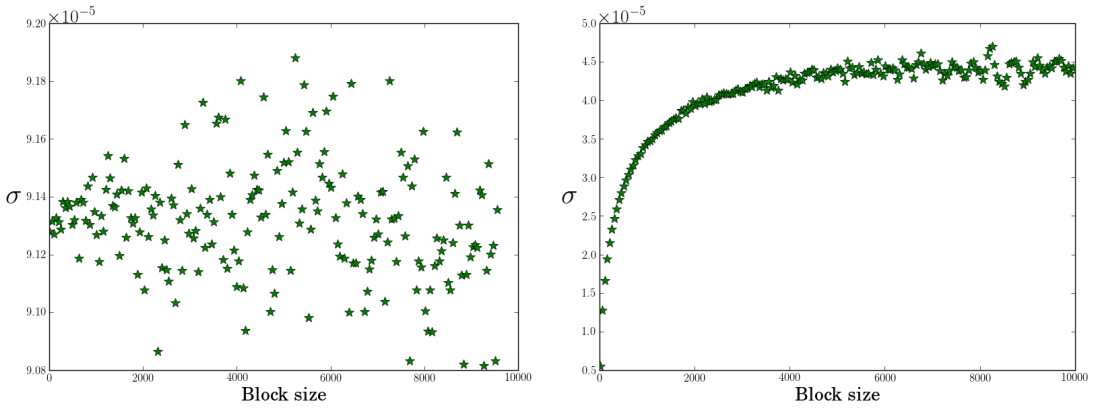

Figure 3.13: Left hand side: Blocking result of (approximately) uncorrelated data generated from a uniform Monte-Carlo integration of $\int_1^2 2x\mathrm{d}x$ resulting in 3.00003 (exact is 3.0). This is in excellent agreement with the magnitude of the error $\sim 9 \cdot 10^{-5}$. On the right hand side we have the blocking result (DMC) for 6-particle $\omega = 0.1$ Quantum Dot, used in Table 5.2. We clearly see the plateau discussed in the blocking section, resulting in a total error in the range of $4.5 \cdot 10^{-5}$ to $5 \cdot 10^{-5}$.

i.e. you do not need to know the error at any stage during the sampling. This means that we can post process the error calculation (given that we store the data sets).

An efficient algorithm for calculating the error of correlated data is *blocking*. This method is described in high detail in ref. [19], 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 increase as n decrease, (see Eq. (3.93))

$$\sigma_n \propto \frac{1}{\sqrt{n}} \tag{3.95}$$

However, treating each block as an individual simulation, we have n_b averages m_n that we can use to calculate the total error in Eq. (3.89), that is, estimate the covariance

$$\overline{m_n^r} \equiv \frac{1}{n_b}\sum_{k=1}^{n_b} m_n^r \tag{3.96}$$

$$\begin{aligned}\sigma^2(m) &= \langle m^2\rangle - \langle m\rangle^2 \\ &\simeq \overline{m_n^2} - (\overline{m_n})^2 \end{aligned} \tag{3.97}$$

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. What we do know however, is that if the system is correlated, there should be a range of different block sizes which fulfill Eq. (3.97) to reasonable precision. The result of a blocking analysis is therefore a series of (n, $\sigma(m)$) pairs. Plotting these pairs should in light of previous arguments result in a increasing curve which stabilizes over a certain span of block sizes. This plateau will then serve as a reasonable approximation to the covariance. See Fig. 3.13 for a demonstration of the blocking technique.

3.9.5 Variance Estimators

The standard intuitive variance estimator given by

$$\sigma^2(x) \simeq \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2 = \left(\frac{1}{n}\sum_{i=1}^{n}x_i^2\right) - \overline{x}^2 \tag{3.98}$$

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

$$\sigma^2(x) \simeq \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2 = \left(\frac{1}{n-1}\sum_{i=1}^{n}x_i^2\right) - \frac{n}{n-1}\overline{x}^2 \tag{3.99}$$

which is only noticeably different from Eq. (3.98) when the sample size gets small, as in blocking analysis. It is therefore standard to use Eq. (3.99) for blocking errors.

Part II

Results

4

Implementation and Validation

I will not discuss the specific implementations in this thesis. For a detailed description of specific functions etc., see the the actual code for comments. The concept of this chapter is to give insight about the ideas behind the implementation. Long story short, alot of hard work has been put into deep object orientation (for details, see Section 2.2) in order to combine the different building blocks of QMC-methods in a natural and coherent way. As with all big coding projects, a major part of development went into planning and structuring.

4.1 Structure and Implementation

In QMC, Quantum Mechanics, and even physics in general, there are natural ways of decoupling the code in order to create a bridge between physics and code. Gathering data into objects representing physical concept to which a reader can relate increases the overall readability of the code. It also dramatically decreases the time it takes to implement or debug new methods, since mathematical intuition can be used in order to trace certain behaviours back to the source [1]. Another reason is to generalize the code for several different cases, without having to rewrite or mess up anything (see the PotionGame example in Section 2.2.5).

4.1.1 Methods used for Increasing Readability and Overall Structure

As an example, the contents of the `Walker` class in Table 4.1. The idea behind this structure is that whenever we need a new walker in e.g. DMC, all we need to do is to create a new instance of `Walker`. Deleting a walker is just as clean. A function which requires access to several elements from Table 4.1 now only requires one argument, namely the walker of interest. Let us look at an example

```
1 DMC::DMC(...) {
2
3     ...
4
5     original_walkers = new Walker*[spesify a number of walkers];
6
7     ...
```

[1] For instance, if something is wrong with the sampling rule, a random walker might behave weird. The natural starting point of debugging is then the object in which the sampling rules are set. In the case of this thesis, this is the Sampling class containing a Diffusion object.

