Analysis and design process for QUANTAPLUS package

Mustafa Ahmed Ali Ahmed,
Mohammed Maher Abdelrahim Mohammed

May 28, 2022

The process can be undertaken in five phases, and a phase 0 that is just the initial commitment to using some kind of structure.

1 Phase 0: Our plan

It is better to sketch our plan later, after fixing the following sections.

1.1 The mission statement

Donald E. Knuth once said: "The best programs are written so that computing machines can perform them quickly and human beings can understand them clearly". Following this philosophy, our mission statement is to build a descriptive C++ package dictated to provide the user (client) with useful tools (quite easy to be used) to handle the numerical calculations necessary to study and visualize different quantum-mechanical problems in an efficient way. We will revisit most of the numerical problems encountered in introductory quantum mechanics.

2 Phase 1: What are we making?

It's necessary to stay focused on the heart of what we're trying to accomplish in this phase: determine what the system (or our package) is supposed to do. It's useful to identify the key features in the system that will reveal some of the fundamental classes we'll be using. These are essentially descriptive answers to the following questions:

- **Who will use this system?**
  We are targeting in the first place students, teachers and researchers. as well as anyone has a basic introductory knowledge about quantum mechanics.
- **What can those actors do with the system?**
  Preform numerical calculations for a wide classes of quantum mechanics

problems, ranging from the simple issues (such as the matrix representations of wave functions and operators, commuting observables, Commutator algebra-physical significance ,and unitary transformations) to the more complicated calculations within atomic and condensed-matter physics.

- **How does this actor do that with this system?**
  Following the well documented QuantaPlus user guide, the actor can directly implement and use the package classes, which are designed to be written in the same (plain-English) language and terminologies used within Quantum Mechanics context.

- **How else might this work if someone else were doing this, or if the same actor had a different objective? (to reveal variations)**
  Obviously, if the user is not armed with the basics knowledge about QM, he/she defiantly will commit some errors prevent him/her form compile their code, or even get some runtime-errors.

- **What problems might happen while doing this with the system? (to reveal exceptions)**
  For example if someone try to assign $n \times 1\,|ket\rangle$ vector to $n \times m$ Quantum Operator (matrix), will lead to compilation error.

  Listing 1: error example

  ```
  //3 components complex ket vector in Hilbert space
  QuantaPlus::ket<std::complex> A(3);
  //3x3 complex Matrix Operator
  QuantaPlus::QM_operator<std::complex> O(3,3);
  //compilation error, since they are physically not the same object
  O = A;
  ```

  Also any attempt to do matrix multiplication violating the usual rule (two matrices can only be multiplied when the number of columns of the first matrix is equal to the number of rows of the second matrix), will throw an exception, as a result that will terminate the program.

3 Phase 2: How will we build it?

In this phase we must come up with a design that describes what the classes look like and how they will interact.

3.1 ket and bra classes

Ket and bra are public derived classes from **Eigen 3** well designed Eigen::Matrix class. Ket vectors represent matrices with one column and Dynamic-size rows, while bra vectors are matrices with one row, and Dynamic-size column.[1]

[1]Dynamic-size means that the numbers of rows or columns are not necessarily known at compile-time. In this case they are runtime variables, and the array of coefficients is allocated dynamically on the heap.

The "responsibilities" of both ket and bra classes: Acting as containers for the quantum-mechanical system states, represented by unit vectors residing in a complex separable Hilbert space.

Listing 2: the ket and bra classes

```
template <class T>
class QuantaPlus::ket : public Eigen::Matrix<T,Eigen::Dynamic,1>
template <class T>
class QuantaPlus::bra : public Eigen::Matrix<T,1, Eigen::Dynamic>
```

3.1.1 Properties of kets, bras, and bra-kets

- Kets: elements of a vector space Dirac denoted the state vector ψ by the symbol $|\psi\rangle$, which he called a ket vector, or simply a ket.

```
QuantaPlus::ket<std::complex> ket_psi;
```

- Bras: we know from linear algebra that a dual space can be associated with every vector space. Dirac denoted the elements of a dual space by the symbol $\langle|$, for example the element $\langle\psi|$ represents a bra.

```
QuantaPlus::bra<std::complex> bra_psi;
```

- Every ket has a corresponding bra, and vice versa (they are dual-conjugate of each other): obtaining one from the other are doable making use of the overloaded QuantaPlus::DualConj() function.

