Quantum++ for the impatient

Author: Vlad Gheorghiu
vgheorgh@gmail.com

Date: November 26, 2014

Contents

List of source codes

1 Introduction

Quantum++ is a C++11 general purpose quantum computing simulator, composed solely of of header files, and uses the Eigen linear algebra library. The simulator defines a large collection of (template) quantum computing related functions and few useful classes. The main data types are complex vectors and complex matrices, as I will describe below. Most functions operate on such vectors/matrices, and *always* return the result by value. Collection of objects are implemented via the standard library container `std::vector<>`, specialized accordingly. Ease of use and performance were among the most important design factors.

2 Installation

To get started with Quantum++, first download the Eigen library from http://eigen.tuxfamily.org and unzip it into the home folder[1], as `$HOME/eigen`. You can change the name of the folder, but in the current document I will use `$HOME/eigen` as the location of Eigen library. Next, download the Quantum++ library from http://vsoftco.github.io/qpp/ and unzip it into the home folder as `$HOME/qpp`. Finally, make sure that your compiler supports C++11 and preferably OpenMP. I recommend `g++`, version 4.8 or later. You are now ready to go!

We now build a simple minimal example to test that the installation was successful. Create a folder called `$HOME/example`, and inside it create the file `example.cpp`, with the content listed in the Listing 1. Next compile the file using a C++11 compliant compiler such as `g++` version 4.8 or later. From inside the

```
#include "qpp.h"

int main()
{
    std::cout << "Hello Quantum++" << std::endl;
}
```

Listing 1: Minimal example

folder `$HOME/example`, type

```
g++ -std=c++11 -isystem$HOME/eigen -isystem$HOME/qpp/include example.cpp -o example
```

Your compile command may differ from the above, depending on the name of your C++ compiler and operating system. If everything went fine, then the above command builds the executable `example` in the current folder. To run it, type `./example` from inside the `$HOME/example` folder. The output should look like

```
>>> Starting Quantum++...
>>> Thu Nov 20 12:00:26 2014

Hello Quantum++

>>> Exiting Quantum++...
>>> Thu Nov 20 12:00:26 2014
```

Congratulations, everything seems to work fine!

3 Basic usage

3.1 Data types

All functions, classes and global objects defined by the library lie inside the `namespace qpp`. To avoid additional typing, I will omit the prefix `qpp::` in the rest of this document. I recommend the using directive

```
using namespace qpp;
```

in your main `.cpp` file.

The most important data types are defined via typedefs in `types.h`[2] (inside the `include` folder of the Quantum++ source distribution). We list them below.

[1] I implicitly assume from now on that you use a UNIX-based system, although everything should translate into Windows as well, with slight modifications

[2] All necessary Quantum++ header files, together with other important system headers, such as `<iostream>`, `<cmath>` etc., are automatically included in `qpp.h`, hence most of the time you should only include `qpp.h` in the main `.cpp` file.

- `cplx` – Complex number, alias for `std::complex<double>`
- `idx` – Index (non-negative integer), alias for `std::size_t`
- `cmat` – Complex dynamic matrices, alias for `Eigen::MatrixXcd`
- `dmat` – Double dynamic matrices, alias for `Eigen::MatrixXd`
- `ket` – Complex dynamic column vector, alias for `Eigen::VectorXcd`
- `bra` – Complex dynamic row vector, alias for `Eigen::RowVectorXcd`
- `dyn_mat<Scalar>` – Dynamic matrix template alias over the field `Scalar`, alias for `Eigen::Matrix<Scalar, Eigen::Dynamic, Eigen::Dynamic>`
- `dyn_col_vect<Scalar>` – Dynamic column vector template alias over the field `Scalar`, alias for `Eigen::Matrix<Scalar, Eigen::Dynamic, 1>`
- `dyn_row_vect<Scalar>` – Dynamic row vector template alias over the field `Scalar`, alias for `Eigen::Matrix<Scalar, 1, Eigen::Dynamic>`

3.2 Constants

The important constants are defined in the file `constants.h` as follows.