Type	Name	Description
mat	r	The position.
mat	r_rel	The relative positions.
rowvec	r2	The squared positions.
mat	phi	The last evaluated single particle orbitals.
field<mat>	dell_phi	The last evaluated gradients of the single particle orbitals.
cube	dJ	The last evaluated terms of the jastrow factor derivatives.
mat	spatial_grad	The gradient of the uncorrelated wave function.
mat	jast_grad	The gradient of the Jastrow factor.
mat	inv	The inverse of the slater matrix.
mat	qforce	The quantum force.
double	spatial_ratio	The current ratio between this walker and another walker.
double	value	The value of the wave function.
double	lapl_sum	The full Laplacian of the wave function.
double	E	The energy of the walker.
bool	is_murdered	A flag for the DMC brancing algorithm.

Table 4.1: Description of the members of the Walker class. All matrices holds information on all particles. The second block corresponds to vectors kept in memory to avoid expensive recalulation. The third block corresponds to scalar values calculated and kept in memory for later use or control.

```
8
9 }
```

```
1 void DMC::initialize() {
2
3     ...
4
5     //Initializing active walkers
6     for (int k = 0; k < n_w; k++){
7         original_walkers[k] = new Walker(n_p, dim);
8     }
9
10    //Seting trial position of active walkers
11    ...
12
13    //Calculating and storing energies of active walkers
14    for (int k = 0; k < n_w; k++) {
15        calculate_energy_necessities(original_walkers[k]);
16        original_walkers[k]->set_E(calculate_local_energy(
              original_walkers[k]));
17    }
18
19    ...
20
21 }
```

Initializing new walkers is, as seen above, unproblematic. Calculating values in the machinery now always involves a corresponding walker. For instance, when looping over walkers in DMC, the amount of juggling is reduced to nothing; all you need to do is to loop over a vector of walkers. This walker can then be sent to any function, resulting in code like e.g.

```
1 double Coulomb::get_pot_E(const Walker* walker) const {
2
3     double e_coulomb = 0;
4
5     for (int i = 0; i < n_p - 1; i++) {
6         for (int j = i + 1; j < n_p; j++) {
7             e_coulomb += 1 / walker->r_rel(i, j);
8         }
```

```
    }

    return e_coulomb;
}
```

The alternative to the code above is to juggle one relative position matrix per walker, ruining both the readability and the overall structure of the code. Another upside to this way of structuring, is that we can tie the source code and the method description closer together. Look at VMC as an example. VMC uses a walker and a trial walker.

```
void VMC::initialize() {

    ...

    sampling->set_trial_pos(original_walker);
    copy_walker(original_walker, trial_walker);

    ...

}
```

Another example where object orientation dramatically increases the readability of the code is the interplay between the Sampling- and Diffusion-class. From **REF TO THEORY** we know that if we use importance sampling, the diffusion follows the Fokker-Planck equation (Eq. **CITE EQ FOKKER-PLANCK**). The implementation is as follows:

```
Importance::Importance(GeneralParams & gP)
: Sampling(gP.n_p, gP.dim) {

    diffusion = new Fokker_Planck(n_p, dim, gP.random_seed);

}
```

4.1.2 Methods for Generalizing the Code

The importance sampling constructor serves as a good example in this case as well. A `Sampling` object type might be an instance of `Importance` or `Brute_Force`, however, we do not need to know this in order to abstractly describe how to diffuse a walker. We do not even need to know whether we are doing VMC or DMC. All we need to know is that the `Diffusion` object within the sampling holds the rules we need once the correct objects are in place. This is reflected in the following code:

```
void Sampling::update_pos(const Walker* walker_pre, Walker* walker_post
    , int particle) const {

    for (int j = 0; j < dim; j++) {
        walker_post->r(particle, j) = walker_pre->r(particle, j)
                + diffusion->get_new_pos(walker_pre, particle, j);
    }

    //more updates through jastrow pointers etc.
    ...

}
```

```
double Diffusion::get_new_pos(const Walker* walker, int i, int j) {
    return gaussian_deviate(&random_seed) * std;
}

double Simple::get_new_pos(const Walker* walker, int i, int j) {
```

```
6       return Diffusion::get_new_pos(walker, i, j);
7   }
8
9   double Fokker_Planck::get_new_pos(const Walker* walker, int i, int j) {
10      return D * timestep * walker->qforce(i, j) + Diffusion::get_new_pos
            (walker, i, j);
11  }
```

This use of polymorphism (as described in Section 2.2.2) is widely used throughout the code to generalize it. As a goal, the idea was to produce a code with the following generalizations:

- As many as possible functions should be written generally for DMC and VMC.
- Objects should not assume the type of any sub-classed object except its own, unless the type is directly implied (importance sampling implies Fokker-Planck diffusion), i.e., deep polymorphism.

This puts a series of constraints on the code; it should be general for (in any combination):

(i) No trailing if-tests or flags handling switched cases. A single test in the main file decides everything once and for all.

(ii) Importance- and Brute Force-sampling.

(iii) Numerical or closed form expressions for the kinetic energy and quantum force.

(iv) Fermions and Bosons.

(v) Any choice of single particle basis, included expanded bases.

(vi) Functionality to add any combination of any potential.

Constraints (i) - (iv)

As mentioned previously, `QMC` holds an object of type `Sampling`, which contains all the general functions for diffusing particles and how to sample them. It also ensures that the walker holds the necessary data in order to continue the sampling, e.g. the Quantum Force if we do importance sampling. This is achieved by having a pure virtual function `get_necessities`, which is overrided by the subclasses `Importance` and `Brute_Force`. In other words: The class structure is set up in such a way that the distinct parts which needs to be general are separated. Polymorphism takes care of the distinction, hence no if-tests are required. Below follows a part of the code illustrating this; the code for copying walkers, calculating energy necessities etc. follows the same idea.

```
1   void Sampling::update_pos(const Walker* walker_pre, Walker* walker_post
        , int particle) const {
2
3       //Positions and orbitals are updated for the particle at hand
4       ...
5
6       //updates the inverse slater in case of a fermion system
7       qmc->get_system_ptr()->calc_for_newpos(walker_pre, walker_post,
            particle);
8
9       //pure virtual function. Function will update quantum force if
            importance sampling.
10      update_necessities(walker_pre, walker_post, particle);
11
12  }
```

In the case of numerical energy calculations, the function which evaluates the gradients can be set to a general numerical function (assuming the standard single particle orbitals are implemented). If the closed form expressions are implemented, these can be accessed directly instead. Fermions and bosons have different implementations of e.g. the spatial ratio of a wavefunction.

