Quantum Monte-Carlo Studies of Generalized Many-body Systems

by

Jørgen Høgberget

THESIS
for the degree of
MASTER OF SCIENCE

(Master in Computational Physics)

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

June 2013

Preface

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).

```

};

class Harmonic_osc : public Potential {
protected:
    double w;

public:

    Harmonic_osc(GeneralParams &);

    double get_pot_E(const Walker* walker) const;

};
```

```
double Coulomb::get_pot_E(const Walker* walker) const {

    double e_coulomb = 0;

    for (int i = 0; i < n_p - 1; i++) {
        for (int j = i + 1; j < n_p; j++) {
            e_coulomb += 1 / walker->r_rel(i,j);
        }
    }

    return e_coulomb;
}

double Harmonic_osc::get_pot_E(const Walker* walker) const {

    double e_potential = 0;

    for (int i = 0; i < n_p; i++) {
        e_potential += 0.5 * w * w * walker->get_r_i2(i);
    }

    return e_potential;
}
```

This subclass hierarchy of potentials instructs the system to access objects of type `Potential*`, calling 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:

```
class System {
protected:
    ...

    std::vector<Potential*> potentials;

    ...
};

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

    //Iterate through the potential objects in the potentials list. (*pot) extracts the
        pointer from the iterator.
    for (std::vector<Potential*>::iterator pot = potentials.begin(); pot != potentials.
        end(); ++pot) {
        potE += (*pot)->get_pot_E(walker);
    }

    return potE;
}
```

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 UPDATE REFERENCE

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

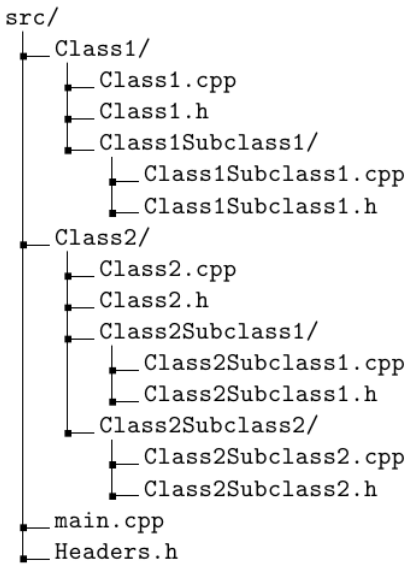

Figure 2.1: An illustration of a standard way to organize source code. The file endings represent C++ code.

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 Class Structures

In scientific programming, we are often in a situation where the simulated process has a physical - or mathematical interpretation. Examples are e.g. particles in molecular dynamics, atoms in bose-einstein condensates, random walkers in diffusion processes, etc. Choosing to create classes representing these quantities will shorten the gap between the mathematical formulation of the problem and the implementation.

In addition we have estimates such as energy, entropy and temperature, all of which are calculated based on equations from statisical mechanics or similar. Having class methods representing these calculations will again shorten the gap. There is no question what is done when the `system.get_potential_energy`

method is called, however, if some random loop appears in the main solver, an initial investigation is required in order to understand the flow of the code.

As described in Section 2.2.2, abstracting e.g. the potential energy function into a system object opens up the possibility of generalizing the code to any potential without altering the main solver. Structure is in other words vital if readability and versatility is desired.

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

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.

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

As will be derived in details in Section 3.1.5, using the Fokker-Planck equation does not violate the original Schrödinger equation, but merely substitute the diffusing value of interest. 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 due to the drift term.

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

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

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].

3.1.5 The connection between anisotropic- and isotropic diffusion models

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 behavior 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

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

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

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.

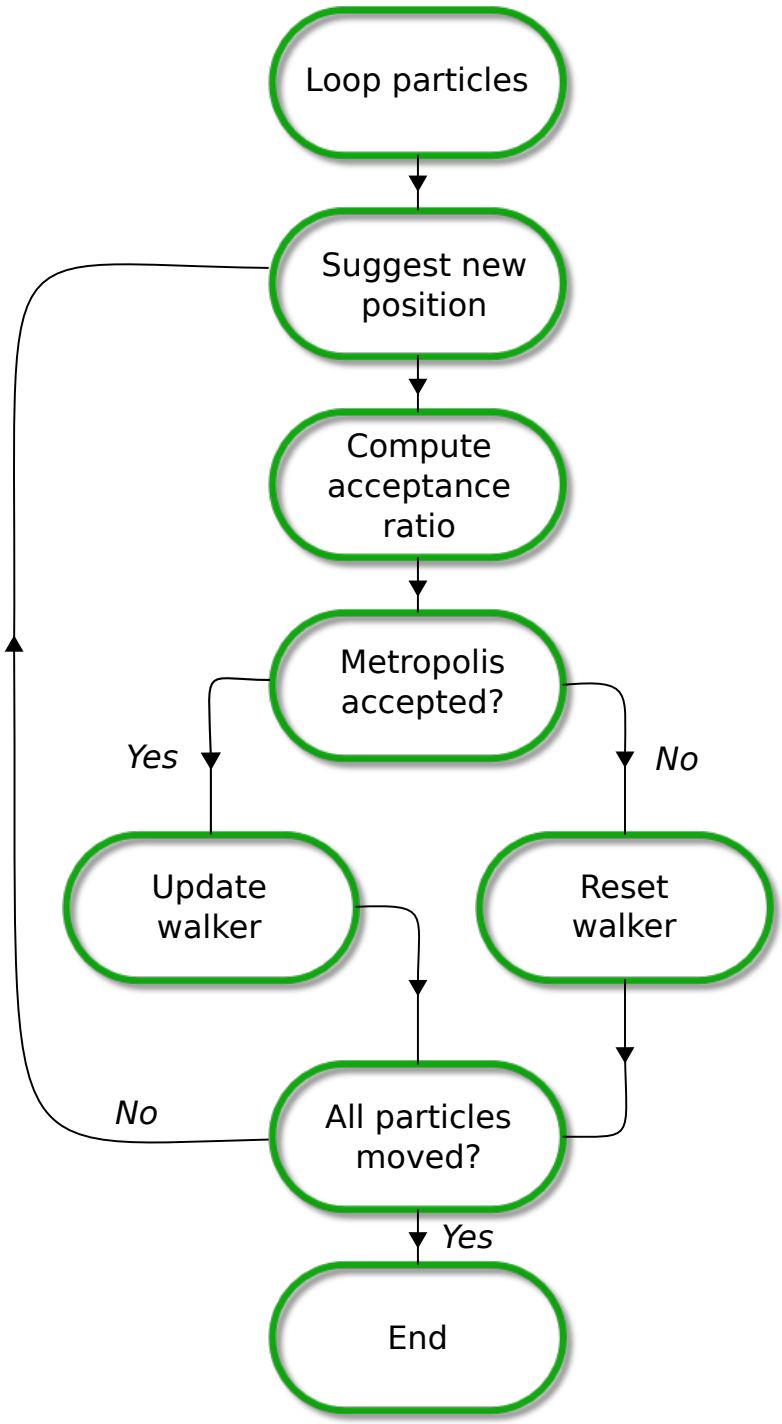

Figure 3.1: Flow chart for iterating a walker through a single time step, i.e. simulation the application of the Green's function from Eq. (3.16) using the Metropolis algorithm. New positions are suggested according to the chosen diffusion model.

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.

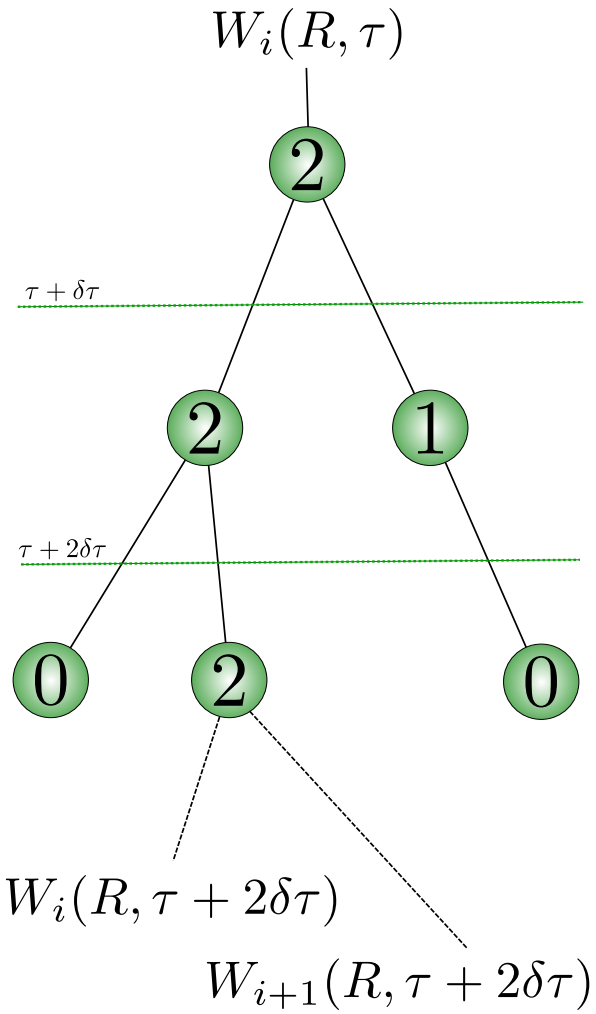

Figure 3.2: Branching illustrated. The initial walker $W_i(R,\tau)$ is branched according to the rules of Section 3.4. The numerical value inside the nodes represents $\overline{G}_B$ from Eq. (3.40). Each horizontal dashed line represent a diffusion step, i.e. a transition in time. Two lines exiting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

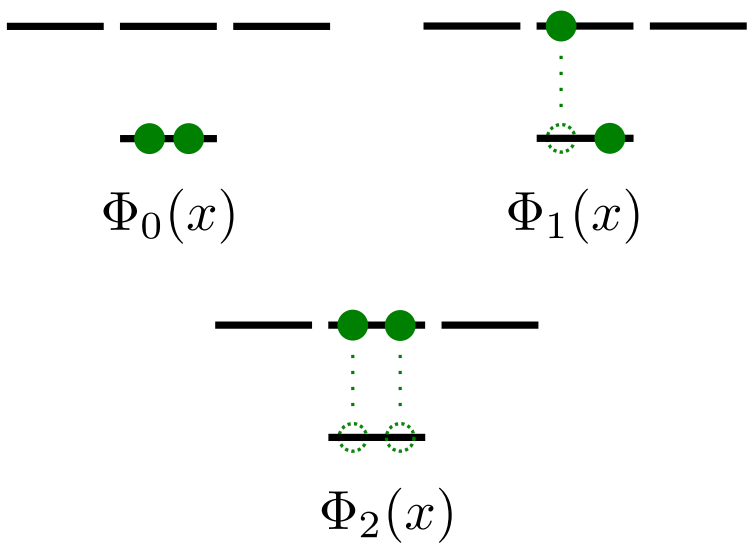

Figure 3.3: Three different electron configurations in an shell structure making up three different $\Phi_k(x)$, i.e. constituents of the many-body basis described in Eq. (3.43). An electron (solid dot) is represented by e.g. the orbital $\phi_{1s}(x_1)$.