- `constexpr double pi = 3.1415...;` – π
- `constexpr double ee = 2.7182...;` – e, base of natural logarithms
- `constexpr idx infty = -1;` – Infinity
- `constexpr idx maxn = 64;` – Maximum number of allowed qu(d)its (subsystems)
- `constexpr double eps = 1e-12;` – Used in comparing floating point values to zero
- `constexpr double chop = 1e-10;` – Used in display manipulators to set to zero numbers which are in absolute values smaller than `chop`
- `constexpr cplx operator""_i(unsigned long long int x)` – User-defined literal for the imaginary number $i := \sqrt{-1}$
- `constexpr cplx operator""_i(unsigned long double int x)` – User-defined literal for the imaginary number $i := \sqrt{-1}$
- `cplx omega(idx D)` – D-th root of unity

3.3 Global singleton classes

Some useful classes are defined as singletons, initialized at runtime, as follows.

- `class qpp::Gates gt;` – Quantum gates
- `class qpp::States` – Quantum states/projectors

Most important gates and states are defined via the singleton classes

`Gates`

and

`States`

and are initialized automatically by the library on entry, before the starting of `int main()`. They are defined in the files `classes/Gates.h` and `classes/States.h`, respectively. The singleton instances `Gates gt;` and `States st;` are defined inside `qpp.h`, and are globally available.

3.4 Simple examples

3.4.1 Gates and states

We are now ready for a more sophisticated example that actually does something useful. Consider the code in Listing 2.

```
1  #include "qpp.h"
2  using namespace qpp;
3
4  int main()
5  {
6      ket psi = mket({1,0});
7      cmat U = gt.CNOTab;
8      ket result = U * psi;
9
10     std::cout << "The result of applying the Controlled-NOT gate CNOTab on |10> is:\n";
11     std::cout << disp(result) << std::endl;
12
13     ket phi = st.z0;
14     U = gt.X;
15     result = U * phi;
16
17     std::cout << "The result of applying the bit-flip gate X on |0> is:\n";
18     std::cout << disp(result) << std::endl;
19 }
```

Listing 2: Gates and states

which outputs

```
>>> Starting Quantum++...
>>> Thu Nov 20 13:30:09 2014

The result of applying the Controlled-NOT gate CNOTab on |10> is:
     0
     0
     0
1.0000
The result of applying the bit-flip gate X on |0> is:
     0
1.0000

>>> Exiting Quantum++...
>>> Thu Nov 20 13:30:09 2014
```

In line 2, we bring the namespace `qpp` into the global namespace. In line 6 we declare a state vector, and assign to it the value $|10\rangle$ via the function `mket()`. We could have also used an Eigen-like equivalent syntax

```
ket psi(2); // must specify the dimension before insertion of elements via <<
psi << 0, 0, 1, 0;
```

however the `mket()` function is more convenient. In line 7 we declare a gate `U` as the Controlled-NOT with control as the first subsystem, and target as the last, using the global singleton `gt`. In line 8 we declare the ket `result` as the result of applying the Controlled-NOT gate to the state $|10\rangle$, i.e. $|11\rangle$.

We then display the result of the computation in lines 10 and 11. In line 11 we use the format manipulator `disp()`, which is especially useful when displaying complex matrices, as it displays the entries of the latter in the form $a + bi$, in contrast to the form (a, b) used by the C++ standard library. The manipulator also accepts additional parameters that allows e.g. setting to zero numbers smaller than some given value (useful to chop small values), and it is in addition overloaded for standard containers, iterators and C-style arrays.

Next, in line 13 we use the singleton `st` to declare `phi` as the zero eigenvector $|0\rangle$ of the Z Pauli operator. In line 14 we assign to the gate `U` the bit flip gate `gt.X`, compute the result of the operation $X|0\rangle$ in line 15, and finally display the result $|1\rangle$ in lines 17 and 18.

