Quantum-mechanical systems in traps and density functional theory

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 7

I Theory 9

2 Scientific Programming 11

2.1 Programming Languages . . . 11

2.1.1 High-level Languages . . . 11

2.1.2 Low-level Languages . . . 12

2.2 Object Orientation . . . 13

2.2.1 Inheritance . . . 13

2.2.2 Pointers, Virtual Functions and Types . . . 15

2.2.3 Const Correctness . . . 18

2.2.4 Accessibility levels and Friend classes . . . 18

2.2.5 Example: PotionGame . . . 19

2.3 Structuring the code . . . 22

2.3.1 File structures . . . 23

2.3.2 Consistent Code Styles . . . 23

2.3.3 Version Control . . . 23

2.3.4 Using GUI Tools . . . 24

3 Quantum Monte Carlo 25

3.1 Modeling Diffusion . . . 25

3.1.1 The Diffusion Equation . . . 25

II Results 29

4 Implementation and Validation 31

4.1 Structure and Implementation . . . 31

4.1.1 Methods used for Increasing Readability and Overall Structure . . . 31

4.1.2 Methods for Generalizing the Code . . . 33

4.1.3 Visualization . . . 37

4.2 Performance Optimizations . . . 37

4.2.1 RAM use . . . 37

4.2.2 CPU-time . . . 39

4.3 Validation . . . 41

5 Results 43

5.1 Validating the code . . . 43

5.1.1 Calculation for non-interacting particles . . . 43

A Dirac Notation 45

B Matrix representation of states and operators 47

Bibliography 51

Chapter 1

Introduction

blah blah

Part I

Theory

Chapter 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 this thesis, Python is the mainly used scripting language.

---

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

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

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

```python
#Sum100Python.py
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:

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

int main(){

```

---

[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.

```
6           int S = 0;
7           for (int i = 1; i <= 100; i++){
8                   S += i;
9           }
10
11          std::cout << S << std::endl;
12
13          return 0;
14  }
```

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

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.

```cpp
#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 spesifically

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

public:
    Potential(int n_p, int dim);
    Potential();

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

};

class Coulomb : public Potential {
public:

    Coulomb(GeneralParams &);

    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).

```cpp

};

class Harmonic_osc : public Potential {
protected:
    double w;

public:

    Harmonic_osc(GeneralParams &);

    double get_pot_E(const Walker* walker) const;

};
```

```cpp
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;
}
```

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

```cpp
class System {
protected:
    ...

    std::vector<Potential*> potentials;

    ...
};

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

    //Iterate through the potentials 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:

```

class QMC {
protected:

    ...

    std::stringstream s;

    ...

    int n_c;

    ...

    int cycle;

    ...

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()

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

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


class HealthPotion(Potion):

    #Constructor is inherited

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

    def setName(self):
        self.name = "Health Potion (" + str(self.amount) + ")"


class EnergyPotion(Potion):

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

    def setName(self):
        self.name = "Energy Potion (" + str(self.amount) + ")"
```

```
#playerClass.py

class Player:

    #Initialize the player at full health with no potions
    def __init__(self, name):
        self.health = 100
        self.energy = 100
        self.name = name

        self.dead = False

        self.potions = []

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

    #Selects the given potion and consumes it. The potion needs to know
    #the player it should affect, hence we send 'self' as an argument.
    def usePotion(self, potionIndex):

        print "%s consumes %s." % (self.name, self.potions[potionIndex].name)

        self.potions[potionIndex].applyPotion(self)
        self.potions.pop(potionIndex)




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

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


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

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

    #lists the potions to the user
    def displayPotions(self):

        if len(self.potions) == 0:
            print "No potions aviable"

        for potion in self.potions:
            print potion.name

    def attack(self, player):

        energyCost = 50
        damage = 40

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

        print "\n%s hit %s for %s using %s energy" % (self.name, player.name,
                                                      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:

```
#potionGameMain.py

from potionClass import *
from playerClass import *

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


player1 = Player('john');
player1.addPotion(HealthPotion(10)); player1.addPotion(EnergyPotion(30))

player2 = Player('james')
player2.addPotion(EnergyPotion(20)); player2.addPotion(EnergyPotion(20))

#Initial output
roundOutput([player1, player2], 0)

#Round one: Each player gets an attack after which both can consume potions
player1.attack(player2)
player1.usePotion(1); player2.usePotion(0)

player2.attack(player1)
player1.usePotion(0); player2.usePotion(0)

roundOutput([player1, player2], 1)
#Round one end.
#...
```

```
~$ python potionGameMain.py

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

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


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

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

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

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

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

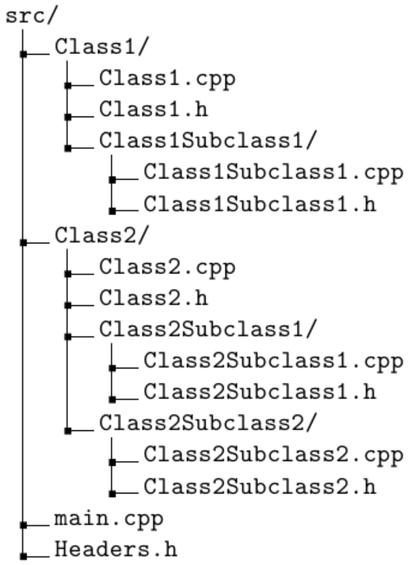
In this section I have not focused too much on scientific computing, but rather on the use of object orientation in general. When the physical methods are discussed in section **Insert physics section**, I will get back to a more specific description of scientific programming.

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

2.3 Structuring the code

Structuring a code is another source of compromises. If the code is short, and has a direct purpose, e.g. to calculate the sum from Eq. (2.1.1), structure is not an issue at all, given that reasonable variable names are provided. However, if the code is more complex, and the methods used are specific implementations of a more general case, e.g. integration, code structuring becomes very important. For details about the structuring of the code used in this thesis, see Section 4.1.

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



```
src/
├── Class1/
│   ├── Class1.cpp
│   ├── Class1.h
│   └── Class1Subclass1/
│       ├── Class1Subclass1.cpp
│       └── Class1Subclass1.h
├── Class2/
│   ├── Class2.cpp
│   ├── Class2.h
│   ├── Class2Subclass1/
│   │   ├── Class2Subclass1.cpp
│   │   └── Class2Subclass1.h
│   └── Class2Subclass2/
│       ├── Class2Subclass2.cpp
│       └── Class2Subclass2.h
├── main.cpp
└── Headers.h
```

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

2.3.1 File structures

Developing codes in scientific scenarios often involves enormous frameworks. For instance, when doing Molecular Dynamics, several collision models, force models etc. is perhaps implemented alongside the main solver. In the case of Markow Chain Monte Carlo methods, different diffusion models (sampling rules) may be selectable. Even though these models are implemented object oriented using polymorphism, the code still gets messy when the amount of subclasses etc. gets large.

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

A standard way to organize code is to have all the source code gathered in a *src* folder, with one folder per distinct class. Subclasses should appear as folders inside the superclass folder. Figure 2.1 shows an example setup for the framework of an object oriented code.

2.3.2 Consistent Code Styles

KEEP TO ONE STANDARD!

2.3.3 Version Control

USE E.G. GIT!

2.3.4 Using GUI Tools

stand-alone openfile dialoges are awesome!