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.

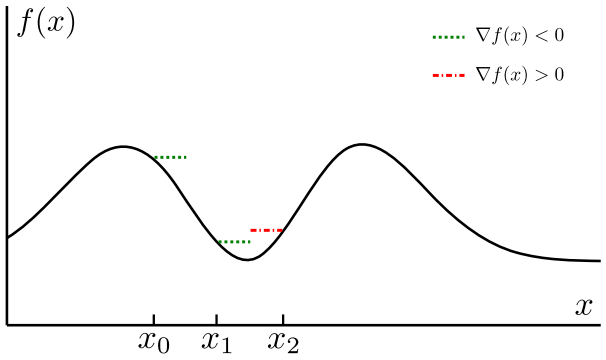

Figure 3.4: Two steps of a one dimensional Gradient Descent process. Steps are taken in the direction of the negative gradient (indicated by dotted lines).

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

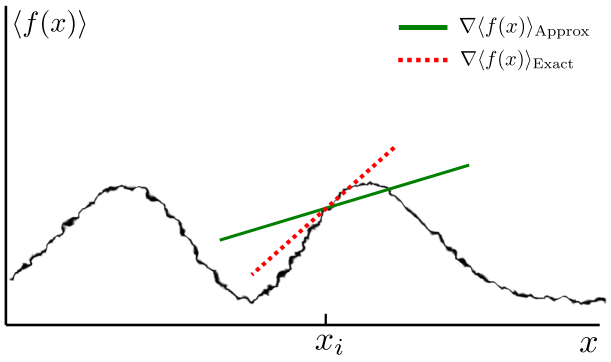

Figure 3.5: A one dimensional plot of an expectation valued function. Smeared lines are representing uncertainties due to rough sampling. The direction of the local gradient (solid green line) at a point x_i is not necessarily a good estimate of the actual analytic gradient (dashed red line).

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.

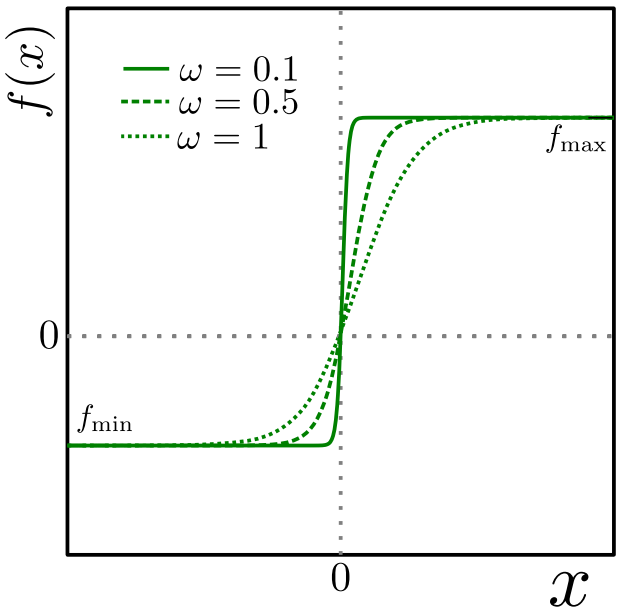

Figure 3.6: Examples of $f(X_i)$ as published in ref. [18]. As $\omega \to 0$, $f(x)$ approaches a step function.

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 $t = 0$.

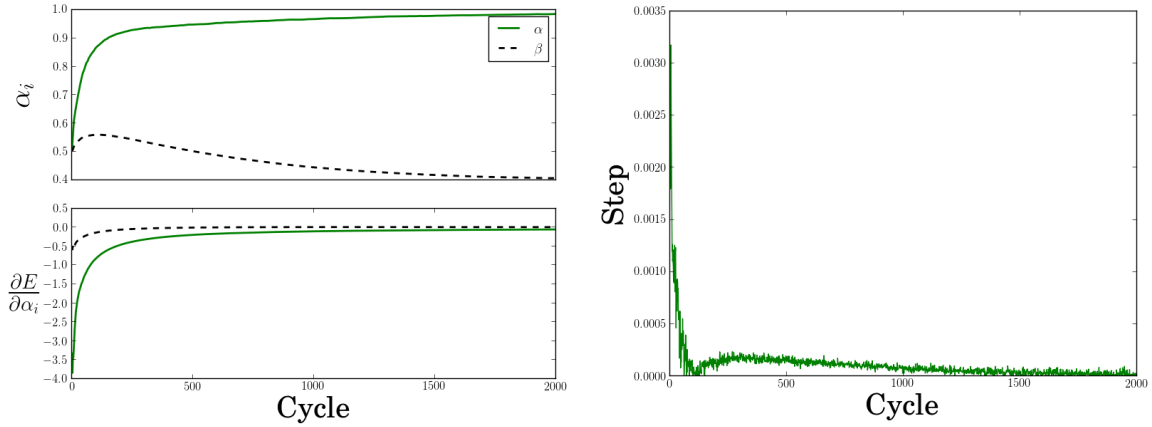

Figure 3.7: Results of Adaptive Stochastic Gradient Descent used on a two-particle Quantum Dot with unit oscillator frequency using 400 cycles pr. gradient sampling and 40 independent walkers. The right figure shows the evolution of the time step. The left figure shows the evolution of the variational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

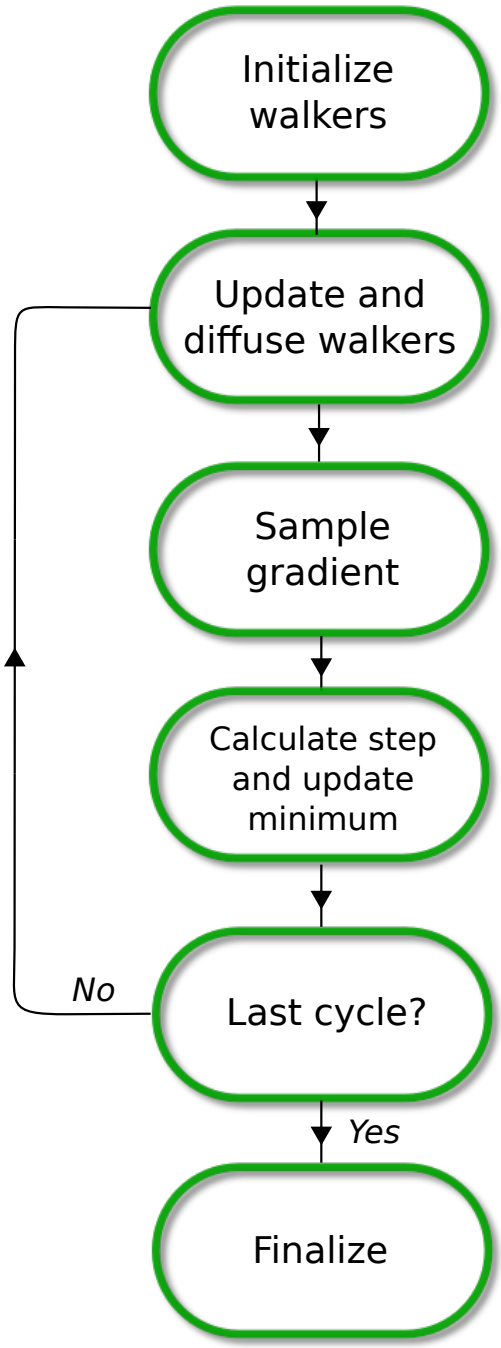

Figure 3.8: Chart flow of ASGD algorithm. Diffusing a walker is done as described in Fig. 3.1. Updating the walkers involves recalculating any values afflicted by updating the minimum. The step is calculated by Eq. (3.70). In case of Quantum Monte-Carlo, the gradient is sampled by Eq. (3.62).

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.

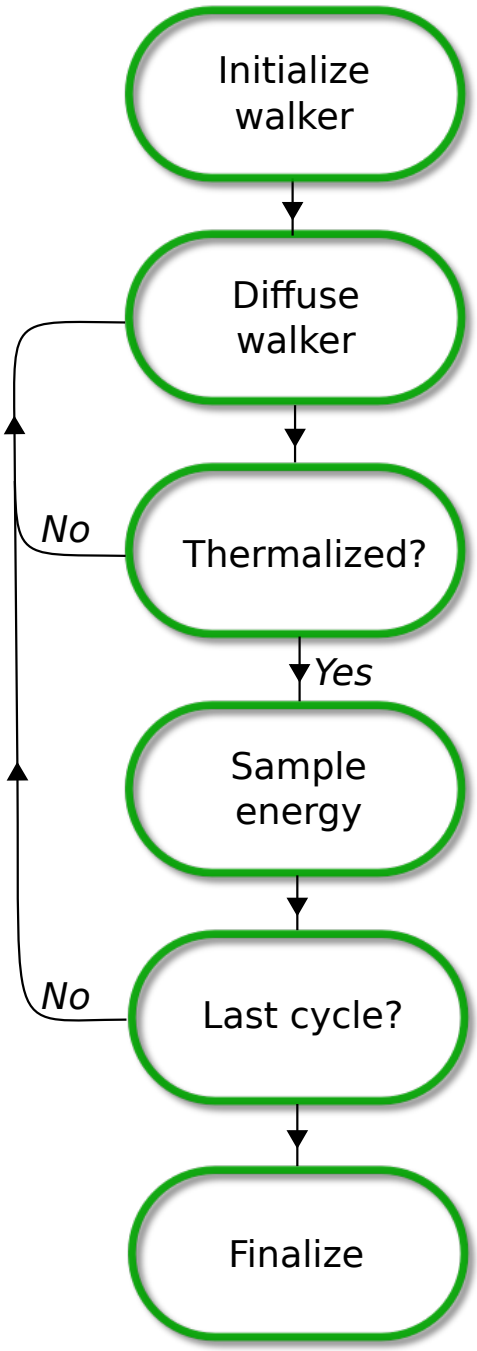

Figure 3.9: Chart flow of the Variational Monte-Carlo algorithm. The second step, *Diffuse Walker*, is the process described in Fig. 3.1. Energies are sampled as described in Section 3.5.3. Thermalization is usually set to a fixed number of cycles.

