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

Chapter 1

Introduction

blah blah

Part I

Theory

Chapter 2

Scientific Programming

The introduction of the computer around 1945 had a major impact on the mathematical fields of science. Previously unsolvable problems were now easily 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 atomic bombs as a result of the Manhattan Project at the end of the second world war **citation needed**.

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*, translates your program from e.g. C++ code into machine code. Other languages as Python are interpreted real-time and therefore require no compilation. 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

This subgroup of languages are often referred to as *scripting languages*. A script is a short code, often with a specific purpose such as analyzing output by e.g. generating tables and figures from raw data, or gluing together different programs which are meant to be run in sequential order.

For simple jobs as these, taking only seconds to run on modern computers, we do not need an optimized code, but rather an easily read, clutter-free code. The 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 I will emphasize the use of Python as a 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 an open source programming language with a focus on simplicity over complexity. 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 is 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 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, 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 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 exactly this, whereas C++ would crash runtime leaving nothing but a "segmentation fault" for the user to interpret. However, when the optimized program ends up running for days, the extra time spent developing it pays off. As a rule of thumb, higher-level languages (without focus on *vectorization* or other optimization techniques) run 100 times slower than lower level ones.

C++

C++ is a language developed by Bjarne Stroustrup in 1979 at Bell Labs. It serves as an extension to the original *C* language, adding e.g. object oriented features such as classes. To calculate the sum in Eq. 2.1.1 with C++, we would need something like this:

```
1 //Sum100C++.cpp
2 #include <iostream>
3
4 int main(){
5
6         int S = 0;
7         for (int i = 1; i <= 100; i++){
8                 S += i;
```

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

```
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*, developed by the Norwegian scientists Ole-Johan Dahl and Kristen Nygaard**Need specifics and citations**. 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*, an extension to standard *structs*. A class is nothing but a collection of variables and functions under a sensible name, however, they provide a great deal of functionality like *inheritance* and accessibility control. To better illustrate the concept of classes and inheritance, let us first look at an example.

2.2.1 Inheritance

All keyboards have two things in common: A board and keys. 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. Let us look at how this concept can be converted into Python code

```
1  #KeyboardClassPython.py
2
3  class Keyboard():
4
5           keys
6           board
7
8           #No implementations (pure virtual)
9           def setupKeys(self):
10
```

```
11          def pressKey(self, key):
12
13
14
15  class ComputerKeyboard(Keyboard):
16
17          language
18
19          def __init__(self, language):
20                  self.language = language
21
22                  self.setupKeys()
23
24          def setupKeys(self):
25                  if self.language == "Norwegian":
26                          keys[0] = "|"
27                          keys[1] = "1"
28                          #...
29
30
31
32                  #...
33
34          def pressKey(self, key):
35                  return self.keys[key]
36
37
38
39  class MusicalKeyboard(Keyboard):
40
41          def __init__(self, nKeys, noteLength, amplitude):
42
43                  self.amplitude = amplitude
44                  self.noteLenght = noteLength
45
46                  keys = zeros(nKeys)
47                  self.setupKeys()
48
49          def setupKeys(self):
50                  keys[0] = lowestNote
51                  #...
52
53                  keys[30] = 440 #Hz
54                  #...
55
56          def pressKey(self, key):
57
58                  t = linspace(0, self.noteLength, 100)
59                  pi = 3.141592
60
61                  return self.amplitude*sin(2*pi*keys[key]*t)
62
63
```

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*.

2.2.2 Virtual Functions, pointers and types

It a virtual function is overridden, the latest implementation will be called. `setupKeys` and `pressKey` are examples of this, however, they are in a sense more than virtual, since they are not even implemented. They have to be overridden in order to work. These functions are referred to as *pure virtual* functions. In python, all class functions, or member functions, are virtual. In C++ however, we have to specify whether or not a function is virtual in the declaration.

Higher level languages like Python handles all the pointers by itself. In low-level languages like C++, however, you need to control these yourself. A pointer is a hexadecimal number representing a memory address. If you pass a pointer to an object, e.g. `Someclass* someobject`, as an argument to a function, whenever that function changes a class variable, the value is changed globally, since the memory address is directly accessed. If you instead choose to send the object without a pointer declaration, e.g. `Someclass someobject`, changing the value will not change the object globally. What happens instead is that you change a local copy of the object. It is a misconception that pointers are your enemies, they are, quite frankly, making your codes much easier.

A pointer holds a type, that is, `int`, `double`, or any class that you have access to. In the following example we will study the interplay between virtual functions, pointers and types:

```
#include <iostream>

using namespace std;

class VirtualTest{
public:
        virtual void virtualFunc(){
                cout << "superclass virtualFunc called" << endl;
        }

        void notVirtualFunc(){
                cout << "superclass notVirtualFunc called" << endl;
        }

};

class subclass : public VirtualTest{
public:
        virtual void virtualFunc(){
                cout << "subclass virtualFunc called" << endl;
        }

        void notVirtualFunc(){
                cout << "subclass notVirtualFunc called" << endl;
        }

};

void function(VirtualTest* object){
        object->virtualFunc();
        object->notVirtualFunc();
}

int main(){

        cout << "-Calling subclass object of type VirtualTest*" << endl;
        VirtualTest* object = new subclass();
        function(object);

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

        cout << endl << "-Directly calling object of type subclass*" << endl;
        object2->virtualFunc();
        object2->notVirtualFunc();

        return 0;
}
```

```
~$ ./virtualFunctionsC++.x
-Calling subclass object of type VirtualTest*
subclass virtualFunc called
```

```
superclass notVirtualFunc called

-Calling subclass object of type subclass*
subclass virtualFunc called
superclass notVirtualFunc called

-Directly calling object of type subclass*
subclass virtualFunc called
subclass notVirtualFunc called
```

In the first call, the pointer is declared as a `VirtualTest*` type, however, it is still initialized to be a subclass pointer in the sense that the subclass' functions are loaded into the object. This results in the virtual function being overridden (as mentioned previously), but since the object type is `VirtualTest*`, C++ does not dig deeper than the superclass when it looks for the non-virtual function implementation. In other words: `virtual` induce a search for deeper implementations of the same function, given that the function is loaded through e.g. `new subclass()`.

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 object. If it receives anything else, it simply attempts to convert it, or *cast* it to a different type; *typecasting*[3]. In this case it works just fine. The third call, outside the function, demonstrates that if we declare it as a subclass type, both functions are overridden, since we do not have to go through the superclass at all.

The strength of using virtual functions in a class hierarchy is that you can easily expand or implement new functionality without completely rewriting the code. As an example, in the `QMC` code, changing potentials, or switching between closed form and numerical expressions for the dell and laplacian, is just a matter of which object's functions are called by the energy function.

```
double QMC::calculate_local_energy(Walker* walker) const {
    return kinetics->get_KE(walker) + system->get_potential_energy(walker);
}
```

```
class Walker {
...

    mat r;
...
};
```

```
class Kinetics {
...

    virtual double get_KE(const Walker* walker) = 0;
...
};

class Numerical : public Kinetics {
...

    virtual double get_KE(const Walker* walker);
...
};
```

```
class Potential {
...

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

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

```
7
8  class Harmonic_osc : public Potential {
9  ...
10
11     virtual double get_pot_E(const Walker* walker) const;
12 ...
13 };
```

```
1  double Harmonic_osc::get_pot_E(const Walker* walker) const {
2
3      double e_potential = 0;
4
5      for (int i = 0; i < n_p; i++) {
6          e_potential += 0.5 * w * w * walker->get_r_i2(i);
7      }
8
9      return e_potential;
10 }
```

The `QMC` member function receives a `Walker*` object, representing a set of positions for all particles. The `kinetics` object is of type `Kinetics*` and holds implementations of either a numerical calculation or a method for extracting closed form expressions from the orbitals. The `get_potential_energy` method is not virtual. It's purpose is to iterate over all potentials given (i.e. a Harmonic Oscillator and Coulomb), extracting their value at the walker's position. Adding a new potential to this list is extremely simple.

The point is that the local energy function is written completely independent of how the potential actually looks. The implementation of a new potential would mean a new subclass of `Potential` with a new implementation of the virtual function `get_local_E`. This makes the code extremely readable (given it is properly commented), since the reader would not have to care about the details if the algorithm is that of interest (and vice versa).

2.2.3 Const Correctness

In the `QMC` 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.

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 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. All public variables are inherited to subclasses when inherited as in the virtual function example[4]. Declarations beneath the `private` part stops all other classes than instances of it self from reaching it, even subclass instances. If you want private variables inherited, the `protected` accessibility level should be used.