Chapter 3

Quantum Monte Carlo

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. *Markow 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 The Diffusion Equation

Isotropic Diffusion

Isotropic diffusion is a process in which diffusing particles sees all possible directions as an equally probable path. Eq. (3.1) 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.

$$\frac{\partial P(\vec{r},t)}{\partial t} = D\nabla^2 P(\vec{r},t) \tag{3.1}$$

From experiment there is a strong indication that the *probability flux*, $j(\vec{r},t)$, is proportional to the gradient of the distribution [5], that is

$$j(\vec{r},t) = -D\nabla|\psi(\vec{r},t)|^2. \tag{3.2}$$

If we on the other hand look at the time derivative of the Wave function squared, and insert expressions for the Schrödinger equation (and its complex conjugate), we can relate it to $j(\vec{r},t)$

$$\begin{aligned}
\frac{\partial|\psi(\vec{r},t)|^2}{\partial t} &= \psi(\vec{r},t)\frac{\partial\psi(\vec{r},t)^*}{\partial t} + \psi(\vec{r},t)^*\frac{\partial\psi(\vec{r},t)}{\partial t} \\
&= i\left(\frac{1}{2}\psi(\vec{r},t)^*\nabla^2\psi(\vec{r},t) - V(\vec{r})|\psi(\vec{r},t)|^2 - \frac{1}{2}\psi(\vec{r},t)\nabla^2\psi(\vec{r},t)^* + V(\vec{r})|\psi(\vec{r},t)|^2\right) \\
&= \frac{1}{2}i\Big(\psi(\vec{r},t)^*\nabla^2\psi(\vec{r},t) - \psi(\vec{r},t)\nabla^2\psi(\vec{r},t)^*\Big) \\
&= -\nabla\cdot\left(\frac{1}{2i}\left(\psi(\vec{r},t)^*\nabla\psi(\vec{r},t) - \psi(\vec{r},t)\nabla\psi(\vec{r},t)^*\right)\right) \\
&= -\nabla\cdot j(\vec{r},t)
\end{aligned}$$

You can argue that the last equality should be a definition, since the equation derived above obviously is the Quantum Mechanical analogue to the continuity equation in classical mechanics[1]. However, this does not change the fact that inserting the result from Eq. (3.2) yields a diffusion equation

$$\begin{aligned}
\frac{\partial|\psi(\vec{r},t)|^2}{\partial t} &= -\nabla\cdot j(\vec{r},t) \\
&\approx D\nabla^2|\psi(\vec{r},t)|^2,
\end{aligned} \tag{3.3}$$

where the value of the diffusion constant, $D = \frac{1}{2}$, can be extracted from the Scrödinger equation [6].

Given that we model the diffusion of walkers by the process described above, the equations themselves serves no practical use. In order to achieve specific sampling rules for our walkers, we need a connection between the time-dependence of the probability distribution and the time-dependence of the 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. Given e.g. the simple isotropic diffusion described previously, solving the Langevin equation using a Forward Euler approximation for the time derivative results in the following relation:

$$x_{i+1} = x_i + \xi, \qquad \begin{aligned} \sigma(\xi) &= \sqrt{2D\delta t}, \\ \langle\xi\rangle &= x_i, \end{aligned} \tag{3.4}$$

where ξ is a normal distributed number. This relation is in agreement with the isotropy of Eq. (3.1) in the sense that the displacement is symmetric around the current position.

---

[1] It should be mentioned that the continuity equation holds as long as the number of particles in the system is conserved [5]. Chemists often break this condition, e.g. when modeling chemical reactions.

Anisotropic Diffusion

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.

$$\frac{\partial P(\vec{r},t)}{\partial t} = D\nabla \cdot \left[\left(\nabla - \vec{F}(\vec{r},t)\right)P(\vec{r},t)\right] \tag{3.5}$$

The remarkable thing is that simple isotropic diffusion processes obey this relation [6]. 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. This method of *Importance Sampling* will be discussed in Section (ref imp).

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. (3.5) 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

$$\begin{aligned} \vec{F}(\vec{r},t) &= \frac{1}{P(\vec{r},t)}\nabla P(\vec{r},t) \\ &= \frac{2}{|\psi(\vec{r},t)|}\nabla|\psi(\vec{r},t)| \qquad (3.6) \end{aligned}$$

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.