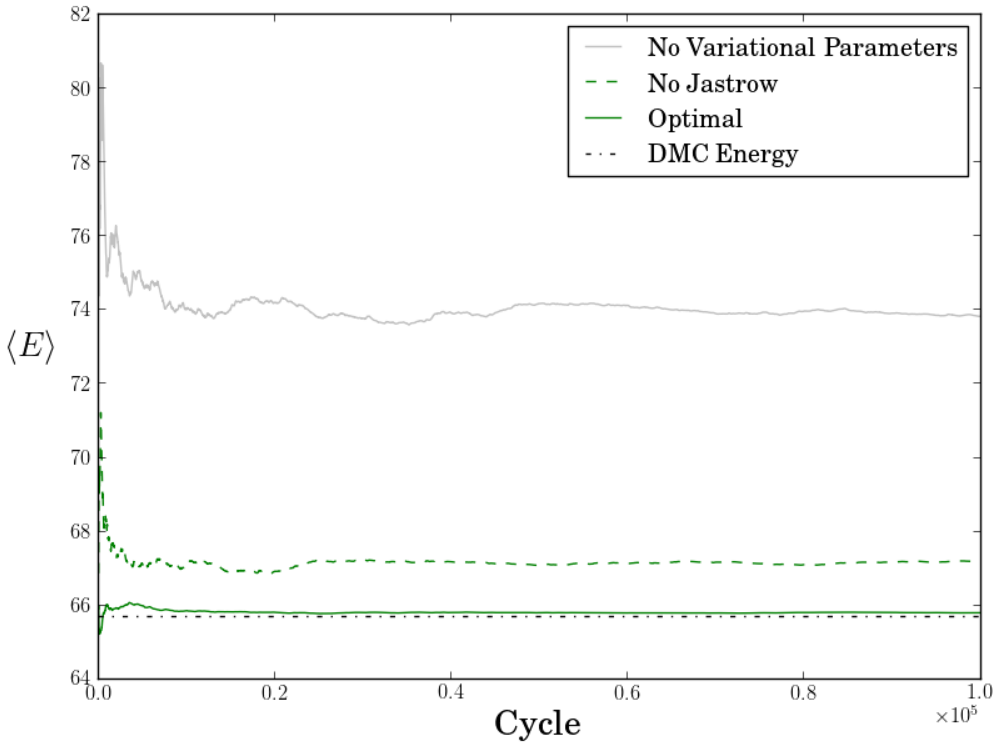

Figure 3.10: Comparison of the VMC energy using different trial wave functions. The DMC energy is believed to be very close to the exact ground state. We see that adding a variational parameter to the trial wave function lowers the energy somewhat, however, when adding the Jastrow factor described in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

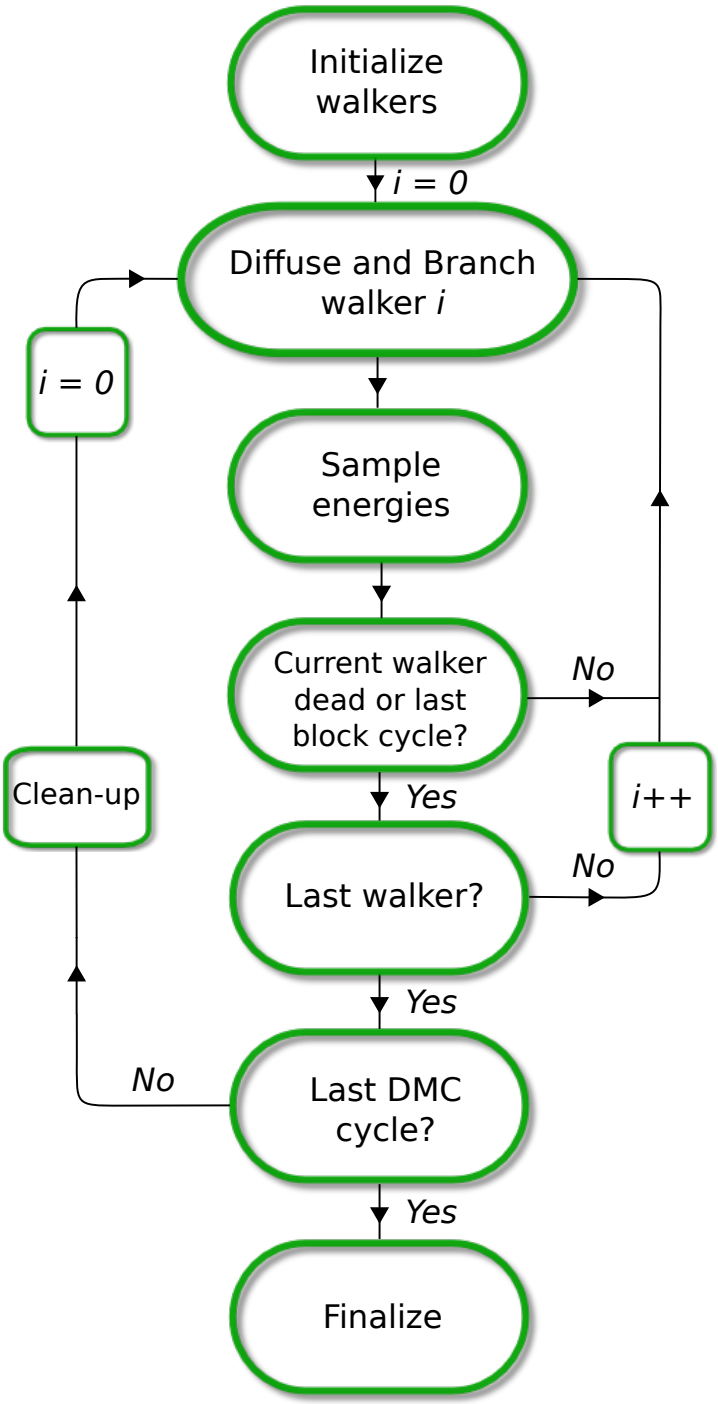

Figure 3.11: Chart flow of the Diffusion Monte-Carlo algorithm. The count variable i is the index of the walker loop. The second step, *Diffuse and Branch Walker*, is the process described in Fig. 3.1 in combination with the branching from Fig. 3.2. Energies are sampled as described in Section 3.8. Thermalization 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)

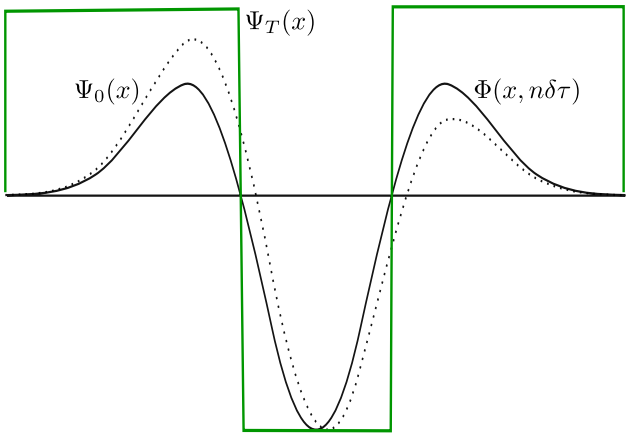

Figure 3.12: An illustration of the fixed node approximation. The dotted line is the exact ground state $\Psi_0(x)$. The distribution of walkers at cycle n, $\Phi(x, n\delta\tau)$, similar in shape with $\Psi_0(x)$, however, sharing nodes with the trial wave function $\Psi_T(x)$ (box-like function for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

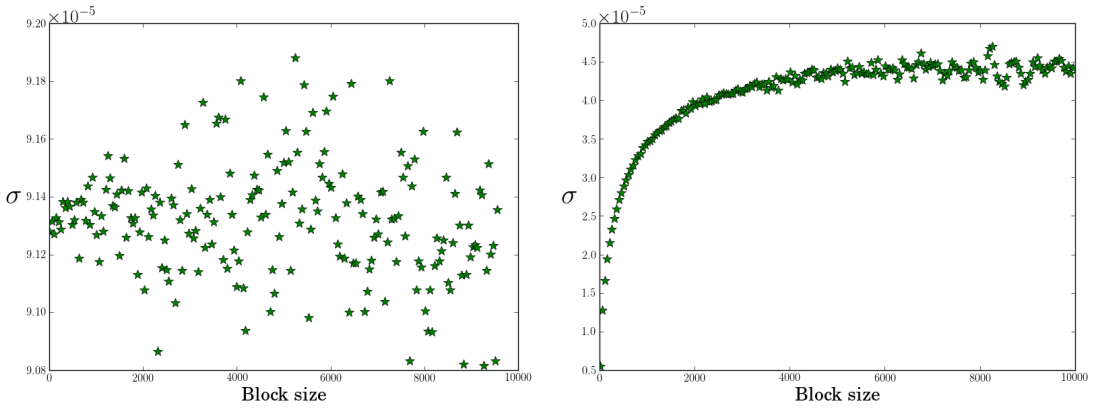

Figure 3.13: Left hand side: Blocking result of (approximately) uncorrelated data generated from a uniform Monte-Carlo integration of $\int_1^2 2x\mathrm{d}x$ resulting in 3.00003 (exact is 3.0). This is in excellent agreement with the magnitude of the error $\sim 9 \cdot 10^{-5}$. On the right hand side we have the blocking result (DMC) for 6-particle $\omega = 0.1$ Quantum Dot, used in Table 5.3. 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.