There is one exception to the rule of protected and private variables, namely *friend* classes. In the `QMC` library, 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:

    int n_c;

    int n_p;
    int n2;
    int dim;

    int cycle;

    ...

    Walker* original_walker;
    Walker* trial_walker;

    ...

public:
...

    friend class Distribution;
...
};
```

```
void Distribution::dump() {

    if ((qmc->cycle > qmc->n_c / 2) && (qmc->cycle % 100 == 0)) {
        for (int i = 0; i < qmc->n_p; i++) {
            for (int j = 0; j < qmc->dim; j++) {
                if (j == qmc->dim - 1) {
                    file << qmc->original_walker->r(i, j);
                } else {
                    file << qmc->original_walker->r(i, j) << " ";
                }
            }
            file << endl;
        }
    }

}
```

Without going into details, we can see that `Distribution` has full access to protected members of `QMC* qmc`. Friend classes is a savior 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 flying 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.

[4]You could inherit with protected option as well, but it is rarely used and messes up the functionality.

2.2.5 Example: PotionGame

To end the section I would like to demonstrate the versatile power of object orientation. Consider the following codes (libraries)

```
#potionClass.py

class Potion:

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

        def applyPotion(self, player):
                #Pure virtual function
                return None

        def setName(self):
                #Virtual function
                self.name = "Undefined"


class HealthPotion(Potion):

        #Constructor is inherited

        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:
        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)

        def usePotion(self, potionIndex):
                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
```

```
28                          self.dead = True;
29
30
31          def changeEnergy(self, amount):
32                  self.energy += amount
33
34                  #Cap energy at [0,100].
35                  if self.energy > 100:
36                          self.energy = 100
37                  elif self.energy < 0:
38                          self.energy = 0
39
40
41          def superPowers(self):
42                  #Pure virtual function
43                  return None
44
45          def displayPotions(self):
46                  if len(self.potions) == 0:
47                          print "No potions aviable"
48                  for potion in self.potions:
49                          print potion.name
50
51  class Superman(Player):
52          def superPowers(self):
53                  self.canFly = True;
```

We have a `Player` class keeping track of a players energy and health level, and which potions the player is carrying. 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. 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 player2 = Player('james')
17
18 player1.addPotion(HealthPotion(10))
19 player1.addPotion(EnergyPotion(30))
20
21 player2.addPotion(EnergyPotion(20))
22 player2.addPotion(EnergyPotion(20))
23
24 roundOutput([player1, player2], 0)
25
26 print "\nPlayer2 hit Player1 for 50 using 100 energy"
27 player1.changeHealth(-50)
28 player2.changeEnergy(-100)
29
30 print "Players used all potions"
31 player1.usePotion(1)
32 player1.usePotion(0)
33 player2.usePotion(1)
34 player2.usePotion(0)
35
```

```
36 print "Player1 hit Player2 for 40 using 90 energy\n"
37 player1.changeEnergy(-90); player2.changeHealth(-40)
38
39 roundOutput([player1, player2], 1)
```

```
~$ 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)


Player2 hit Player1 for 50 using 100 energy
Players used all potions
Player1 hit Player2 for 40 using 90 energy


Round 1:
john (hp/e=60/10):
No potions aviable

james (hp/e=60/40):
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

blah blah messy=bad, organized = good

2.3.1 File structures

blah blah src folder good yes

2.3.2 IDEs

Netbeans good yes, spider good yes

Part II

Results

Chapter 3

Implementation and Validation

For a detailed description of specific functions etc., see the comments in the actual code. The general idea of the implementation is to use the power of object orientation (for details, see Section 2.2) to compile the different building blocks of QMC-methods in a natural coherent way. The first step to accomplishing this is to extract the different (more or less) independent parts of the machinery.

3.1 Structure and Implementation

In QMC, and Quantum Mechanics in general, there are several natural ways of decoupling the code. Gathering data into objects are in some cases done to increase the readability and the overall structure, which dramatically decreases the time it takes to implement or debug new methods. 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).

3.1.1 Methods used for Increasing Readability and Overall Structure

As an example, take a look at the contents of the Walker class in Table 3.1.1. The power of this way of structuring the code, 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`. A function which requires access to several elements from Table 3.1.1 now only requires one argument, namely the walker of interest. Let us look at an example

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

    ...

    int max_walkers = K * n_w;
    original_walkers = new Walker*[max_walkers];

    ...

}
```

```
void DMC::initialize() {

    jastrow->initialize();

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

Table 3.1: Description of the members in an instance of the Walker class. All matrices holds information on all particles.

`arma::mat r`	The position.
`arma::mat r_rel`	The relative positions.
`arma::rowvec r2`	The squared positions.
`arma::mat qforce`	The quantum force.
`arma::mat spatial_grad`	The gradient of the uncorrelated wave function.
`arma::mat jast_grad`	The gradient of the Jastrow factor.
`arma::mat inv`	The inverse of the slater matrix.
`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`	False if active, true if not.