3.4.2 Measurements

Let us now complicate things a bit and introduce measurements. Consider the example in Listing 3. A

```
1  #include "qpp.h"
2  using namespace qpp;
3
4  int main()
5  {
6      ket psi = mket({0,0});
7      cmat U = gt.CNOTab * kron(gt.H, gt.Id2);
8      ket result = U * psi; // we have the Bell state (|00>+|11>)/sqrt(2)
9
10     std::cout << "We just produced the Bell state:\n";
11     std::cout << disp(result) << std::endl;
12
13     // apply a bit flip on the second qubit
14     result = apply(result, gt.X, {1}); // we produced (|01>+|10>)/sqrt(2)
15     std::cout << "We produced the Bell state:\n";
16     std::cout << disp(result) << std::endl;
17
18     // measure the first qubit in the X basis
19     auto m = measure(result, gt.H, {0});
20     std::cout << "Measurement result: " << std::get<0>(m);
21     std::cout << std::endl << "Probabilities: ";
22     std::cout << disp(std::get<1>(m),", ") << std::endl;
23     std::cout << "Resulting states: " << std::endl;
24     for(auto&& elem: std::get<2>(m))
25             std::cout << disp(elem) << std::endl << std::endl;
26 }
```

Listing 3: Measurements

possible output of this program is

```
>>> Starting Quantum++...
>>> Thu Nov 20 15:39:17 2014

We produced the Bell state:
0.7071
     0
     0
0.7071
We produced the Bell state:
```

```
     0
0.7071
0.7071
     0
Measurement result: 1
Probabilities: [0.5000, 0.5000]
Resulting states:
0.5000   0.5000
0.5000   0.5000

 0.5000   -0.5000
-0.5000    0.5000


>>> Exiting Quantum++...
>>> Thu Nov 20 15:39:17 2014
```

In line 7, we use the function `kron()` to create the tensor product (Kronecker product) of the Hadamard gate on the first qubit and identity on the second qubit. In line 8 we compute the result of the operation $CNOT_{ab}(H \otimes I)|00\rangle$, which is the Bell state $(|00\rangle + |11\rangle)/\sqrt{2}$. We display it in lines 10 and 11.

In line 14 we use the function `apply()` to apply the gate X on the second qubit[3] of the previously produced Bell state. The function `apply()` takes as its third parameter a list of subsystems, and in our case `{1}` denotes the *second* subsystem, not the first. The function `apply()`, as well as many other functions that we will encounter, have a variety of useful overloads, see `doc/refman.pdf` for a detailed library reference. In lines 15 and 16 we display the newly created Bell state.

In line 19 we use the function `measure()` to perform a measurement of the first qubit (subsystem `{0}`) in the X basis. You may be confused by the apparition of `gt.H`, however this overload of the function `measure()` takes as its second parameter the measurement basis, specified as the columns of a complex matrix. In our case, the eigenvectors of the X operator are just the columns of the Hadamard matrix. As mentioned before, as all other library functions, `measure()` returns by value, hence it does not modify its argument. The return of `measure` is a tuple consisting of the measurement result, the outcome probabilities, and the possible output states. Technically `measure()` returns a

```
std::tuple<std::size_t, std::vector<double>, std::vector<cmat>>
```

Instead of using this long type definition, we use the new C++11 `auto` keyword to define the type of the result `m` of `measure()`. In lines 20–25 we use the standard `std::get<>` function to retrieve each element of the tuple, then display the measurement result, the probabilities and the resulting output states.

3.4.3 Quantum operations

4 Brief description of Quantum++ files

5 Advanced topics

5.1 Exceptions

5.2 Aliasing

5.3 Optimizations

5.4 Extending Quantum++

[3] Quantum++ uses the C/C++ numbering convention, with indexes starting from zero.