4

Generalization and Optimization

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

This chapter will cover the underlaying assumptions and goals in Section 4.1, the ..

4.1 Underlying Assumptions and Goals

Thousands of lines of code should be written once and for all[1]. This section will cover the assumptions made and the goals set, preliminary to the coding process.

4.1.1 Assumptions

The code scheme was laid down based on the following assumptions

(i) The constituents of the simulated systems are either pure fermionic or pure bosinic.

(ii) A fermionic system is a two-level system described by one Slater determinant pr. spin eigenvalue[a].

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

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

[a] The actual code is developed to easily add support for n-level systems.

Other assumptions, such as the time-independence of the Hamiltonian, arise due to the QMC solver, not the specific implementation, and are hence covered in Chapter 3.

[1] This, however, is rarely the case. The code used in this thesis has been completely restructured four times.

4.1.2 Generalization Goals

The implementation should be general for:

(i) Fermions and Bosons.

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

(iii) Different minimization algorithms.

(iv) Any Jastrow factor.

(v) Any error estimation algorithm.

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

(vii) Any combination of any potentials.

In addition, the following constraint is set on solvers:

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

The challenge is, despite the vast array of combinations, to preserve simplicity and structure as layers of complexity are added. If-testing inside the solvers in order to achieve generalization are in other words out of the question (see the next section).

4.1.3 Optimization Goals

Designed for the CPU, runtime optimizations are favored over memory optimizations. The following list may appear short, but every step brings immense amounts of complexity to the implementation

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

(ii) Generalization should not be achieved through if-tests in repeated function calls, but rather through polymorphism (see Section 2.2.2).

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

4.2 Specifics Regarding Generalization

Generalization is achieved through the use of deep object orientation, i.e. polymorphism, rather than if-testing. These concepts are described in Section 2.2. The assumptions listed in Section 4.1.1 are applied if not otherwise is stated.

For details regarding the implementation of methods, see the code in [12].

4.2.1 Generalization Goals (i)-(vii)

As discussed in Section 3.5.1, the mathematical difference between fermions and bosons (of importance to QMC) is how the many-body wave functions are constructed from the single-particle basis. In the case of fermions, the expression is given in terms of two Slater determinants, while for bosons, it is simply the product of all states.

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

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

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

    double wf = 1;

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

    return wf;
}
```

Fermion/Boson overloaded pure virtual methods exist for all methods involving evaluation of the many body wave function, e.g. the spatial ratio and the Laplacian sum. When the `QMC` solver asks the `System*` object for a spatial ratio, depending on whether Fermions or Bosons are loaded run-time, the Fermion or Boson spatial ratio is evaluated.

This way of splitting the system class also takes care of optimization goal (ii) in Section 4.1.3 regarding no use of repeating if-tests.

Similar polymorphic splitting is introduced in the following classes:

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

Implementing a new Jastrow Factor done simply by creating a new subclass of `Jastrow`: Simple yet versatile. For more details, see Section 2.2. The splitting done in `QMC` is done to avoid rewriting a lot of general QMC code.

A detailed description of the generalization of potentials (point (vii)) is given in Section 2.2.2.

4.2.2 Generalization Goal (vi) and Expanded bases

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

$$\psi_\gamma^{\text{Exp.}}(r_j) = \sum_{\alpha=0}^{B-1} C_{\gamma\alpha}\phi_\alpha(r_j) \tag{4.1}$$

where B is the size of the expanded basis.

```
1  class ExpandedBasis : public Orbitals {
2
3  ...
4
5  protected:
6
7      int basis_size;
8      arma::mat coeffs;
9      Orbitals* basis;
10
11     //Hartree-Fock
12     void calculate_coefficients();
13
14 };
```

In order to calculate the expansion coefficients, a *Hartree-Fock* solver has been implemented. For a given set of orbitals, e.g. harmonic oscillator, everything that is needed in order to obtain an expanded basis, is to implement expressions for the one-body - and two-body interaction elements.

The implementation of Eq. (4.1) are done by overloading the original `Orbitals.phi` method

```
1  double ExpandedBasis::phi(const Walker* walker, int particle, int q_num) {
2
```

```
    double value = 0;

    //Dividing basis_size by half assuming a two-level system.
    //In case of Bosons, expanding s.p. w.f. does not make sence.
    for (int m = 0; m < basis_size/2; m++) {
        value += coeffs(q_num, m) * basis->phi(walker, particle, m);
    }

    return value;