Constraint (v)

A single particle orbital is nothing but a function of a walker's coordinates and variational parameters. An if-test hierarchy on the Quantum Number is the simplest way to implement a single particle basis, however, they can all be avoided using polymorphism. The class `BasisFunctions` represents an abstract function, which can takes on input a walker and evaluates an arbitrary expression. A subclass will hold the specific implementation, e.g. the first excited level of a harmonic oscillator.

```
class BasisFunctions {
public:
    BasisFunctions();

    virtual double eval(const Walker* walker, int i) = 0;
};

double alphaHO_3::eval(const Walker* walker, int i) {

    //4*k2*y2 - 2
    H = 4*(*k2)*walker->r(i, 1)*walker->r(i, 1) - 2;
    return H * (*exp_factor);

}
```

These functions are loaded into an array representing the single particle basis in the `Orbitals` class' constructor.

```
AlphaHarmonicOscillator::AlphaHarmonicOscillator(GeneralParams & gP,
    VariationalParams & vP)
: Orbitals(gP.n_p, gP.dim) {

    //Creating pointers in order to link the Orbital and BasisFunction
        parameters.
    this->alpha = new double();
    this->k = new double();
    this->k2 = new double();
    this->exp_factor = new double();

    this->w = gP.systemConstant;
    set_parameter(vP.alpha, 0);
    get_qnums();

    basis_functions[0] = new alphaHO_0(k, k2, exp_factor);
    basis_functions[1] = new alphaHO_1(k, k2, exp_factor);
    basis_functions[2] = new alphaHO_2(k, k2, exp_factor);
    basis_functions[3] = new alphaHO_3(k, k2, exp_factor);
    ...
```

All orbital files are generated automatically by a Python-script. The reasoning behind the unset pointers are to create a link between the parameters in `BasisFunctions` and `Orbitals`, such that we do not have to change the value in all of the basis function objects (60 for 30 particles), we simply have to change the value in the orbitals class and the rest will follow.

With this rigid setup, evaluating orbitals can now we done in the following manner (similar for the gradient and Laplace)

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

The reason why `phi` is a virtual function, is so that the `ExpandedBasis` class can override it. With this systematic setup of single particle orbitals, it is very simple and intuitive to implement the expanded basis functionality

```
class ExpandedBasis : public Orbitals {

...

protected:

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

};

ExpandedBasis::ExpandedBasis(GeneralParams & gp, Orbitals* basis, int
    basis_size, std::string coeffPath)
: Orbitals(gp.n_p, gp.dim) {

    this->basis = basis;
    this->basis_size = basis_size;
    coeffs = arma::zeros<arma::mat > (n2, basis_size);

    //read coefficients

}
```

The expanded basis class is implemented as a subclass to the original orbital class, however, it is designed to work along side it. Instead of a list of orbital basis functions, it holds a `Orbital` object, containing the basis in which we want to expand, alongside a matrix containing the coefficients of expansion. Implementing the new single particle functions is simply done by virtual function overriding as below:

```

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

    double value = 0;
    for (int m = 0; m < basis_size; m++) {
        value += coeffs(q_num, m) * basis->phi(walker, particle, m);
    }

    return value;

}
```

And of course similarly for the gradient and Laplace. The expression is very close to the raw mathematical description of a basis expansion.

Constraint (vi)

When we are loading a set of single particle states, we are loading those who best match the given Hamiltonian of our system. For quantum dots, we load harmonic oscillator states, for atoms we load hydrogen states and so on. By having great flexibility in both the Hamiltonian and the basis functions means the code is easily adaptable to other systems. The flexibility of the potential class is already discussed in Section 2.2.2.

The complete list of classes working in the same way (iteration over loaded elements, no if-tests) is

- `Potential` See section 2.2.2 for detials.
- `ErrorEstimator`: Initialized to `Blocking` or `SimpleVar`. Samples data.
- `OutputHandler`: Saves specified data to a specified file, e.g. `stdoutDMC`.

Creating optimized and general code takes longer to develop, but pays off when it comes to later implementations of extensions to the library.

4.1.3 Visualization

DCVIZ?

4.2 Performance Optimizations

4.2.1 RAM use

the RAM use of VMC is close to nothing; we only have two walkers with each their set of matrices. However, DMC has the potential to use a lot of RAM, as thousands of walkers are allocated. The walkers account for close to all of the RAM used by the methods, as the rest of the framework consist only of a small amount of doubles (8 bytes) and integers (4 bytes). A short analysis of the RAM spent per `Walker` object yields

•1 boolean	1 byte
•3 integers	12 bytes
•3 doubles	24 bytes
•4 `n_p` × `dim` matrices	32 `n_p` · `dim` bytes
•1 `n_p` × `n_p` matrix	8 `n_p` · `n_p` bytes
•2 `n_p` × `n_p/2` matrices	4 `n_p` · `n_p` bytes
•1 array of length `n_p`	8 `n_p` bytes
•1 field of length `n_p`	8 `n_p` bytes
•1 `n_p` × `n_p/2` × `dim` field	8 `n_p` bytes
•1 `n_p` × `n_p` × `dim` cube	8 `n_p` bytes
Total:	37 + 32 `n_p` · `dim` + 12 `n_p` · `n_p` + 8 `n_p` bytes

For VMC with 30 particles and quantum dots in 2 dimensions this gives a walker RAM usage of 25 kB, which is practically nothing. For DMC however, we have e.g. 1000 active walkers and 4000 inactive. The inactive ones can be left uninitialized, saving the RAM of all matrix initializations (37 RAM per walker).