```
    }

    //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]));
    }

    //Creating unactive walker objects (note: 3. arg=false implies dead)
    for (int k = n_w; k < K * n_w; k++) {
        original_walkers[k] = new Walker(n_p, dim, false);
    }

}
```

```
Walker::Walker(int n_p, int dim, bool alive) {
    this->dim = dim;
    this->n_p = n_p;
    this->n2 = n_p / 2;


    if (alive) {
        is_murdered = false;
    } else {
        is_murdered = true;
    }

    r = zeros<mat> (n_p, dim);
    r_rel = zeros<mat> (n_p, n_p);
    qforce = zeros<mat> (n_p, dim);
    jast_grad = zeros<mat> (n_p, dim);
    spatial_grad = zeros<mat> (n_p, dim);

    r2 = zeros(1, n_p);

    value = 0;
    lapl_sum = 0;
    spatial_ratio = 0;
    inv = zeros<mat> (n2, n_p);

}
```

As we can see from the code above, initializing new walkers is unproblematic. When e.g. looping over walkers in DMC, the amount of juggling is reduced to nothing; all you need to do is to loop over an array of walkers. This walker can then be sent to any function, resulting in code like e.g.

```
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. Stating e.g. that at cycle zero, the original and trial walker should be equal, is now implemented in the following way:

```
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(int n_p, int dim, double timestep, long random_seed, double D)
: Sampling(n_p, dim) {
    diffusion = new Fokker_Planck(n_p, dim, timestep, random_seed, D);
}
```

3.1.2 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 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 will give us the values we need once the correct objects are in place. This is reflected in the following code:

```
void QMC::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)
                + sampling->get_new_pos(walker_pre, particle, j);
    }

    ...

}
```

```
double Sampling::get_new_pos(const Walker* walker_pre, int particle, int j) const {
    return diffusion->get_new_pos(walker_pre, particle, j);
}
```

```cpp
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 virtual functions to generalize the code is used throughout the entire code. The goals of this thesis was to produce a code with the following generalizations:

- As many as possible functions should be written generally for DMC and VMC (see Section **REF** for a complete list of the common parts.)
- 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).

This puts a series of constraints on the code; it should be general for:

(i) Importance- and Brute Force-sampling.

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

(iii) Fermions and Bosons.

(iv) Any choice of single particle basis.

(v) Functionality to add any potential.

It is also implemented a general Jastrow factor, a Walker class which is easily extendable, and an output function which performs any form of output (or nothing), without having to flag / comment the code (it works in a similar way to how QMC extracts potential energies, which will be discussed later).

Constraints (i) - (iii)

As discussed in the beginning of this chapter, `QMC` holds an object of type `Sampling`, which holds all the functions for moving particles, and ensures that the walker has access to the correct necessities prior to e.g. the energy calculations (which again is different for `Numerical` or `Closed_Form`). Below follows a part of the code illustrating this; the code for copying walkers, calculating energy necessities etc. follows the same idea.

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

    //position updated
    ...

    sampling->update_necessities(walker_pre, walker_post, particle);

}
```

```
void Brute_Force::update_necessities(const Walker* walker_pre, Walker* walker_post, int
    particle) {
    qmc->get_wf_value(walker_post);
}

...

void Importance::update_necessities(const Walker* walker_pre, Walker* walker_post, int
    particle) {
    qmc->get_kinetics_ptr()->update_necessities_IS(walker_pre, walker_post, particle);
    qmc->get_kinetics_ptr()->get_QF(walker_post);
}
```

```
void Numerical::update_necessities_IS(const Walker* walker_pre, Walker* walker_post, int
    particle) const {
    qmc->get_wf_value(walker_post);
}

...

void Closed_form::update_necessities_IS(const Walker* walker_pre, Walker* walker_post,
    int particle) const {
    walker_post->spatial_ratio = qmc->get_system_ptr()->get_spatial_ratio(walker_pre,
        walker_post, particle);
    qmc->get_system_ptr()->calc_for_newpos(walker_pre, walker_post, particle);
    qmc->get_gradients(walker_post, particle);
}
```

```
double Fermions::get_spatial_ratio(const Walker* walker_pre, const Walker* walker_post,
    int particle) const {
    int q_num;
    double s_ratio;

    s_ratio = 0;
    for (q_num = 0; q_num < n2; q_num++) {
        s_ratio += orbital->phi(walker_post, particle, q_num) * walker_pre->inv(q_num,
            particle);
    }

    return s_ratio;
}

...

void Fermions::calc_for_newpos(const Walker* walker_old, Walker* walker_new, int particle
    ) const {
    update_inverse(walker_old, walker_new, particle);
}



BOSONS NOT IMPLEMENTED
```

