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 Dale and Sverdrup**Need specifics and citations**. It quickly became the state-of-the-art in programming, and is today used throughout the world. It is brilliant in the way that it ties our everyday intuition into the programming language. To illustrate this, let us look at a simple 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
```

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

tion, 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.

```
1  double QMC::calculate_local_energy(Walker* walker) const {
2      return kinetics->get_KE(walker) + system->get_potential_energy(walker);
3  }
```

```
1  class Walker {
2  ...
3
4      mat r;
5  ...
6  };
```

```
1  class Kinetics {
2  ...
3
4      virtual double get_KE(const Walker* walker) = 0;
5  ...
6  };
7
8  class Numerical : public Kinetics {
9  ...
10
11     virtual double get_KE(const Walker* walker);
12 ...
13 };
```

```
1  class Potential {
2  ...
3
4      virtual double get_pot_E(const Walker* walker) const = 0;
5  ...
6  };
7
8  class Harmonic_osc : public Potential {
9  ...
10
```

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

```
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

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

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.

Part II

Results

Chapter 3

Results

3.1 Validating the code

3.1.1 Calculation for non-interacting particles

[1]

Bibliography

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