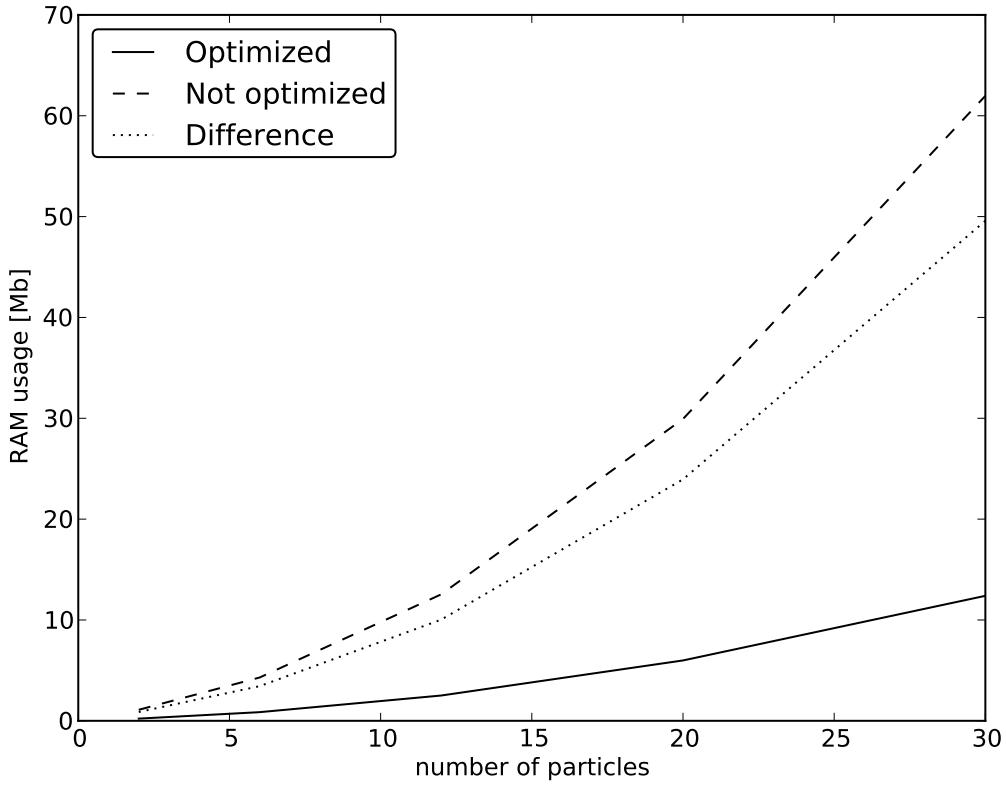

Figure 4.1: Number of particles vs. the theoretical RAM usage in a typical DMC run for the optimized and unoptimized case. Calculated for two dimensions, 1000 active walkers and 5000 inactive.

In Figure 4.2.1 we see how the RAM usage in a typical DMC calculation scales with the number of particles. For 30 particles, we have approximately 60 MB of RAM spent on walkers. This is still practically nothing compared to the available RAM on modern computers (minimum 2 GB).

The conclusion stands that as long as there are no memory leaks, time spent optimizing the memory use is time wasted. Memory optimizations might damage the readability, so the code was left more or less unoptimized in this part.

4.2.2 CPU-time

Looking at Table 4.1, the only thing we have to store in order for the machinery to work is the position matrix. Every other matrix is initialized to optimize CPU time.

During a Quantum Monte Carlo sampling process, certain values, such as the relative distances, are used to calculate quantities such as the energy. The most brute force way of handling situations like this is to calculate everything at the time it is needed. However, once a diffusion step is made,we use the same relative distances both in the Jastrow factor and its gradient. Calculating them twice is a waste of time. We can instead store them in a matrix, and access this matrix in both functions:

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

Another upside with this way of storing the relative distances, is that moving particle i in our code, only changes the i'th row in the matrix, and therefore we need only to recalculate N elements (N being the number of particles in our system). For the same reason, storing the gradients, Laplacian sums and the squared radii is matrices also optimize the code.

Having closed form expressions for the gradients and Laplacians for the different parts of the wave function is another source of dramatic speed-up. The local kinetic energy ($(E_k)_L$) and the Quantum Force ($\mathbf{F}_i$) can be expressed in terms of the separate parts of the wave function:

$$\begin{aligned} (E_k)_L &= -\frac{1}{2}\frac{1}{\psi_T}\nabla_i^2\psi_T \\ &= -\frac{1}{2}\frac{1}{|S|\psi_C}\nabla_i^2(|S|\psi_C) \\ &= -\frac{1}{2}\frac{1}{|S|\psi_C}\nabla_i\left(\psi_C\nabla_i|S| + |S|\nabla_i\psi_C\right) \\ &= -\frac{1}{2}\Big[\frac{1}{\psi_C}\nabla_i^2\psi_C + \frac{2}{|S|\psi_C}\nabla_i|S|\cdot\nabla_i\psi_C \\ &\qquad + \frac{1}{|S|}\nabla_i^2|S|\Big]. \end{aligned} \tag{4.1}$$

$$\begin{aligned} \mathbf{F}_i &= \frac{2}{|S|\psi_C}\nabla_i(|S|\psi_C) \\ &= 2\left(\frac{1}{\psi_C}\nabla_i\psi_C + \frac{1}{|S|}\nabla_i|S|\right), \end{aligned} \tag{4.2}$$

where i denotes particle number, and $|S|$ and ψ_C are respectively the spatial wave function and the Jastrow factor.