The Langevin Equation for the Fokker-Planck Equation

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

$$\frac{\partial x_i}{\partial t} = DF(\vec{r},t)_i + \eta, \tag{3.7}$$

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

$$x_{i+1} = DF(\vec{r},t)_i \delta t + \xi, \tag{3.8}$$

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 [7], [8] and [9].

Part II

Results

Chapter 4

Implementation and Validation

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

4.1 Structure and Implementation

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

4.1.1 Methods used for Increasing Readability and Overall Structure

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

```
DMC::DMC(...) {

    ...

    original_walkers = new Walker*[spesify a number of walkers];

    ...

}
```

---

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

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

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

```cpp
void DMC::initialize() {

    ...

    //Initializing active walkers
    for (int k = 0; k < n_w; k++){
        original_walkers[k] = new Walker(n_p, dim);
    }

    //Seting trial position of active walkers
    ...

    //Calculating and storing energies of active walkers
    for (int k = 0; k < n_w; k++) {
        calculate_energy_necessities(original_walkers[k]);
        original_walkers[k]->set_E(calculate_local_energy(original_walkers[k]));
    }

    ...

}
```

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

```cpp
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;
}
```

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

```
void VMC::initialize() {

    ...

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

    ...

}
```

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

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

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

}
```

4.1.2 Methods for Generalizing the Code

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

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

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

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

}
```

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

double Simple::get_new_pos(const Walker* walker, int i, int j) {
    return Diffusion::get_new_pos(walker, i, j);
}

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

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

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

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

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

(ii) Importance- and Brute Force-sampling.

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

(iv) Fermions and Bosons.

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

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

Constraints (i) - (iv)

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

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

    //Positions and orbitals are updated for the particle at hand
    ...

    //updates the inverse slater in case of a fermion system
    qmc->get_system_ptr()->calc_for_newpos(walker_pre, walker_post, particle);

    //pure virtual function. Function will update quantum force if importance sampling.
    update_necessities(walker_pre, walker_post, particle);

}
```

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

Constraint (v)

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

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

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

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

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

}
```

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

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

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

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

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

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

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

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

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

```
class ExpandedBasis : public Orbitals {

...

protected:

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

};

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

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

    //read coefficients

}
```

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

```

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

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

    return value;

}
```

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

Constraint (vi)

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

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

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

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

4.1.3 Visualization

DCVIZ?