We can see that the branching goes as follows (corresponding if-test hierarchy pseudo-code):

```
if BF:
    Calculate new wavefunction (fermion = slater, boson = bosonic)
else if IS:
    if Numerical Kinetics:
        Calculate new wavefunction

    else if Closed Form Kinetics:
        Get new gradients

        if fermions:
            Calculate slater determinant ratio
            Update the inverse matrix
        else if bosons:
            Calculate bosonic wavefunction ratio
```

Creating this sort of general code without the use of virtual functions is a complete mess of if-tests, whereas object orientation cleans up most of the mess, abstracting the implementation to easily read statements. On top of this, everything itself depends on the choice of single particle basis, which leads us to the next constraint.

Constraint (iv)

Not only do we want a given single particle basis, but the constituents of this basis should themselves be allowed to have any form (i.e. be a superposition of another basis). The way this is handled in the code is to have the single particle basis listed as an array or `function` objects. The purpose of this object is nothing but being initialized and evaluated. An example is the ground state of the harmonic oscillator:

```
Orbitals::Orbitals(int n_p, int dim) {
    this->n_p = n_p;
    this->dim = dim;

    int max_implemented = 15; //for 30 particles
    basis_functions = new function*[max_implemented];
}

...


oscillator_basis::oscillator_basis(int n_p, int dim, double alpha, double w)
: Orbitals(n_p, dim) {

    ...

    basis_functions[0] = new HO_1(this->alpha, w);
    basis_functions[1] = new HO_2(this->alpha, w);
    basis_functions[2] = new HO_3(this->alpha, w);
    ...
}

...

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

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

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

...

class HO_1 : public function {
protected:
    double* alpha;
    double w;

public:

    HO_1(double* alpha, double w);

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

...

double HO_1::eval(const Walker* walker, int i) const {
    double r2 = walker->get_r_i2(i);
```

```
26
27      double x = walker->r(i, 0);
28      double y = walker->r(i, 1);
29
30      double H = 1;
31
32      return H * exp(-0.5 * (*alpha) * w * r2);
33  }
```

All orbital files are generated automatically by a Python-script. The reason why, in this case, `alpha` is a pointer, is so that the value can be changed within orbitals, and then consequently be changed within the function class (a must during minimization).

However, we can abstract it even more. Implementing a single particle basis expanded in e.g. harmonic oscillator basis functions is more or less trivial in this framework. Following is a pseudo code to illustrate this claim:

```
1  class Expanded_Functions : public functions{
2  public:
3      Expanded_Functions(int m, file_spesifics);
4      virtual double eval(const Walker* walker, int i) const;
5
6  protected:
7      int m;
8
9      double* coeffs;
10     function** expanded_basis;
11
12 };
13
14 Expanded_Functions::Expanded_Functions(int m, file_spesifics){
15     this->m = m;
16     coeffs = new double[m];
17     expanded_basis = new function*[m];
18
19     expanded_basis[0] = new HO_1(alpha=1, w);
20   ...
21
22
23     //Load coeffs from file e.g. from Hartree Fock runs
24 }
25
26 double Expanded_Functions::eval(const Walker* walker, int i) const {
27
28     double value = 0;
29     for (int i = 0; i < m; i++){
30         value += coeffs[i]*expanded_basis[i]->eval(walker, i);
31     }
32
33     return value;
34
35 }
```

Constraint (v)

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. Having great flexibility in both the Hamiltonian and the basis functions means the code is easily adaptable to other systems.