For Fermions, it can be shown that we have the following relations for the Slater determinant[20]:

$$\begin{aligned} \frac{1}{|S|}\nabla_i|S| &= \sum_{j=1}^{n}\nabla_i\phi_j(\vec{r}_i)S_{ji}^{-1} \\ \frac{1}{|S|}\nabla_i^2|S| &= \sum_{j=1}^{n}\nabla_i^2\phi_j(\vec{r}_i)S_{ji}^{-1}, \end{aligned} \tag{4.3}$$

where S^{-1} is the inverse of the Slater matrix, and n is the dimensionality of the Slater matrix, in our case $n = N/2$, where N is the number of electrons. $\phi_j(\vec{r}_i)$ are the single-particle basis functions, where j denotes the quantum numbers. For example $j = 0$ is the ground state, $j = 1$ first excited state and so on.

These values of the gradient and Laplacian of the single-particle wave functions needs to be tabulated. Most of the single particle bases used are expressed using simple mathematics, such as Hermite polynomials for the case of harmonic oscillator, and the derivatives can therefore be calculated pretty easily.

Calculating an inverse does not seem to optimize much. However, we can run an updating algorithm for the inverse, which ends up saving a lot of time. The ratio of the spatial determinants can also be expressed using this inverse[20]. This means that no explicit wave function calculation is needed (we can calculate the ratio between two Jastrow factors without problem). The expression for the ratio in terms of the inverse is

$$R_S = \sum_{j=1}^{n} \phi_j(\vec{r}_{i,\text{new}}) S^{-1}_{ji}(\vec{r}_{\text{old}}), \tag{4.4}$$

where i is the particle being moved.

In the case of $s = \frac{1}{2}$-Fermions, the code holds two slater determinants (spin up and down); we need two inverse matrices. All if-tests on whether or not to access spin up or down is however avoided by merging the two inverse matrices into one augmented matrix

$$S^{-1} = \left[S^{-1}_{\uparrow} S^{-1}_{\downarrow} \right].$$

This way of storing the data completely removes the need of if-tests on spin.

Updating the inverse matrix can be done once we got the new position and the ratio

$$S^{-1}_{kj}(\vec{r}_{\text{new}}) = \begin{cases} S^{-1}_{kj}(\vec{r}_{\text{old}}) - \frac{1}{R_S} S^{-1}_{ki}(\vec{r}_{\text{old}}) \sum_{l=1}^{n} \phi_l(\vec{r}_{i,\text{ new}}) S^{-1}_{lj}(\vec{r}_{\text{old}}) & j \neq i \\ \frac{1}{R_S} S^{-1}_{ki}(\vec{r}_{\text{old}}) & j = i \end{cases} \tag{4.5}$$

Equal expressions like the the ones listed in Eq. (4.4) can be found for the case of the Pade-Jastrow factor as well:

$$\begin{aligned} \frac{1}{\psi^{PJ}_C} \nabla_i \psi^{PJ}_C &= \sum_{j \neq i} \frac{\vec{r}_{ij}}{r_{ij}} \frac{a_{ij}}{(1+\beta r_{ij})^2} \\ \frac{1}{\psi^{PJ}_C} \nabla^2_i \psi^{PJ}_C &= \left| \frac{1}{\psi^{PJ}_C} \nabla_i \psi^{PJ}_C \right|^2 + 2 \sum_{i<j} \frac{a_{ij}(1-\beta r_{ij})}{r_{ij}(1+\beta r_{ij})^3}. \end{aligned} \tag{4.6}$$

Notice that a_{ij} is written as a matrix element. If we were to calculate a for every single run in the loop, the double if-tests would drain a lot of CPU time. Prior to the sampling, in the `Jastrow::initialize` function, the values of a are generated once and for all and stored in a matrix. This matrix remains unchanged throughout the entire sampling, and no if-tests are necessary.

For the simplest case of single particle harmonic oscillator basis functions (Eq. (**REF OSC BASIS**), the expressions for the derivatives of the wave function with respect to the variational parameters is

$$\frac{1}{\psi_T} \frac{\partial \psi_T}{\partial \alpha} = -\frac{1}{2} \omega \sum_{i=1}^{N} r_i^2, \tag{4.7}$$

$$\frac{1}{\psi_T} \frac{\partial \psi_T}{\partial \beta} = -\sum_{i<j} \frac{a_{ij} r_{ij}^2}{(1+\beta r_{ij})^2}. \tag{4.8}$$

which can be used to speed up the process of minimizing.

Optimization (without approximations) is all about not calculating more than you have to. Calculating gradients is the part of the code which require the most CPU time. The most time consuming functions is the gradients. Looking at Eq. (4.4) this comes as no surprise; the single particle wave functions contain a call to the exponential function, which is terribly slow and needs to be deadly accurate. The time consumption in the Jastrow gradient arise from the fact that once a particle is moved, the entire gradient changes.

4.3 Validation

things should not be wrong. It is bad.

test:

N	ω	E_{VMC}	E_{DMC}	α	β
2	0.28	1.02197	1.0216	0.970202	0.254158
	0.5	1.66023	1.65994	0.981901	0.312174
	1.0	3.00054	3.00002	0.988761	0.398956
6	0.28	7.62259	7.59967	0.87322	0.322838
	0.5	11.8092	11.7845	0.896501	0.411444
	1.0	20.1896	20.1606	0.920368	0.55734
12	0.28	25.7088	25.63814	0.797355	0.365122
	0.5	39.2395	39.15974	0.859145	0.481956
	1.0	65.7958	65.70032	0.873605	0.656703

Table 4.2: J. Hogberget

N	ω	E_{DMC}
2	0.28	N/A
	0.5	1.65975(2)
	1.0	3.00000(3)
6	0.28	7.6001(1)
	0.5	11.7888(2)
	1.0	20.1597(2)
12	0.28	25.6356(1)
	0.5	39.159(1)
	1.0	65.700(1)

Table 4.3: M. Pedersen Lohne et al.