4.2 Performance Optimizations

4.2.1 RAM use

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

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

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

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

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

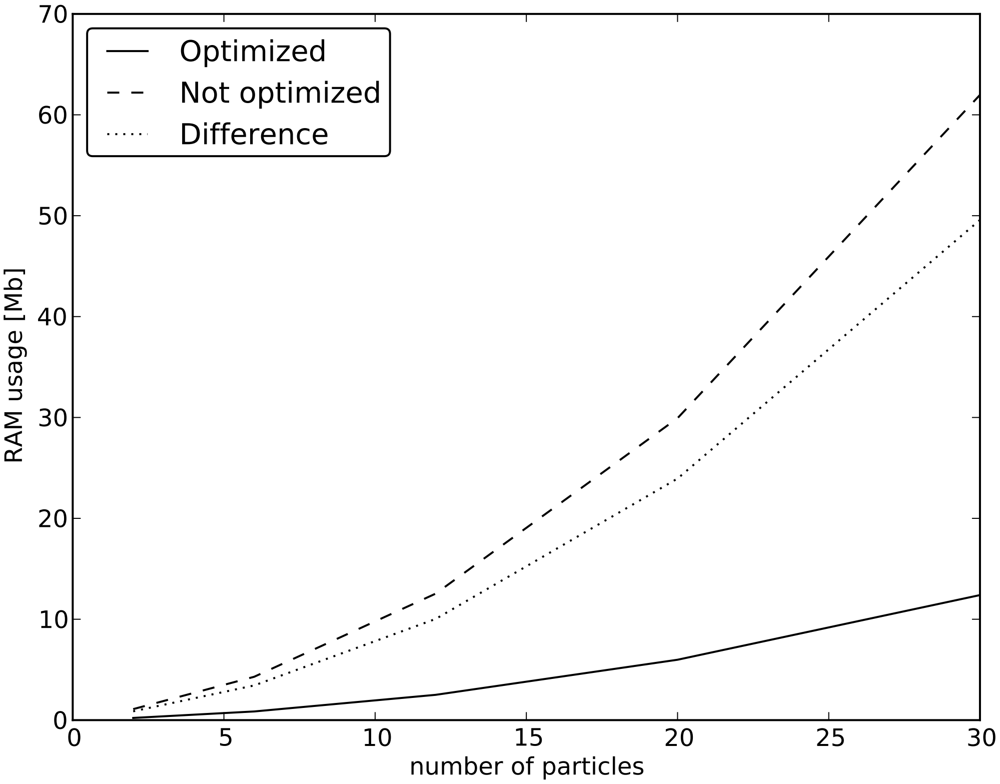


Figure 4.1: Number of particles vs. the theoretical RAM usage in a typical DMC run for the optimized and unoptimized case. Calculated for two dimensions, 1000 active walkers and 5000 inactive.

4.2.2 CPU-time

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

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

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

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

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

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

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

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

For Fermions, it can be shown that we have the following relations for the Slater determinant[10]:

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

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

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

Calculating an inverse does not seem to optimize much. However, we can run an updating algorithm for the inverse, which ends up saving a lot of time. The ratio of the spatial determinants can also be expressed using this inverse[10]. This means that no explicit wave function calculation is needed (we can calculate the ratio between two Jastrow factors without problem). The expression for the ratio in terms of the inverse is

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

where i is the particle being moved.

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

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

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

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

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

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

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

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

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

$$\begin{aligned} \frac{1}{\psi_T}\frac{\partial \psi_T}{\partial \alpha} &= -\frac{1}{2}\omega \sum_{i=1}^{N} r_i^2, \qquad (4.7) \\ \frac{1}{\psi_T}\frac{\partial \psi_T}{\partial \beta} &= -\sum_{i<j} \frac{a_{ij} r_{ij}^2}{(1+\beta r_{ij})^2}. \qquad (4.8) \end{aligned}$$

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

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

4.3 Validation

things should not be wrong. It is bad.

test:

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

Table 4.2: J. Hogberget

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

Table 4.3: M. Pedersen Lohne et al.

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

Table 4.4:

Chapter 5

Results

5.1 Validating the code

5.1.1 Calculation for non-interacting particles

[2]

Appendix 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

$$\begin{aligned} |\psi\rangle &= N\Big[\,|{\uparrow\downarrow}\rangle - i\,|{\downarrow\uparrow}\rangle\,\Big] && \text{(A.1)} \\ \langle\psi| &= N\Big[\,\langle{\uparrow\downarrow}| + i\,\langle{\downarrow\uparrow}|\,\Big] && \text{(A.2)} \end{aligned}$$

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.

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

---

[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[11].

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:

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

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

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

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

Appendix B

Matrix representation of states and operators

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

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

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

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

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

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

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

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

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

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

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

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

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

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

---

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

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

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

        n_w_tot = arma::accu(n_w_list);

    }
#else
    n_w_tot = n_w;
#endif

}

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

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

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


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


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


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


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

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

#endif
}
```

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

Table B.1: 30 particles lolol

Bibliography

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

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

[3] ——, *A primer on scientific programming with Python.* Berlin; Heidelberg; New York: Springer, 2011. [Online]. Available: http://www.worldcat.org/search?qt=worldcat_org_all&q=9783642183652; http://www.bibsonomy.org/bibtex/2c5c7c9849e177a04c30dd31b0b5615f7/ulger

[4] G. O'Regan, *A Brief History of Computing, Second Edition.* Springer, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-1-4471-2359-0;http://www.bibsonomy.org/bibtex/29ccda94c4c5c89a6ee3ef44d8d10671e/dblp

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

[6] 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.

[7] 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

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

[9] 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.

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

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