The flexibility in the kinetic term has already been described. The way the code loads potential terms is through a `System` function

```
void System::add_potential(Potential* pot) {
    potentials.push_back(pot);
}
```

where `potentials` is a vector of potential pointers. When we extract the potential energy term, we simply iterate over the pointers in the array and call a member function

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

    for (std::vector<Potential*>::iterator pot = potentials.begin(); pot != potentials.
        end(); ++pot) {
        potE += (*pot)->get_pot_E(walker);
    }

    return potE;
}
```

Implementing new potentials is also extremely simple; just have the constructor take reasonable parameters, and implement the expression. All in all the code servers great flexibility on all parts without suffering from if-tests or constituting of several unlinked parts. It takes way longer to develop such a code, but it all pays off when it comes to later implementations or extensions of the library.

3.1.3 RAM optimizations

VMC uses close to zero memory (RAM); 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 byte) and integers (4 byte). 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` matrices	8 `n_p` · `n_p` bytes
•1 `n_p` × `n_p/2` matrix	4 `n_p` · `n_p` bytes
•1 array of length `n_p`	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 3.1.3 we see how the RAM usage per walker 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 is therefore 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

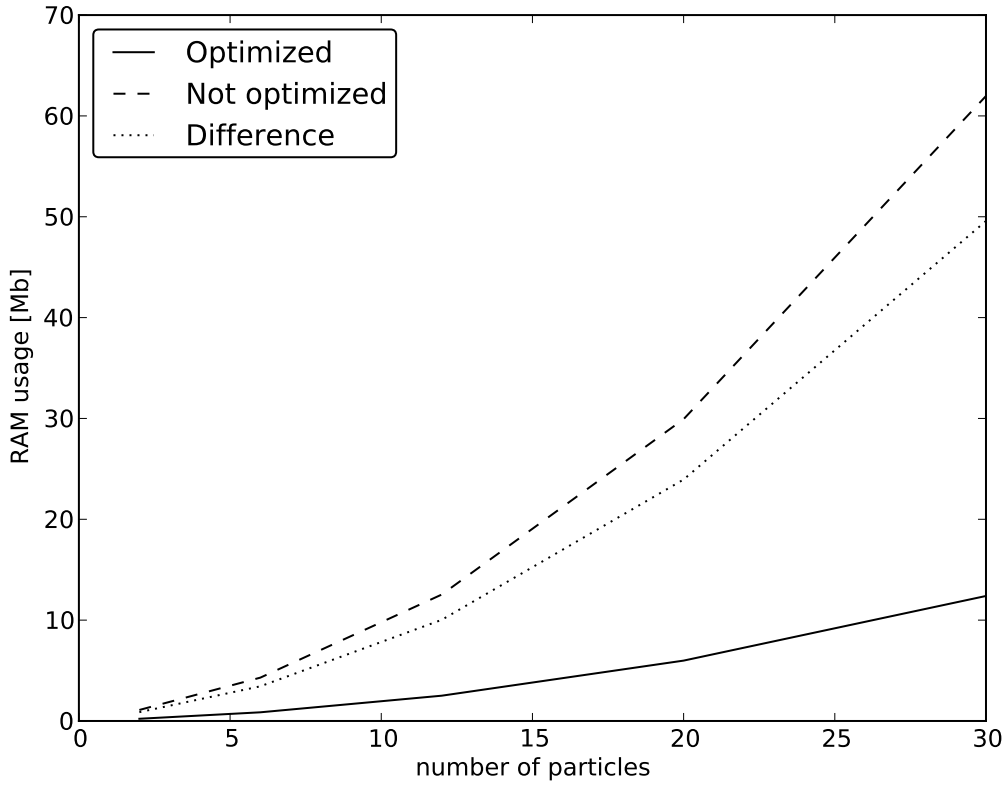

Figure 3.1: Number of particles vs. the theoretical RAM usage per walker for the optimized and unoptimized case. Calculated for two dimensions, 1000 active walkers and 5000 inactive.

less unoptimized in this part.

3.1.4 CPU-time Optimization

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

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

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

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

$$\begin{aligned} (E_k)_L &= -\frac{1}{2}\frac{1}{\psi_T}\nabla_i^2\psi_T \\ &= -\frac{1}{2}\frac{1}{|S|\psi_C}\nabla_i^2(|S|\psi_C) \\ &= -\frac{1}{2}\frac{1}{|S|\psi_C}\nabla_i\left(\psi_C\nabla_i|S| + |S|\nabla_i\psi_C\right) \\ &= -\frac{1}{2}\Big[\frac{1}{\psi_C}\nabla_i^2\psi_C + \frac{2}{|S|\psi_C}\nabla_i|S|\cdot\nabla_i\psi_C \\ &\qquad + \frac{1}{|S|}\nabla_i^2|S|\Big]. \end{aligned} \tag{3.1}$$

$$\begin{aligned} \mathbf{F}_i &= \frac{2}{|S|\psi_C}\nabla_i(|S|\psi_C) \\ &= 2\left(\frac{1}{\psi_C}\nabla_i\psi_C + \frac{1}{|S|}\nabla_i|S|\right), \end{aligned} \tag{3.2}$$

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[1]:

$$\begin{aligned}\frac{1}{|S|}\nabla_i|S| &= \sum_{j=1}^{n}\nabla_i\phi_j(\vec{r}_i)S^{-1}_{ji} \\ \frac{1}{|S|}\nabla_i^2|S| &= \sum_{j=1}^{n}\nabla_i^2\phi_j(\vec{r}_i)S^{-1}_{ji},\end{aligned} \tag{3.3}$$

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[**?**]. This means that no explicit wave function calculation is needed (we can calculate the ratio between two Jastrow factors without problem). The expression for the ratio in terms of the inverse is