}
```

See Section **REF HF** for details regarding Hartree-Fock theory.

4.2.3 Generalization Goal (viii)

Functionality for evaluating derivatives numerically is important for two reasons; the first being debugging, the second being the cases where no closed-form expressions for the derivatives can be obtained[2].

As an example, `Orbitals.dell_phi`, which returns a single particle derivative, is virtual, and can be overloaded to call the numerical derivative implementation `Orbitals.num_diff`[3]. The same goes for the Jastrow factor and the variational derivatives in the minimizers. Similar implementations exist for the Laplacian.

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

4.3 Optimizations due to a Single two-level Determinant

Assumption (iii) unlocks the possibility to optimize the expressions involving the Slater-determinant dramatically. Similar optimizations for bosons due to assumption (iv) are considered trivial and will not be covered in detail. See the code in [12] for details regarding bosons.

The full trial wave function Ψ_T is given by Eq. (3.55). I will drop the function arguments to clean up the expressions. Written in terms of a spatial function $|D|$ and a Jastrow function J, we get

$$\Psi_T = |D^{\uparrow}||D^{\downarrow}|J \tag{4.2}$$

The Quantum Force becomes

$$\begin{aligned} F_i &= 2\frac{\nabla_i \left(|D^{\uparrow}||D^{\downarrow}|J\right)}{|D^{\uparrow}||D^{\downarrow}|J} \\ &= 2\left(\frac{\nabla_i|D^{\uparrow}|}{|D^{\uparrow}|} + \frac{\nabla_i|D^{\downarrow}|}{|D^{\downarrow}|} + \frac{\nabla_i J}{J}\right) \\ &= 2\left(\frac{\nabla_i|D^{\alpha}|}{|D^{\alpha}|} + \frac{\nabla_i J}{J}\right) \end{aligned} \tag{4.3}$$

[2] In most cases they can, however, in some cases the expressions are so heavy that they are not practical to use.

[3] An alternative would be to perform the derivative on the full many-body wave function, however, this would demand another layer of polymorphism and make the code less intuitive.

where α is the spin configuration of particle i. The counterpart to α has no dependence of particle i and will hence be zero in the expression above.

Equally for the Laplacians used in the local energy, we get

$$
\begin{aligned}
E_L &= \sum_i \frac{1}{\Psi_T}\nabla_i^2\Psi_T + V \qquad (4.4)\\
\frac{1}{\Psi_T}\nabla_i^2\Psi_T &= \frac{1}{|D^\uparrow||D^\downarrow|J}\nabla_i^2|D^\uparrow||D^\downarrow|J \\
&= \frac{\nabla_i^2|D^\uparrow|}{|D^\uparrow|} + \frac{\nabla_i^2|D^\downarrow|}{|D^\downarrow|} + \frac{\nabla_i^2 J}{J} \\
&\quad + 2\frac{\left(\nabla_i|D^\uparrow|\right)\left(\nabla_i|D^\downarrow|\right)}{|D^\uparrow||D^\downarrow|} + 2\frac{\left(\nabla_i|D^\uparrow|\right)\left(\nabla_i J\right)}{|D^\uparrow|J} + 2\frac{\left(\nabla_i|D^\downarrow|\right)\left(\nabla_i J\right)}{|D^\downarrow|J} \\
&= \frac{\nabla_i^2|D^\alpha|}{|D^\alpha|} + \frac{\nabla_i^2 J}{J} + 2\frac{\nabla_i|D^\alpha|}{|D^\alpha|}\frac{\nabla_i J}{J} \qquad (4.5)
\end{aligned}
$$

Finally, the expression for the R_ψ ratio for Metropolis is

$$
\begin{aligned}
R_\psi &= \frac{\Psi_T^{\text{new}}}{\Psi_T^{\text{old}}} \\
&= \frac{|D^\uparrow|^{\text{new}}|D^\downarrow|^{\text{new}}J^{\text{new}}}{|D^\uparrow|^{\text{old}}|D^\downarrow|^{\text{old}}J^{\text{old}}} \\
&= \frac{|D^\alpha|^{\text{new}}}{|D^\alpha|^{\text{old}}}\frac{J^{\text{new}}}{J^{\text{old}}} \qquad (4.6)
\end{aligned}
$$

where either the spin up or the spin down determinant is unchanged by the step.

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

4.4 Optimizations due to Single-particle Moves

Moving only one particle at the time (see the diffusion algorithm in Fig. 3.1), means changing a single row in the Slater-determinant at the time. Changes to a single row means that almost every *co-factor* remains unchanged. Since all of the expressions deduced in the previous section contains ratios of the spatial wave functions, expressing these determinants in terms of their co-factors should reveal cancellation of terms.

4.4.1 Optimizing the Slater ratios

The inverse of the Slater-matrix is given in terms of its *adjugate* by the following relation [20] (spin-configuration parameter α will be skipped for now)

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

The adjugate of a matrix is the transpose of the cofactor matrix C. Given in terms of matrix element equations, we have

$$D_{ij}^{-1} = \frac{C_{ji}}{|D|} \tag{4.7}$$

$$D_{ji} = \phi_i(r_j) \tag{4.8}$$

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

$$|D| = \sum_i D_{ji} C_{ji}. \tag{4.9}$$

The spatial part of the R_ψ ratio is then (inserting Eq. (4.9) into Eq. (4.6))

$$R_S = \frac{\sum_i D_{ji}^{\text{new}} C_{ji}^{\text{new}}}{\sum_i D_{ji}^{\text{old}} C_{ji}^{\text{old}}} \tag{4.10}$$

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

$$C_{ji}^{\text{new}} = C_{ji}^{\text{old}} = (D_{ij}^{\text{old}})^{-1} |D^{\text{old}}|, \tag{4.11}$$

where the inverse relation of Eq. (4.7) has been used. Inserting this into Eq. (4.10) yields

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

The diagonal element of the identity matrix is by definition unity. Inserting this fact combined with the relation from Eq. (4.8) yields the optimized expression for the ratio

$$R_S = \sum_i \phi_i(r_j^{\text{new}}) (D_{ij}^{\text{old}})^{-1} \tag{4.12}$$

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

Similar reductions can be applied to all the Slater ratio expressions from the previous section, see refs. [7, 11]:

$$\frac{\nabla_i |D|}{|D|} = \sum_k \nabla_i \phi_k(r_i^{\text{new}}) (D_{ki}^{\text{new}})^{-1} \tag{4.13}$$

$$\frac{\nabla_i^2 |D|}{|D|} = \sum_k \nabla_i^2 \phi_k(r_i^{\text{new}}) (D_{ki}^{\text{new}})^{-1} \tag{4.14}$$

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

Closed form expressions for the derivatives and Laplacians of the single particle may be implemented and accessed when calling these functions to avoid expensive numerical calculations. See Appendices **REF SYMPYGEN** for a tabulation of closed form expressions.

4.4.2 Optimizing the Inverse

One might question the efficiency of calculating inverse matrices compared to brute force estimation of the determinants. However, as for the ratio in Eq. (4.12), using co-factor expansions we can construct an updating algorithm which dramatically decreases the cost of calculating the inverse of the new Slater matrix.

Let i be the currently moved particle, the new inverse is given in terms of the old by the following expression [7, 11]

$$\tilde{I}_{ij} = \sum_l D_{il}^{\text{new}} (D_{lj}^{\text{old}})^{-1} \tag{4.15}$$

$$(D_{kj}^{\text{new}})^{-1} = (D_{kj}^{\text{old}})^{-1} - \frac{1}{R_S} (D_{ji}^{\text{old}})^{-1} \tilde{I}_{ij} \qquad j \neq i \tag{4.16}$$

$$(D_{ki}^{\text{new}})^{-1} = \frac{1}{R_S} (D_{ki}^{\text{old}})^{-1} \qquad \text{else} \tag{4.17}$$

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

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

4.4.3 Optimizing the Padé Jastrow factor Ratio

As for the Green's function ratio is Eq. (3.37), the ratio between two Jastrow factors are best calculating as exponentiation the logarithm

$$\log \frac{J^{\text{new}}}{J^{\text{old}}} = \sum_{k<j=1}^{N} \frac{a_{kj} r_{kj}^{\text{new}}}{1 + \beta r_{kj}^{\text{new}}} - \frac{a_{kj} r_{kj}^{\text{old}}}{1 + \beta r_{kj}^{\text{old}}} \tag{4.18}$$

$$\equiv \sum_{k<j=1}^{N} g_{kj}^{\text{new}} - g_{kj}^{\text{old}} \tag{4.19}$$

The relative distances r_{kj} behave much like the cofactors in Section 4.4.1; changing r_i only changes r_{ij}, in other words

$$r_{kj}^{\text{new}} = r_{kj}^{\text{old}} \qquad k \neq i \tag{4.20}$$

which inserted into Eq. (4.19) yields

$$\begin{aligned}\log \frac{J^{\text{new}}}{J^{\text{old}}} &= \sum_{k<j\neq i} g_{kj}^{\text{old}} - g_{kj}^{\text{old}} + \sum_{j=1}^{N} g_{ij}^{\text{new}} - g_{ij}^{\text{old}} \\ &= \sum_{j=1}^{N} a_{ij} \left(\frac{r_{ij}^{\text{new}}}{1+\beta r_{ij}^{\text{new}}} - \frac{r_{ij}^{\text{old}}}{1+\beta r_{ij}^{\text{old}}} \right)\end{aligned} \tag{4.21}$$

Exponentiating both sides reveals the final optimized ratio

$$\frac{J^{\text{new}}}{J^{\text{old}}} = \exp\left[\sum_{j=1}^{N} a_{ij} \left(\frac{r_{ij}^{\text{new}}}{1+\beta r_{ij}^{\text{new}}} - \frac{r_{ij}^{\text{old}}}{1+\beta r_{ij}^{\text{old}}} \right) \right] \tag{4.22}$$

4.5 Optimizing the Padé Jastrow Derivative Ratios

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

4.5.1 The Gradient

Using the notation of Eq. (4.19), the x component of the Padé Jastrow gradient ratio for particle i then becomes

$$\frac{1}{J}\frac{\partial J}{\partial x_i} = \frac{1}{\prod_{k<l} \exp g_{kl}} \frac{\partial}{\partial x_i} \prod_{k<l} \exp g_{kl} \tag{4.23}$$

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

$$\begin{aligned}\frac{1}{J}\frac{\partial J}{\partial x_i} &= \sum_{k\neq i} \frac{1}{\exp g_{ik}} \frac{\partial}{\partial x_i} \exp g_{ik} \\ &= \sum_{k\neq i} \frac{1}{\exp g_{ik}} \exp g_{ik} \frac{\partial g_{ik}}{\partial x_i} \\ &= \sum_{k\neq i} \frac{\partial g_{ik}}{\partial x_i} \\ &= \sum_{k\neq i} \frac{\partial g_{ik}}{\partial r_{ik}} \frac{\partial r_{ik}}{\partial x_i}\end{aligned} \tag{4.24}$$

$$\begin{aligned}\frac{\partial g_{ik}}{\partial r_{ik}} &= \frac{\partial}{\partial r_{ik}}\left(\frac{a_{ik}r_{ik}}{1+\beta r_{ik}}\right) \\ &= \frac{a_{ik}}{1+\beta r_{ik}} - \frac{a_{ik}r_{ik}}{(1+\beta r_{ik})^2}\beta \\ &= \frac{a_{ik}(1+\beta r_{ik}) - a_{ik}\beta r_{ik}}{(1+\beta r_{ik})^2} \\ &= \frac{a_{ik}}{(1+\beta r_{ik})^2}\end{aligned} \tag{4.25}$$

$$\begin{aligned}\frac{\partial r_{ik}}{\partial x_i} &= \frac{\partial}{\partial x_i}\sqrt{(x_i-x_k)^2+(y_i-y_k)^2+(z_i-z_k)^2} \\ &= \frac{1}{2}2(x_i-x_k)/\sqrt{(x_i-x_k)^2+(y_i-y_k)^2+(z_i-z_k)^2} \\ &= \frac{x_i-x_k}{r_{ik}}\end{aligned} \tag{4.26}$$

Combining these expressions yields

$$\frac{1}{J}\frac{\partial J}{\partial x_i} = \sum_{k\neq i}\frac{a_{ik}}{r_{ik}}\frac{x_i-x_k}{(1+\beta r_{ik})^2} \tag{4.27}$$

When changing Cartesian variable in the differentiation, the only change to the expression is that the corresponding Cartesian variable changes in the numerator of Eq. (4.27). In other words, generalizing to the full gradient is done by substituting the Cartesian difference with the position vector difference.

$$\frac{\nabla_i J}{J} = \sum_{k\neq i=1}^{N}\frac{a_{ik}}{r_{ik}}\frac{\vec{r}_i-\vec{r}_k}{(1+\beta r_{ik})^2} \tag{4.28}$$

4.5.2 The Laplacian

The method of deducing the closed form expression for the Laplacian of the Padé Jastrow factor is identical to that of the gradient. The full calculation is done in ref. [11]. The expression becomes

$$\frac{\nabla_i^2 J}{J} = \left|\frac{\nabla_i J}{J}\right|^2 + \sum_{k\neq i=1}^{N}\left(\frac{d-1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}} + \frac{\partial^2 g_{ik}}{\partial r_{ik}^2}\right) \tag{4.29}$$

where d is the number of dimensions, arising due to the fact that the Laplacian, unlike the gradient, is a summation of contributions from all dimensions. A simple differentiation of Eq. (4.25) with respect to r_{ik} yields

$$\frac{\partial^2 g_{ik}}{\partial r_{ik}^2} = -\frac{2a_{ik}\beta}{(1+\beta r_{ik})^3} \tag{4.30}$$

Inserting Eq. (4.25) and Eq. (4.30) into Eq. (4.29) yields

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

which with a little cleanup results in

$$\frac{\nabla_i^2 J}{J} = \left|\frac{\nabla_i J}{J}\right|^2 + \sum_{k\neq i} a_{ik}\frac{(d-1)+(d-3)\beta r_{ik}}{r_{ik}(1+\beta r_{ik})^3} \tag{4.31}$$

The local energy calculation needs the sum of the Laplacians for all particles (see Eq. (4.4)). In other words, the quantity of interest becomes

$$\sum_i \frac{\nabla_i^2 J}{J} = \sum_i \left|\frac{\nabla_i J}{J}\right|^2 + \sum_{i\neq k}^{N} a_{ik}\frac{(d-1)+(d-3)\beta r_{ik}}{r_{ik}(1+\beta r_{ik})^3} \tag{4.32}$$

However, due to the symmetry of r_{ik}, the second term count equal values twice. We can therefore optimize the calculation by only summing for $k > i$, and rather multiply the sum by two. Bringing it all together, we get

$$\sum_i \frac{\nabla_i^2 J}{J} = \sum_i \left|\frac{\nabla_i J}{J}\right|^2 + 2\sum_{i<k}^{N} a_{ik}\frac{(d-1)+(d-3)\beta r_{ik}}{r_{ik}(1+\beta r_{ik})^3} \tag{4.33}$$

4.6 Tabulating Recalculated Data

The optimizations covered in this section will exclusively arise from point (i) in section 4.1.3. Avoiding recalculating expressions also include exploiting, such as was done in the Padé Jastrow Laplacian in Eq. (4.33).

rrel, r2, phimatrix, delphi, oppdatere jastrow gradient, qnum indie terms ++ a s a s d a d sd f s f s a sa dsa sda a a ads ad gf asrfs asdf as fas a asd asd as

4.6.1 The relative distance matrix

In the discussions regarding the optimization of the Jastrow ratio, it became clear that moving one particle only changed N of the relative distances. Storing these values in a matrix, consequently updating the values corresponding to the moved particle, ensures that these values are calculated once and for all.

$$r_{rel} = r_{rel}^T = \begin{pmatrix} 0 & r_{12} & r_{13} & \cdots & r_{1N} \\ & 0 & r_{23} & \cdots & r_{2N} \\ & & \ddots & \ddots & \vdots \\ & \cdots & & 0 & r_{(N-1)N} \\ & & & & 0 \end{pmatrix}. \tag{4.34}$$

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

    ...

    //Updating the part of the r_rel matrix which is changed by moving the [particle]
    for (int j = 0; j < n_p; j++) {
        if (j != particle) {
            walker_post->r_rel(particle, j) = walker_post->r_rel(j, particle)
                    = walker_post->calc_r_rel(particle, j);
        }
    }

    ...

}
```

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

Similar storage has been done for the squared distance vector.