N	ω	E_{VMC}	E_{DMC}	α
2	0.5	1.0	1.0	1.0
	1.0	2.0	2.0	1.0
6	0.5	5.0	5.0	1.0
	1.0	10.0	10.0	1.0
12	0.5	14.0	14.0	1.0
	1.0	28.0	28.0	1.0
20	0.5	30.0	30.0	1.0
	1.0	60.0	60.0	1.0
30	0.5	55.0	55.0	1.0
	1.0	110.0	110.0	1.0

Table 4.4:

5

Results

5.1 Validating the code

5.1.1 Calculation for non-interacting particles

5.2 QDOTS RESULTS

[2]

5.3 FIXED NODE TESTS

N	ω	E_{VMC}	E_{DMC}	E^a_{ref}	E^b_{ref}
2	0.1	0.44130(5)	0.44079(1)	-	-
	0.28	1.02215(5)	1.02164(1)	0.99263	-
	0.5	1.66021(5)	1.65977(1)	1.643871	1.65975(2)
	1.0	3.00030(5)	3.00000(1)	2.9902683	3.00000(3)
6	0.1	3.5690(3)	3.55385(5)	3.49991	-
	0.28	7.6216(4)	7.60019(6)	7.56972	7.6001(1)
	0.5	11.8103(4)	11.78484(6)	11.76228	11.7888(2)
	1.0	20.1902(4)	20.15932(8)	20.14393	20.1597(2)
12	0.1	12.3162(5)	12.26984(8)	12.2253	-
	0.28	25.7015(6)	25.63577(9)	25.61084	-
	0.5	39.2343(6)	39.1596(1)	39.13899	39.159(1)
	1.0	65.7905(7)	65.7001(1)	65.68304	65.700(1)
20	0.1	30.0729(8)	29.9779(1)	29.95345	-
	0.28	62.0543(8)	61.9268(1)	61.91368	61.922(2)
	0.5	94.0236(9)	93.8752(1)	93.86145	93.867(3)
	1.0	156.062(1)	155.8822(1)	155.8665	155.868(6)
30	0.1	60.584(1)	60.4205(2)	60.43000	-
	0.28	124.181(1)	123.9683(2)	123.9733	-
	0.5	187.294(1)	187.0426(2)	187.0408	-
	1.0	308.858(1)	308.5627(2)	308.5536	-
42	0.1	107.881(1)	107.6389(2)	-	-
	0.28	220.161(1)	219.8426(2)	219.8836	-
	0.5	331.002(1)	330.6306(2)	330.6485	-
	1.0	544.2(8)	542.9428(8)	542.9528	-

Table 5.1: Results for Quantum Dots with fixed node approximation calculated on the cluster Abel using 10^8 VMC cycles, 64000 walkers, with 2000 DMC cycles on 128 cores. Ref: *a*: [*Sarah*], *b*: [21]

N	ω	E_{VMC}	E_{DMC}	E^a_{ref}
2	0.1	0.44128(5)	0.44079(1)	-
	0.28	1.02216(4)	1.02164(1)	-
	0.5	1.66025(4)	1.65977(1)	1.65975(2)
	1.0	3.00036(4)	3.00000(1)	3.00000(3)
6	0.1	3.5693(3)	3.55374(5)	-
	0.28	7.6214(3)	7.60016(5)	7.6001(1)
	0.5	11.8103(3)	11.78489(6)	11.7888(2)
	1.0	20.1906(4)	20.15945(7)	20.1597(2)
12	0.1	12.3159(5)	12.26986(8)	-
	0.28	25.7000(6)	25.6358(1)	-
	0.5	39.2351(6)	39.1594(1)	39.159(1)
	1.0	65.7905(6)	65.7000(1)	65.700(1)
20	0.1	30.0732(8)	29.9779(2)	-
	0.28	62.0511(9)	61.9265(2)	61.922(2)
	0.5	94.0247(9)	93.8752(2)	93.867(3)
	1.0	156.0630(9)	155.8821(2)	155.868(6)
30	0.1	60.585(1)	60.4207(2)	-
	0.28	124.181(1)	123.9682(2)	-
	0.5	187.293(1)	187.0430(2)	-
	1.0	308.859(1)	308.5626(2)	-
42	0.1	107.8800(4)	107.638(2)	-
	0.28	220.1(2)	219.8426(3)	-
	0.5	331.002(4)	330.6307(2)	-
	1.0	-	-	-

Table 5.2: Results for Quantum Dots without fixed node approximation calculated on the cluster Abel using 10^8 VMC cycles, 64000 walkers, with 2000 DMC cycles on 128 cores. Ref *a*: [21]

A

Dirac Notation

Due to the orthogonal nature of Hermitian operators' eigenfunctions[1], the inner product between two states constructed from them will result in a lot of integrals being zero, one, or eigenvalues for that matter. Writing the integrals in their full form then feels like a waste of space and time. Even specifying e.g. a position basis is obsolete. Abstracting the wave functions from a given parameter space (e.g. $\mathbb{C}^n$) into a *Hilbert space*[2] is what is called the *Dirac notation*, or the *Bra-ket notation*.

The basic idea is that since the coordinate representation of a wave function is the projection of an abstract state on the position basis through an inner product, we can separate the different pieces of the inner product:

$$\psi(\vec{r}) = \langle r, \psi_j \rangle \equiv \langle r|\psi_j\rangle = \langle r| \times |\psi_j\rangle \,.$$

The notation is designed to be simple. The right hand side of the inner product is called a *ket*, while the left hand side is called a *bra*. Combining both of them leaves you with an inner product bracket, hence the names. Let us look at an example where this notation is extremely powerful. Imagine a coupled two-particle spin-$\frac{1}{2}$ system in the following state

$$|\psi\rangle = N\Big[\,|\uparrow\downarrow\rangle - i\,|\downarrow\uparrow\rangle\,\Big] \tag{A.1}$$

$$\langle\psi| = N\Big[\,\langle\uparrow\downarrow| + i\,\langle\downarrow\uparrow|\,\Big] \tag{A.2}$$

Using the fact that both the full two-particle state and the two-level spin states should be orthonormal, we can with this notation calculate the normalization factor without explicitly calculating anything.

$$\begin{aligned}
\langle\psi|\psi\rangle &= N^2\Big[\,\langle\uparrow\downarrow| + i\,\langle\downarrow\uparrow|\,\Big]\Big[\,|\uparrow\downarrow\rangle - i\,|\downarrow\uparrow\rangle\,\Big] \\
&= N^2\Big[\,\langle\uparrow\downarrow\,|\,\uparrow\downarrow\rangle + i\,\langle\downarrow\uparrow\,|\,\uparrow\downarrow\rangle - i\,\langle\uparrow\downarrow\,|\,\downarrow\uparrow\rangle + \langle\downarrow\uparrow\,|\,\downarrow\uparrow\rangle\,\Big] \\
&= N^2\Big[1 + 0 - 0 + 1\Big] \\
&= 2N^2
\end{aligned}$$

[1] Eigenfunctions of a Hermitian operator always make up a complete orthogonal set.