Listing 3: the ket and bra dual Conjugate functions

```
template <typename T>
ket<T> DualConj(const bra<T> &b)
template <typename T>
bra<T> DualConj(const ket<T> &b)
```

- In quantum mechanics, since the scalar product is a complex number given by the QuantaPlus::BraKet() function.

Listing 4: Bra-ket: Dirac notation for the scalar product

```
template <typename T>
T BraKet(const bra<T> &b, const ket<T> &kt)
```

The “collaborations” of the bra and ket classes: since they are subclasses of Eigen::Matrix, they’re normally interacting with the all **Eigen 3** library template classes and methods, with all other properties.

3.2 QM_operator class

QuantaPlus::QM_operator is a subclass of Eigen::Matrix, represents square matrices with Dynamic-size rows and column, and acts as super-class for QuantaPlus::Angular_Momentum one.

Listing 5: QM_operator class

```
template<class T>
class QuantaPlus::QM_operator: public
    Eigen::Matrix<T,Eigen::Dynamic,Eigen::Dynamic>
```

The "collaborations" of the QM_operator class: since they are subclasses of Eigen::Matrix, they're normally interacting with the all **Eigen 3** library template classes and methods, with all other properties. Moreover interacting with bra and ket

3.3 Angular_Momentum class

QuantaPlus::Angular_Momentum public derived from the QuantaPlus::QM_operator.

Listing 6: Angular_Momentum class

```
template <class T>
class QuantaPlus::Angular_Momentum : public QuantaPlus::QM_operator<T>
```

Listing 7: Angular_Momentum public methods

```
Angular_Momentum<T>::Angular_Momentum_OperatorJSqr(const double& );
Angular_Momentum<T>::Angular_Momentum_OperatorJx(const double& );
Angular_Momentum<T>::Angular_Momentum_OperatorJy(const double& );
Angular_Momentum<T>::Angular_Momentum_OperatorJz(const double& );
Angular_Momentum<T>::Angular_Momentum_OperatorJPlus(const double& );
Angular_Momentum<T>::Angular_Momentum_OperatorJMinus(const double& );
Angular_Momentum<T>::RotationByAngle(const double & );
```

The "collaborations" of the Angular_Momentum class: since they are subclasses of Eigen::Matrix, they're normally interacting with the all **Eigen 3** library template classes and methods, with all other properties. Moreover interacting with bra, ket and any QM_operato

4 Phase 3: Build the core

This is the initial conversion from the rough design into a compiling and executing body of code that can be tested, and especially that will prove or disprove your architecture. This is not a one-pass process, but rather the beginning of a series of steps that will iteratively build the system, as you'll see in phase 4. Our goal is to find the core of your system architecture that needs to be implemented in order to generate a running system, no matter how incomplete that system is in this initial pass. You're creating a framework that you can build upon with further iterations. We're also performing the first of many system integrations and tests, and giving the stakeholders feedback about what their system will look like and how it is progressing. Ideally, you are also exposing some of the critical risks. You'll probably also discover changes and improvements that can be made to your original architecture things you would not have learned without implementing the system.

Part of building the system is the reality check that you get from testing against your requirements analysis and system specification (in whatever form they exist). Make sure that your tests verify the requirements and use cases. When the core of the system is stable, you're ready to move on and add more functionality.

5 Phase 4: Iterate the use cases

Once the core framework is running, each feature set we add is a small project in itself. We add a feature set during an iteration, a reasonably short period of development. How big is an iteration? Ideally, each iteration lasts one to three weeks (this can vary based on the implementation language). At the end of that period, we have an integrated, tested system with more functionality than it had before.

6 Phase 5: Evolution

Evolution occurs when we build the system, see that it matches our requirements, and then discover it wasn't actually what you wanted. When we see the system in operation, we find that we really wanted to solve a different problem. If we think this kind of evolution is going to happen, then we owe it to ourselves to build our first version as quickly as possible so we can find out if it is indeed what you want.