4.6.2 The Slater related matrices

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

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

$$D \equiv \left[D^{\uparrow}\, D^{\downarrow}\right] = \begin{bmatrix} \phi_1(r_0) & \phi_1(r_1) & \cdots & \phi_1(r_N) \\ \phi_2(r_0) & \phi_2(r_1) & \cdots & \phi_2(r_N) \\ \vdots & \vdots & & \vdots \\ \phi_{N/2}(r_0) & \phi_{N/2}(r_1) & \cdots & \phi_{N/2}(r_N) \end{bmatrix} \tag{4.35}$$

$$\nabla D \equiv \begin{bmatrix} \nabla\phi_1(r_0) & \nabla\phi_1(r_1) & \cdots & \nabla\phi_1(r_N) \\ \nabla\phi_2(r_0) & \nabla\phi_2(r_1) & \cdots & \nabla\phi_2(r_N) \\ \vdots & \vdots & & \vdots \\ \nabla\phi_{N/2}(r_0) & \nabla\phi_{N/2}(r_1) & \cdots & \nabla\phi_{N/2}(r_N) \end{bmatrix} \tag{4.36}$$

4.6.3 Avoiding spin tests

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

$$\begin{array}{ll} i < N/2 & D^{\uparrow}(i) \\ i \geq N/2 & D^{\downarrow}(i - N/2) \end{array} \tag{4.37}$$

However, simply concatenating the Slater related matrices solves the entire problem. This has already been done in Eq. (4.35) and Eq. (4.36). Applying this to the inverse yields

$$D^{-1} \equiv \left[(D^{\uparrow})^{-1} (D^{\downarrow})^{-1} \right] \tag{4.38}$$

Leaving no if-tests required in order to sort the spin splitting. In e.g. the updating algorithm for the inverse, we must now simply keep track of which part we need to update by selecting a start and stop criteria, the rest of the code is general.

4.6.4 The Padé Jastrow gradient

Consider Eq. (4.28). Defining

$$\mathrm{d}\vec{J}_{ik} = -\mathrm{d}\vec{J}_{ki} \equiv \frac{a_{ik}}{r_{ik}} \frac{\vec{r}_i - \vec{r}_k}{(1+\beta r_{ik})^2}, \tag{4.39}$$

the gradient can be written in a more compact form

$$\frac{\nabla_i J}{J} = \sum_{k \neq i=1}^{N} \mathrm{d}\vec{J}_{ik}. \tag{4.40}$$

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

$$\mathrm{d}\vec{J} = -\mathrm{d}\vec{J}^T \equiv \begin{pmatrix} 0 & \mathrm{d}\vec{J}_{12} & \mathrm{d}\vec{J}_{13} & \cdots & \mathrm{d}\vec{J}_{1N} \\ & 0 & \mathrm{d}\vec{J}_{23} & \cdots & \mathrm{d}\vec{J}_{2N} \\ & & \ddots & \ddots & \vdots \\ & (-) & & 0 & \mathrm{d}\vec{J}_{(N-1)N} \\ & & & & 0 \end{pmatrix}. \tag{4.41}$$

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

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

Calculating the gradient is now only a matter of summing the rows of the matrix in Eq. (4.41). On a more careful notice, we can further optimize the calculation of the gradient by taking into account that we have access to the previous gradient

$$\frac{\nabla_i J^{\mathrm{old}}}{J^{\mathrm{old}}} = \sum_{k \neq i=1}^{N} \mathrm{d}\vec{J}^{\mathrm{old}}_{ik} \tag{4.42}$$

$$\frac{\nabla_i J^{\mathrm{new}}}{J^{\mathrm{new}}} = \sum_{k \neq i=1}^{N} \mathrm{d}\vec{J}^{\mathrm{new}}_{ik} \tag{4.43}$$

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

$$\frac{\nabla_{i\neq p} J^{\text{new}}}{J^{\text{new}}} = \sum_{k\neq i\neq p} \mathrm{d}\vec{J}_{ik}^{\text{old}} + \mathrm{d}\vec{J}_{ip}^{\text{new}} \tag{4.44}$$

$$= \left[\sum_{k\neq i\neq p} \mathrm{d}\vec{J}_{ik}^{\text{old}} + \mathrm{d}\vec{J}_{ip}^{\text{old}}\right] - \mathrm{d}\vec{J}_{ip}^{\text{old}} + \mathrm{d}\vec{J}_{ip}^{\text{new}} \tag{4.45}$$

$$= \frac{\nabla_i J^{\text{old}}}{J^{\text{old}}} - \mathrm{d}\vec{J}_{ip}^{\text{old}} + \mathrm{d}\vec{J}_{ip}^{\text{new}} \tag{4.46}$$

reducing the calculation to three flops. For the case $i = p$, the entire sum as in Eq. (4.43) must be calculated.

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

    //i < p
    for (int i = 0; i < p; i++) {
        for (int k = 0; k < dim; k++) {
            walker_post->jast_grad(i, k) = walker_pre->jast_grad(i, k)
                    + walker_post->dJ(i, p, k) - walker_pre->dJ(i, p, k);
        }
    }

    //i = p
    for (int k = 0; k < dim; k++) {

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

        for (int j = p + 1; j < n_p; j++) {
            sum += walker_post->dJ(p, j, k);
        }

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

    }

    //i > p
    for (int i = p + 1; i < n_p; i++) {
        for (int k = 0; k < dim; k++) {
            walker_post->jast_grad(i, k) = walker_pre->jast_grad(i, k)
                    + walker_post->dJ(i, p, k) - walker_pre->dJ(i, p, k);
        }
    }

}
```

This optimization scales very well with increasing number of particles.

4.6.5 The single-particle Wave Functions

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

For systems of many particles, the function call `Orbitals.phi(walker, i, qnum)` needs to figure out which expression is related to which quantum number. The brute force implementation is to simply if-test on the quantum number, and calculate the corresponding expression inside the correct if-test.

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

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

    ...

}
```

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

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

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

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

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

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

}
```

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

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

with no if-tests required.

Other more specific optimizations can be implemented, so called *quantum number independent factors*, $\overline{Q}_i$. When moving particle p, as discussed previously, a single column in the Slater matrix from Eq. (4.35) needs to be updated. All these terms are calculated in the same position. This implies that terms independent of the quantum number will be equal. Looking at the single particle states for harmonic oscillator and hydrogen listed in the Appendix, their exponential form results in that an exponential factor independent of the quantum number is present in all terms.

$$\overline{Q}_i^{\text{H.O.}} = e^{-\frac{1}{2}\alpha\omega r_i^2} \tag{4.47}$$

$$\overline{Q}_i^{\text{Hyd.}} = e^{-\frac{1}{n}\alpha Z r_i} \tag{4.48}$$

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

The implementation is rather simple. Upon moving a particle, the virtual function

`Orbitals.set_qnum_indie_terms` is called, updating the value of e.g. an `exp_factor` pointer shared by all the loaded `BasisFunctions` objects and the `Orbitals` class.

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

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

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

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

}
```

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

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

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

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

    H = y;
    return H*(*exp_factor);