$$R_S = \sum_{j=1}^{n}\phi_j(\vec{r}_{i,\text{new}})S^{-1}_{ji}(\vec{r}_{\text{old}}), \tag{3.4}$$

where i is the particle being moved.

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

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

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

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

$$S^{-1}_{kj}(\vec{r}_{\text{new}}) = \begin{cases} S^{-1}_{kj}(\vec{r}_{\text{old}}) - \frac{1}{R_S}S^{-1}_{ki}(\vec{r}_{\text{old}})\sum_{l=1}^{n}\phi_l(\vec{r}_{i,\text{ new}})S^{-1}_{lj}(\vec{r}_{\text{old}}) & j \neq i \\ \frac{1}{R_S}S^{-1}_{ki}(\vec{r}_{\text{old}}) & j = i \end{cases} \tag{3.5}$$

Equal expressions like the the ones listed in Eq. (3.4) can be found for the case of the Pade-Jastrow factor as well:

$$\begin{aligned}\frac{1}{\psi^{PJ}_C}\nabla_i\psi^{PJ}_C &= \sum_{j\neq i}\frac{\vec{r}_{ij}}{r_{ij}}\frac{a_{ij}}{(1+\beta r_{ij})^2} \\ \frac{1}{\psi^{PJ}_C}\nabla_i^2\psi^{PJ}_C &= \left|\frac{1}{\psi^{PJ}_C}\nabla_i\psi^{PJ}_C\right|^2 + 2\sum_{i<j}\frac{a_{ij}(1-\beta r_{ij})}{r_{ij}(1+\beta r_{ij})^3}.\end{aligned} \tag{3.6}$$

Notice that a_{ij} is written as a matrix element. If we were to calculate a for every single run in the loop, the double if-tests would drain a lot of CPU time. Prior to the sampling, in the `Jastrow::initialize`

function, the values of a are generated once and for all and stored in a matrix. This matrix remains unchanged throughout the entire sampling, and no if-tests are necessary.

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

$$\frac{1}{\psi_T}\frac{\partial \psi_T}{\partial \alpha} = -\frac{1}{2}\omega\sum_{i=1}^{N} r_i^2, \tag{3.7}$$

$$\frac{1}{\psi_T}\frac{\partial \psi_T}{\partial \beta} = -\sum_{i<j}\frac{a_{ij}r_{ij}^2}{(1+\beta r_{ij})^2}. \tag{3.8}$$

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

Optimization (without approximations) is all about not calculating more than you have to. Calculating gradients is the part of the code which require the most CPU time. The most time consuming functions is the gradients. Looking at Eq. (3.4) 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.

3.2 Validation

things should not be wrong. It is bad.

Chapter 4

Results

4.1 Validating the code

4.1.1 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

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

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

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[3].

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

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

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

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

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

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

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

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

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

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

Bibliography

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

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

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