[2] A Hilbert space is an inner product space spanned by the different states. For every state, there exists a complementary state which is the Hermitian conjugate of the original[5].

This implies as we expected $N = 1/\sqrt{2}$. With this powerful notation at hand, we can easily show properties such as the *completeness relation* of a set. We start by expanding one state $|\phi\rangle$ in a complete set of different states $|\psi_i\rangle$:

$$\begin{aligned} |\phi\rangle &= \sum_i c_i |\psi_i\rangle \\ \langle\psi_k|\phi\rangle &= \sum_i c_i \langle\psi_k|\psi_i\rangle \\ &= c_k \\ |\phi\rangle &= \sum_i \langle\psi_i|\phi\rangle |\psi_i\rangle \\ &= \left[\sum_i |\psi_i\rangle \langle\psi_i|\right] |\phi\rangle , \end{aligned}$$

which implies that

$$\sum_i |\psi_i\rangle \langle\psi_i| = \mathbb{1} \tag{A.3}$$

for any complete set of orthonormal states $|\psi_i\rangle$. For a continuous basis like e.g. the position basis we have a a similar relation:

$$\int |\psi(x)|^2 dx = 1 \tag{A.4}$$

$$\begin{aligned} \int |\psi(x)|^2 dx &= \int \psi^*(x)\psi(x)dx \\ &= \int \langle\psi|x\rangle \langle x|\psi\rangle \, dx \\ &= \langle\psi| \left[\int |x\rangle \langle x| \, dx\right] |\psi\rangle . \end{aligned} \tag{A.5}$$

Combining eq. A.4 and eq. A.5 with the fact that $\langle\psi|\psi\rangle = 1$ yields the identity

$$\int |x\rangle \langle x| \, dx = \mathbb{1}. \tag{A.6}$$

B

Matrix representation of states and operators

One of the most common ways to represent states and operators, at least in computational quantum mechanics, is using vectors and matrices. It is crucial to note, however, that we are discussing a *representation* of states and operators; the theory itself is general, and independent of whatever convenient choice of representation we make.

The matrix representation of an operator $\widehat{\mathbf{A}}$ in necessarily dependent of our choice of basis. To illustrate this we look at the matrix representation of the Hamiltonian. It satisfies the time independent Schrödinger equation

$$\widehat{\mathbf{H}} \left|\psi_{E_i}\right\rangle = E_i \left|\psi_{E_i}\right\rangle .$$

Using *spectral decomposition* on $\widehat{\mathbf{H}}$ we get

$$\widehat{\mathbf{H}} = \sum_k E_k \left|\psi_{E_k}\right\rangle \left\langle\psi_{E_k}\right| , \tag{B.1}$$

which by definition of $\widehat{\mathbf{H}}$ is diagonal in the energy eigenstates:

$$H = \begin{pmatrix} E_0 & 0 & 0 & \cdots & 0 \\ 0 & E_1 & 0 & & \vdots \\ 0 & 0 & \ddots & & \\ \vdots & & & & \\ 0 & \cdots & & & E_N \end{pmatrix} . \tag{B.2}$$

However, if we perform a change of basis from $|\psi_{E_i}\rangle$ to an arbitrary complete set of orthogonal states $|\phi_i\rangle$ by using the completeness relation from eq. A.3, we get the following relation

$$
\begin{aligned}
\widehat{\mathbf{H}} &= \sum_k E_k \left|\psi_{E_k}\right\rangle \left\langle\psi_{E_k}\right| \\
&= \sum_k \sum_{i,j} E_k \left|\phi_i\right\rangle \left\langle\phi_i|\psi_{E_k}\right\rangle \left\langle\psi_{E_k}|\phi_j\right\rangle \left\langle\phi_j\right| \\
&= \sum_k \sum_{i,j} \left|\phi_i\right\rangle \left\langle\phi_i\right| \widehat{\mathbf{H}} \left|\psi_{E_k}\right\rangle \left\langle\psi_{E_k}|\phi_j\right\rangle \left\langle\phi_j\right| \\
&= \sum_{i,j} \left|\phi_i\right\rangle \left\langle\phi_i\right| \widehat{\mathbf{H}} \Big[\sum_k \left|\psi_{E_k}\right\rangle \left\langle\psi_{E_k}\right| \Big] \left|\phi_j\right\rangle \left\langle\phi_j\right| \\
&= \sum_{i,j} \left|\phi_i\right\rangle \left\langle\phi_i\right| \widehat{\mathbf{H}} \left|\phi_j\right\rangle \left\langle\phi_j\right| \\
&= \sum_{i,j} \left\langle\phi_i\right| \widehat{\mathbf{H}} \left|\phi_j\right\rangle \left|\phi_i\right\rangle \left\langle\phi_j\right| \\
&= \sum_{i,j} H_{ij} \left|\phi_i\right\rangle \left\langle\phi_j\right| , \qquad \text{(B.3)}
\end{aligned}
$$

which is not diagonal in the new basis. This is usually the starting point when we do physics, since the goal of the computation is to obtain the true eigenstates and eigenvectors of a Hamiltonian. If we choose an initial complete orthonormal basis, we can always set up the matrix and diagonalize it[1].