}

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

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

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

}
```

This optimization is an enormous speedup for many particle simulations.

All single particle expressions given are calculated using *SymPy*. See Appendix **REF SYMPY** for details. For Quantum Dots, 84 `BasisFunctions` objects are needed for a 42-particle simulation, reducing the number of exponential calls with 83 pr. particle pr. walker pr. cycle, which for an average DMC calculation results in $24 \cdot 10^9$ saved exponential calls.

4.7 CPU Cache Optimization

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

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

The memory is sent to the cache as arrays, which means that storing walker data sequentially in memory is the way to go in order to make take full use of the processor cache. This is ensured by not using

pointers within the walker objects, as pointers are free to be declared in any memory address.

4.7.1 Disadvantage of Generalizing the code

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

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

4.7.2 Consequences

The QMC code has support for both kinds of scenarios. Compiling the source code with a main file which fixes the system variables will result in a more efficient executable. For the purpose of this thesis, this was not done; all system variables are controlled via a configuration script (see `runQMC.py` in ref. [12]).

Part II

Results

5

Results

5.1 Optimization Results

The optimizations listed in this section has been estimated using a 30 particle Quantum Dot system with a Padé Jastrow Factor.

Profiling the code revealed that, not surprisingly, 99% of the runtime was spent in the `QMC.diffuse_walker` function for both VMC, DMC and ASGD. Optimizing the code then solely involved optimizing this function.

The profiling tool of choice was KCacheGrind, aviable at the Ubuntu Software Center. KCacheGrind lists relative time spent in functions graphically in blocks, whose size is proportional to the time spent inside the function, much like standard hard drive listing software does with files and file size.

Prior to the listed optimizations, all other optimizations mentioned in previous sections has been implemented. The reason why they are not explicitly listed is the fact that they were implemented from start, and are considered "standard optimizations". Writing a program without them was considered pointless.

5.1.1 Storing the Slater Matrix

This optimization is described in detail in Section 4.6.2. In addition to storing the slater, the calculation of $\tilde{I}$ from the Slater inverse updating algorithm in Eq. (4.16) was pre calculated outside the main loops.

The percentages listed in the following table is the total time spent inside this specific function relative to all other functions.

`Orbitals.phi`	
Relative time spent prior to optimization	80.88%
Relative time spent after optimization	8.2%
Relative speedup	9.86

The speedup is not because of optimizations within the function itself, but rather due to far less calls to the function. If the calculation of $\tilde{I}$ was done outside the main loops in the first place, the speedup would be far less.

5.1.2 Optimized Jastrow Gradient

The optimization described in this Section is discussed in detail in Section 4.6.4.

The percentages listed in the following table is the total time spent inside this specific function relative to all other functions.

`Jastrow.get_grad` + `Jastrow.calc_dJ`	
Relative time spent prior to optimization	40%
Relative time spent after optimization	5.24%
Relative speedup	7.63

Exploiting the symmetries of the Padé Jastrow Gradient in addition to calculating the new gradient based on the old is in other words extremely efficient. Keep in mind however, that these results are for a high number of particles. For e.g. two particles, this optimization would not matter at all.

5.1.3 Storing the Orbital Derivatives

This optimization is covered in detail in Section 4.6.2. Much like for the Slater matrix, the optimization in this case comes from the fact that the function itself is called fewer times, rather than being optimized.

The percentages listed in the following table is the total time spent inside this specific function relative to all other functions.

`Orbitals.dell_phi`	
Relative time spent prior to optimization	56.27%
Relative time spent after optimization	7.31%
Relative speedup	7.70

5.1.4 Storing the Quantum Number Independent Terms

This optimization is covered in detail in Section 4.6.5. This optimization lowers the number of exponential function calls, and hence optimizes the calculation time of single particle states, its gradients and Laplacians.

The percentages listed in the following table is the total time spent inside this specific function relative to all other functions.

`Jastrow.get_grad` + `Jastrow.calc_dJ`	
Relative time spent prior to optimization	5.85%
Relative time spent after optimization	0.13%
Relative speedup	45

This result is not surprisingly equal to 15 quantum numbers (for 30 particles) times three. One from the orbitals, and two from their gradients. Prior to this optimization, 45 exponential calls was needed to fill a row in the Slater matrix and the derivative matrix; this has been reduced to one.

5.1.5 Overall Optimization

Combining all the optimizations listed in this chapter, the final runtime was reduced to 5% of the original. The final scaling was

5.2 Validating the code

5.2.1 Calculation for non-interacting particles

5.3 QDOTS RESULTS

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]

[2]

5.4 FIXED NODE TESTS

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

Auto-generation with SymPy

"*SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible. SymPy is written entirely in Python and does not require any external libraries.*" - The SymPy Home Page

The focus on this appendix will be on using SymPy to calculate closed form expressions for single particle wave functions needed to optimize the calculations of the Slater gradient and Laplacian. For systems of many particles, it is crucial to optimize to have these expressions in order for the code to remain efficient.

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

B.1 Usage

SymPy is, as described in the introductory quote, designed to be simple to use. This section will cover the basics needed to calculate gradients and Laplacians, auto-generating C++ - and Latex code.

B.1.1 Doing Symbolic Algebra

In order for SymPy to recognize e.g. `x` as a symbol, that is, a *mathematical variable*. In contrast to programming variables, symbols are not initialized to a value. Initializing symbols can be done in several ways, the two most common are listed below

```
In [1]: from sympy import Symbol, symbols

In [2]: x = Symbol('x')

In [3]: y, z = symbols('y z')

In [7]: x*x+y
Out[7]: 'x**2 + y'
```

The `Symbol` function handles single symbols, while `symbols` can initialize several symbols simultaneously. The string argument might seem redundant, however, this represents the *label* displayed using print functions. In addition, key word arguments can be sent to the symbol functions, flagging variables as e.g. positive, real, etc.

```
In [1]: from sympy import Symbol, symbols, im

In [2]: x2 = Symbol('x^2', real=True, positive=True) #Flagged as real. Note the label.

In [3]: y, z = symbols('y z') #Not flagged as real

In [7]: x2+y #x2 is printed more nicely given a describing label
Out[7]: 'x^2 + y'

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

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

B.1.2 Exporting C++ and Latex Code

Exporting code is extremely simple. Consider the following example

```
In [1]: from sympy import symbols, printing, exp

In [2]: x, x2 = symbols('x x^2')

In [3]: printing.ccode(x*x*x*x*exp(-x2*x))
Out[3]: 'pow(x, 4)*exp(-x*x^2)'

In [4]: printing.ccode(x*x*x*x)
Out[4]: 'pow(x, 4)'

In [5]: print printing.latex(x*x*x*x*exp(-x2))
\frac{x^{4}}{e^{x^{2}}}
```

The following expression is the direct output from line five compiled in Latex

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

B.1.3 Calculating Derivatives

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

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

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

Calculating the gradients and Laplacian is very simply by using the `diff` function

```
In [1]: from sympy import symbols, diff, exp, sqrt

In [2]: x, y, z, Z = symbols('x y z Z')

In [3]: r = sqrt(x*x + y*y + z*z)

```

```
In [4]: r
Out[4]: '(x**2 + y**2 + z**2)**(1/2)'

In [5]: phi = (Z*r - 2)*exp(-Z*r/2)

In [6]: phi
Out[6]: '(Z*(x**2 + y**2 + z**2)**(1/2) - 2)*exp(-Z*(x**2 + y**2 + z**2)**(1/2)/2)'

In [7]: diff(phi, x)
Out[7]: '-Z*x*(Z*(x**2 + y**2 + z**2)**(1/2) - 2)*exp(-Z*(x**2 + y**2 + z**2)**(1/2)/2)
    /(2*(x**2 + y**2 + z**2)**(1/2)) + Z*x*exp(-Z*(x**2 + y**2 + z**2)**(1/2)/2)/(x**2 +
    y**2 + z**2)**(1/2)'
```

Now, this looks like a nightmare. However, SymPy has great support for simplifying expressions. The following code demonstrated this quite nicely

```
...

In [6]: phi
Out[6]: '(Z*(x**2 + y**2 + z**2)**(1/2) - 2)*exp(-Z*(x**2 + y**2 + z**2)**(1/2)/2)'

In [7]: from sympy import factor, Symbol, printing

In [8]: R = Symbol('r') #Creates a symbolic equivalent of the mathematical r

In [9]: diff(phi, x).factor() #Gathers all the terms nicely
Out[9]: '-Z*x*(Z*(x**2 + y**2 + z**2)**(1/2) - 4)*exp(-Z*(x**2 + y**2 + z**2)**(1/2)/2)
    /(2*(x**2 + y**2 + z**2)**(1/2))'

In [10]: diff(phi, x).factor().subs(r, R) #replaces (x^2 + y^2 + z^2)^(1/2) with r
Out[10]: '-Z*x*(Z*r - 4)*exp(-Z*r/2)/(2*r)'

In [11]: print printing.latex(diff(phi, x).factor().subs(r, R))
- \frac{Z x \left(Z r -4\right)}{2 r e^{\frac{1}{2} Z r}}
```

This version of the expression is much more satisfying to the eye. Result:

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

Estimating the Laplacian is just a matter of summing double derivatives

```
...

In [12]: (diff(diff(phi, x), x) +
    ....:  diff(diff(phi, y), y) +
    ....:  diff(diff(phi, z), z)).factor().subs(r, R)
Out[12]: 'Z*(Z**2*x**2 + Z**2*y**2 + Z**2*z**2 - 10*Z*r + 16)*exp(-Z*r/2)/(4*r)'

In [13]: (diff(diff(phi, x), x) + #Not quite satisfying.
    ....:  diff(diff(phi, y), y) + #Let's collect the 'Z' terms.
    ....:  diff(diff(phi, z), z)).factor().collect(Z).subs(r, R)
Out[13]: 'Z*(Z**2*(x**2 + y**2 + z**2) - 10*Z*r + 16)*exp(-Z*r/2)/(4*r)'

In [14]: (diff(diff(phi, x), x) + #Still not satisfying.
    ....:  diff(diff(phi, y), y) + #The r^2 terms needs to be substituted as well.
    ....:  diff(diff(phi, z), z)).factor().collect(Z).subs(r, R).subs(r**2, R**2)
Out[14]: 'Z*(Z**2*r**2 - 10*Z*r + 16)*exp(-Z*r/2)/(4*r)'

In [15]: (diff(diff(phi, x), x) + #Let's try to factorize once more.
    ....:  diff(diff(phi, y), y) +
    ....:  diff(diff(phi, z), z)).factor().collect(Z).subs(r, R).subs(r**2, R**2).factor()
Out[15]: 'Z*(Z*r - 8)*(Z*r - 2)*exp(-Z*r/2)/(4*r)'
```

Getting the right factorization may come across as tricky, but with minimal training this poses no real problems.

B.2 Using the auto-generation Script

The Python superclass `orbitalsGenerator` (the generator) aims to serve as a interface with the C++ `BasisFunctions` class, automatically generating the C++ code containing all the implementations of the derivatives for the given single particle states. The single particle states are implemented in the generator by subclasses overloading system specific virtual functions:

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

Implementing these functions will generate a Latex file listing the calculated expressions (see Sections B.3 and B.4), the constructor needed by the `Orbitals` subclass holding the generated orbitals and C++ header and source files containing the `BasisFunctions` implementation. In the case of Harmonic Oscillator functions, over 2000 lines of code is auto-generated.

$H_0(kx)$	1
$H_1(kx)$	$2kx$
$H_2(kx)$	$4k^2x^2 - 2$
$H_3(kx)$	$8k^3x^3 - 12kx$
$H_4(kx)$	$16k^4x^4 - 48k^2x^2 + 12$
$H_5(kx)$	$32k^5x^5 - 160k^3x^3 + 120kx$
$H_0(ky)$	1
$H_1(ky)$	$2ky$
$H_2(ky)$	$4k^2y^2 - 2$
$H_3(ky)$	$8k^3y^3 - 12ky$
$H_4(ky)$	$16k^4y^4 - 48k^2y^2 + 12$
$H_5(ky)$	$32k^5y^5 - 160k^3y^3 + 120ky$

Table B.1: Hermite polynomials used to construct orbital functions

B.3 Harmonic Oscillator Orbitals

Orbitals are constructed in the following fashion:

$$\phi(\vec{r})_{n_x,n_y} = H_{n_x}(kx)H_{n_y}(ky)e^{-\frac{1}{2}k^2r^2}$$

where $k = \omega\alpha$, with ω being the oscillator frequency and α being the variational parameter.

$\phi_0 \to \phi_{0,0}$	
$\phi(\vec{r})$	1
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2 x$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2 y$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 2\right)$

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

$\phi_1 \to \phi_{0,1}$	
$\phi(\vec{r})$	y
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2 xy$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\left(ky - 1\right)\left(ky + 1\right)$
$\nabla^2\phi(\vec{r})$	$k^2 y\left(k^2r^2 - 4\right)$

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

$\phi_2 \to \phi_{1,0}$	
$\phi(\vec{r})$	x
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\left(kx - 1\right)\left(kx + 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2 xy$
$\nabla^2\phi(\vec{r})$	$k^2 x\left(k^2r^2 - 4\right)$

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

$\phi_3 \to \phi_{0,2}$	
$\phi(\vec{r})$	$2k^2y^2 - 1$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2 x\left(2k^2y^2 - 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2 y\left(2k^2y^2 - 5\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 6\right)\left(2k^2y^2 - 1\right)$

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

$\phi_4 \to \phi_{1,1}$	
$\phi(\vec{r})$	xy
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-y\left(kx - 1\right)\left(kx + 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-x\left(ky - 1\right)\left(ky + 1\right)$
$\nabla^2\phi(\vec{r})$	$k^2 xy\left(k^2r^2 - 6\right)$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

B.4 Hydrogenic Orbitals

Orbitals are constructed in the following fashion:

$$\phi(\vec{r})_{n,l,m} = L^{2l+1}_{n-l-1}\Big(\frac{2r}{n}k\Big)S^m_l(\vec{r})e^{-\frac{r}{n}k}$$

where n is the principal quantum number, $k = \alpha Z$ with Z being the nucleus charge and α being the variational parameter.

$$l = 0,\ 1,\ ...,\ (n-1)$$

$$m = -l,\ (-l+1),\ ...,\ (l-1),\ l$$

$\phi_0 \to \phi_{1,0,0}$	
$\phi(\vec{r})$	1
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kx}{r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{ky}{r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kz}{r}$
$\nabla^2\phi(\vec{r})$	$\frac{k(kr-2)}{r}$

Table B.23: Orbital expressions hydrogenicOrbitals : 1, 0, 0. Factor e^{-kr} is omitted.

$\phi_1 \to \phi_{2,0,0}$	
$\phi(\vec{r})$	$kr-2$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kx(kr-4)}{2r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{ky(kr-4)}{2r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kz(kr-4)}{2r}$
$\nabla^2\phi(\vec{r})$	$\frac{k(kr-8)(kr-2)}{4r}$

Table B.24: Orbital expressions hydrogenicOrbitals : 2, 0, 0. Factor $e^{-\frac{1}{2}kr}$ is omitted.

$\phi_2 \to \phi_{2,1,0}$	
$\phi(\vec{r})$	z
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kxz}{2r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{kyz}{2r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kz^2-2r}{2r}$
$\nabla^2\phi(\vec{r})$	$\frac{kz(kr-8)}{4r}$

Table B.25: Orbital expressions hydrogenicOrbitals : 2, 1, 0. Factor $e^{-\frac{1}{2}kr}$ is omitted.

$\phi_3 \rightarrow \phi_{2,1,1}$	
$\phi(\vec{r})$	x
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kx^2-2r}{2r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{kxy}{2r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kxz}{2r}$
$\nabla^2\phi(\vec{r})$	$\frac{kx(kr-8)}{4r}$

Table B.26: Orbital expressions hydrogenicOrbitals : 2, 1, 1. Factor $e^{-\frac{1}{2}kr}$ is omitted.

$\phi_4 \rightarrow \phi_{2,1,-1}$	
$\phi(\vec{r})$	y
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kxy}{2r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{ky^2-2r}{2r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kyz}{2r}$
$\nabla^2\phi(\vec{r})$	$\frac{ky(kr-8)}{4r}$

Table B.27: Orbital expressions hydrogenicOrbitals : 2, 1, -1. Factor $e^{-\frac{1}{2}kr}$ is omitted.

C

Visualization of Data

With an increasingly massive code framework comes the increased need of data analysis tools. Data analysis usually involves visualization of the data; there is only so much information a single number can hold. To supplement the QMC code, a visualization tool has been developed, dramatically easing the implementation of additional visualization scripts.

C.1 DCViz

The tool DCViz (Dynamic Column data Visualizer) is a Python based visualization tool designed to plot data stored in columns. The plot library used is *Matplotlib*. The data can be plotted dynamically at a specified interval, and is hence designed to run parallel to the main application, e.g. DMC. The convergence of the DMC method is much better represented by a trailing energy graph than simply a stream of numbers. The same goes for the minimization process.

C.1.1 Implementing a Visualization Tool

Just like for the `orbitalsGenerator` tool (see Appendix B), specific implementations come in the shape of subclasses of the DCViz superclass. All the functionality regarding setting up figures, re-plotting dynamically, clearing figures to avoid random crashes if used repeatedly, etc. is inherited from the superclass. The necessary elements to implement is

`figMap`	A dictionary representing the names and structure of the plotted figures. If e.g. two figures `fig1` and `fig2` are wanted, where `fig1` should contain three sub-figures `subFig1`, `subFig2` and `subFig3`, the figure map should be set up as `figMap = {"fig1": ["subFig1", "subFig2", "subFig3"], "fig2": ["subFig4"]}` In class member functions, e.g. `self.subFig1` will be available as the figure representing the first sub-figure of the first figure.
`nametag`	A string representing the name of the output files associated with the given implementation (subclass). Full regular expressions support. E.g. `nametag = "DMC_out_\d+\.dat"`. All files loaded which matches the name-tag will be automatically plotted by the corresponding subclass implementation.
`plot(self, data)`	The loaded file will be loaded into a `data` object, which is an iterator where each elements represents a column in the data file. Designed such that expressions such as `col1, col2 = data` is possible (and fast). For the raw data loaded in a matrix, use `data.data` to directly access the loaded file. In the plot function, the sub-figures introduced in `figMap` should be loaded with data through e.g. `self.subFig1.plot(col1, col2)`. The superclass' main loop will take care of the rest.

An example implementation would be

```
class myTestClass(DCVizPlotter):
    nametag =  'testcase\d\.dat' #filename with regex support

    #1 figure with 1 subfigure
    figMap = {'fig1': ['subfig1']}

    #skip first row.
    skipRows = 1

    def plot(self, data):
        column1 = data[0]

        self.subfig1.set_title('I have $\LaTeX$ support!')

        self.subfig1.set_ylim([-1,1])

        self.subfig1.plot(column1)
```

C.1.2 Additional Support

Additional parameters can be overloaded for additional functionality

`nCols`	The number of columns present in the file. Will be automatically detected unless the data is stored in binary format.
`skipRows`	The number of initial rows to skip. Will be automatically detected unless the data is stores as a single column.
`skipCols`	The number of initial columns to skip. Defaults to zero.
`armaBin`	Boolean flag. If set to true, the data is assumed to be stored in Armadillo's binary format (doubles). Number of columns and rows will be read from the file header.
`fileBin`	Boolean flag. If set to true, the data is assumed to be stores in binary format. The number of columns must be specified.

The LaTeXsupport is enabled if the correct packages is installed.

Families

A specific implementation can be flagged to belong to a family by setting the class member variable `isFamilyMember` to true. If this flag is true, the folder where the originally data was loaded will be scanned for additional matches, all of which will be loaded into the `data` input to the plot function. In this case each element of the `data` list would be a column iterator as explained previously.

To keep track of which file a given data-set was loaded from, a list `self.familyFileNames` is created, where element i is the filename corresponding to `data[i]`.

A class member string `familyName` can be overridden to display a more general name in the auto-detection feedback. An example implementation using data file families would be

```
class myTestClassFamily(DCVizPlotter):
    nametag =  'testcaseFamily\d\.dat' #filename with regex support

    #1 figure with 3 subfigures
    figMap = {'fig1': ['subfig1', 'subfig2', 'subfig3']}

    #skip first row of each data file.
    skipRows = 1

    #Using this flag will read all the files matching the nametag
    #(in the same folder.) and make them aviable in the data arg
    isFamilyMember = True
    familyName = "testcase"

    def plot(self, data):

        #figures[0] is 'fig1' figures. the 0'th element is the
        #self.fig1 itself. Subfigures are always index [1:]
        mainFig = self.figures[0][0]
        mainFig.suptitle('I have $\LaTeX$ support!')
        subfigs = self.figures[0][1:]

        #Notice we plot fileData.data (In order to get the numpy object)
        #and not fileData alone, as fileData is a 'dataGenerator' instance
        #used to speed up file reading. Alternatively, we could send data[:]
        for subfig, fileData in zip(subfigs, data):
            subfig.plot(fileData.data)
            subfig.set_ylim([-1,1])
```

loading e.g. `testcaseFamily0.dat` would automatically load `testcaseFamily1.dat` etc. as well.

C.1.3 Usage: The API, Terminal Client and GUI

All listed interfaces to the DCViz core has full warning support and reconfigurability. DCViz, like the QMC code in general, is designed to be used by other people than the Author.

The Application Programming Interface (API)

The DCViz library has been developed to interface nicely with any python script [1] Given a path to the data file, all that is needed in order to visualize it is

```
import DCVizWrapper as viz
dynamicMode = False #or true
```

[1] DCViz can also be called through C++, however, no header has been implemented for this. Typically one would use the std::system or std::thread to start the script in the background.

```
3
4 ...
5 #Perform some calculations and store these in the file myDataFile (including path)
6
7 #DCVizWrapper.main() automatically detects the subclass implementation
8 #matching the specified file. Thread safe and easily interruptable.
9 viz.main(path=myDataFile, dynamic=dynamicMode)
```

Using the Terminal

The `DCVizWrapper.py` script can also be called directly from a terminal using the path as first command line argument. If the option `-d` is supplied, dynamic mode is activated.

The GUI

The script `DCVizGUI.py` sets up a GUI for visualizing data using DCViz. The GUI is implemented using pyside (python wrapper for QT), and is designed to be simple. Data files are loaded from an open-file dialog, and will appear in a drop-down menu once loaded. The play button executes the main loop of the currently selected data file. Dynamic mode is selected though a check-box, and the refresh interval is set by a slider (from zero to ten seconds). Warnings can be disabled through the configuration file.

The GUI file can be called from another script (threaded) with main path as first commandline arg.

Ha med screenshot av DCViz i action.

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

[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

[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://dx.doi.org/10.1007/s11263-008-0168-y

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

[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