Doing this basis change, we have also deduced the general form of the matrix elements in a given basis:

$$
A_{ij} = \left\langle\psi_i\right| \widehat{\mathbf{A}} \left|\psi_j\right\rangle , \qquad \text{(B.4)}
$$

$$
A = \begin{pmatrix}
\left\langle\psi_0\right| \widehat{\mathbf{A}} \left|\psi_0\right\rangle & \left\langle\psi_0\right| \widehat{\mathbf{A}} \left|\psi_1\right\rangle & \left\langle\psi_0\right| \widehat{\mathbf{A}} \left|\psi_2\right\rangle & \cdots & \left\langle\psi_0\right| \widehat{\mathbf{A}} \left|\psi_N\right\rangle \\
\left\langle\psi_1\right| \widehat{\mathbf{A}} \left|\psi_0\right\rangle & \left\langle\psi_1\right| \widehat{\mathbf{A}} \left|\psi_1\right\rangle & \left\langle\psi_1\right| \widehat{\mathbf{A}} \left|\psi_2\right\rangle & & \vdots \\
\left\langle\psi_2\right| \widehat{\mathbf{A}} \left|\psi_0\right\rangle & \left\langle\psi_2\right| \widehat{\mathbf{A}} \left|\psi_1\right\rangle & \ddots & & \\
\vdots & & & & \\
\left\langle\psi_N\right| \widehat{\mathbf{A}} \left|\psi_0\right\rangle & \cdots & & & \left\langle\psi_N\right| \widehat{\mathbf{A}} \left|\psi_N\right\rangle
\end{pmatrix} . \qquad \text{(B.5)}
$$

The matrix elements are calculated as integrals, e.g. the expectation value of the energy in an interacting quantum dot using single particle harmonic oscillator wave functions.

[1] The brute force method of doing this (up to a given truncation in the infinite basis) is called *full configuration interaction* (FCI) or *full scale diagonalization*.

```
void DMC::node_comm() {
#ifdef MPI_ON
    if (parallel) {
        MPI_Allreduce(MPI_IN_PLACE, &E, 1, MPI_DOUBLE, MPI_SUM,
            MPI_COMM_WORLD);
        MPI_Allreduce(MPI_IN_PLACE, &samples, 1, MPI_INT, MPI_SUM,
            MPI_COMM_WORLD);

        MPI_Allgather(&n_w, 1, MPI_INT, n_w_list.memptr(), 1, MPI_INT,
            MPI_COMM_WORLD);

        n_w_tot = arma::accu(n_w_list);

    }
#else
    n_w_tot = n_w;
#endif

}

void DMC::switch_souls(int root, int root_id, int dest, int dest_id) {
    if (node == root) {
        original_walkers[root_id]->send_soul(dest);
        n_w--;
    } else if (node == dest) {
        original_walkers[dest_id]->recv_soul(root);
        n_w++;
    }
}

void DMC::normalize_population() {
#ifdef MPI_ON
    using namespace arma;

    if (!(cycle % (n_c / check_thresh) == 0) || cycle > (int) n_c * 0.9)
        return;


    int avg = n_w_tot / n_nodes;
    umat swap_map = zeros<umat>(n_nodes, n_nodes);
    uvec snw = sort_index(n_w_list, 1);


    int root = 0;
    int dest = n_nodes - 1;
    while (root < dest) {
        if (n_w_list(snw(root)) > avg) {
            if (n_w_list(snw(dest)) < avg) {
                swap_map(snw(root), snw(dest))++;
                n_w_list(snw(root))--;
                n_w_list(snw(dest))++;
            } else {
                dest--;
            }
        } else {
            root++;
        }
    }


    uvec test = sum(swap_map, 1);
    if (test.max() < sendcount_thresh) {
        test.clear();
        swap_map.clear();
        return;
    }


    for (int root = 0; root < n_nodes; root++) {
        for (int dest = 0; dest < n_nodes; dest++) {
            if (swap_map(root, dest) != 0) {
                for (int sendcount = 0; sendcount < swap_map(root, dest);
                    sendcount++) {
                    switch_souls(root, n_w - 1, dest, n_w);
                }
            }
        }
    }

    swap_map.clear();
    test.clear();
    MPI_Barrier(MPI_COMM_WORLD);

#endif
```

```
80 }
```

N	ω	E_{VMC}	E_{DMC}	α	β
20	0.01	6.21374	6.14963	0.421499	0.108668
	0.1	30.0811	29.9775	0.673606	0.283105
	0.28	62.0604	61.9265	0.763565	0.417363
	0.5	94.027	93.8745	0.801079	0.533715
	1.0	156.05	155.884	0.83984	0.732855
30	0.01	12.5867	12.505	0.419354	0.131479
	0.1	60.5914	60.4193	0.62963	0.306035
	0.28	124.179	123.968	0.727472	0.446524
	0.5	187.295	187.042	0.76652	0.576088
	1.0	308.859	308.564	0.809796	0.794661

Table B.1: 30 particles lolol

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] ——, *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; http://www.bibsonomy.org/bibtex/2c5c7c9849e177a04c30dd31b0b5615f7/ulger

[4] 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;http://www.bibsonomy.org/bibtex/29ccda94c4c5c89a6ee3ef44d8d10671e/dblp

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

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

[7] 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.

[8] 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

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

[10] 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.

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

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

[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] 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

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

[17] 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.

[18] 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://dblp.uni-trier.de/db/journals/ijcv/ijcv81.html#KleinPSV09;http://dx.doi.org/10.1007/s11263-008-0168-y;http://www.bibsonomy.org/bibtex/2a23447f4a7e318e7fc3166a88075dceb/dblp

[19] 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

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

[21] M. Pedersen Lohne, G. Hagen, M. Hjorth-Jensen, S. Kvaal, and F. Pederiva, "*Ab initio* computation of the energies of circular quantum dots," *Phys. Rev. B*, vol. 84, p. 115302, Sep 2011. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevB.84.115302