Quantum++ for the impatient

Author: Vlad Gheorghiu
vgheorgh@gmail.com

Date: December 2, 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 directory[1], as `$HOME/eigen`. You can change the name of the directory, but in the current document I will use `$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 directory 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 next build a simple minimal example to test that the installation was successful. Create a directory called `$HOME/qpp_examples`, and inside it create the file `ex1.cpp`, with the content listed in the Listing 1. A verbatim copy of Listing 1 is also available at `$HOME/qpp/examples/ex1.cpp`.

```cpp
// Minimal example
// Source: ./examples/ex1.cpp
#include <qpp.h>

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

Listing 1: Minimal example

Next compile the file using a C++11 compliant compiler such as `g++` version 4.8 or later. From inside the directory `$HOME/qpp_examples`, type

```
g++ -std=c++11 -isystem $HOME/eigen -I $HOME/qpp/include ex1.cpp -o ex1
```

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 `ex1` in the directory `$HOME/qpp_examples`. To run it, type `./ex1` from inside the directory `$HOME/qpp_examples`. The output should look like

```
>>> Starting Quantum++...
>>> Tue Dec  2 14:34:07 2014

Hello Quantum++!

>>> Exiting Quantum++...
>>> Tue Dec  2 14:34:07 2014
```

Congratulations, everything seems to be working!

3 Data types, constants and global objects

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

```cpp
using namespace qpp;
```

in your main `.cpp` file.

[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

3.1 Data types

The most important data types are defined via typedefs in the header file types.h[2] (inside the include directory of the Quantum++ source distribution). We list them in Table 1.

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>

Table 1: User-defined data types

3.2 Constants

The important constants are defined in the header file constants.h and are listed in Table 2.

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 numbers to zero
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 $e^{2\pi i/D}$

Table 2: User-defined constants

3.3 Global singleton classes and instances

Some useful classes are defined as singletons, are globally available, and are initialized at runtime in the header file qpp.h, before the starting of int main(), as listed in Table 3.

4 Simple examples

All examples of this section are copied verbatim from the directory ./examples and compiled successfully. Each listing corresponds to a unique example, e.g. Listing 2 corresponds to ./examples/ex2.cpp. For

[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.

`const Init& init = Init::get_instance();`	Library initialization (welcome messages etc.)
`const Codes& codes = Codes::get_instance();`	Quantum error correcting codes
`const Gates& gt = Gates::get_instance();`	Quantum gates
`const States& st = States::get_instance();`	Quantum states
`RandomDevices& rdevs = RandomDevices::get_instance()`	Random number generator engines

Table 3: Global singleton classes and instances

convenience, the location of the source file is also displayed in the first line of each example, as a C++ comment.

The examples are simple and demonstrate the main feature of Quantum++. They cover only a small part of library functions, but enough to get the interested user started. For extensive about all library functions, including various overloads, the user should consult the complete reference located at `./doc/refman.pdf`. A more comprehensive (but also more complicated) example, that consists of a collection of quantum information processing routines, is located at `./examples/example.cpp`.

4.1 Gates and states

We introduce the main objects used by Quantum++: gates, states and basic operations. Consider the code in Listing 2, which may output

```
>>> Starting Quantum++...
>>> Tue Dec  2 14:34:07 2014

The result of applying the bit-flip gate X on |0> is:
     0
1.0000
The result of applying the gate CNOTab on |10> is:
     0
     0
     0
1.0000
Generating the random one-qubit gate U:
-0.9557 + 0.1186i  0.2089 + 0.1703i
 0.2503 + 0.0999i  0.3603 + 0.8931i
The result of applying the Controlled-U gate on |10> is:
               0
               0
-0.9557 + 0.1186i
 0.2503 + 0.0999i

>>> Exiting Quantum++...
>>> Tue Dec  2 14:34:07 2014
```

In line 2, we bring the namespace `qpp` into the global namespace.

In line 7 we use the singleton `st` to declare `psi` as the zero eigenvector $|0\rangle$ of the Z Pauli operator. In line 8 we assign to the gate `U` the bit flip gate `gt.X`, compute the result of the operation $X|0\rangle$ in line 9, and display the result $|1\rangle$ in lines 11 and 12. In line 12 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

```
1  // Gates and states
2  // Source: ./examples/ex2.cpp
3  #include <qpp.h>
4  using namespace qpp;
5
6  int main()
7  {
8      ket psi = st.z0; // |0> state
9      cmat U = gt.X;
10     ket result = U * psi;
11
12     std::cout << "The result of applying the bit-flip gate X on |0> is:\n";
13     std::cout << disp(result) << std::endl;
14
15     psi = mket({1, 0}); // |10> state
16     U = gt.CNOTab; // Controlled-NOT
17     result = U * psi;
18
19     std::cout << "The result of applying the gate CNOTab on |10> is:\n";
20     std::cout << disp(result) << std::endl;
21
22     U = randU(2);
23     std::cout << "Generating a random one-qubit gate U:\n";
24     std::cout << disp(U) << std::endl;
25
26     result = applyCTRL(psi, U, {0}, {1});
27     std::cout << "The result of applying the Controlled-U gate on |10> is:\n";
28     std::cout << disp(result) << std::endl;
29 }
```

Listing 2: Gates and states

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.

In line 14 we reassign to `psi` the state $|10\rangle$ via the function `mket()`. We could have also used the Eigen insertion operator

```
ket psi(4); // must specify the dimension before insertion of elements via <<
psi << 0, 0, 1, 0;
```

however the `mket()` function is more concise. In line 15 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 16 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 18 and 19.

Next, in line 21 we generate a random unitary gate via the function `randU()`, then in line 25 apply the Controlled-U, with control as the first qubit and target as the second qubit, to the state `psi`. Finally, we display the result in lines 26 and 27.

4.2 Measurements

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

```cpp
// Measurements
// Source: ./examples/ex3.cpp
#include <qpp.h>
using namespace qpp;

int main()
{
    ket psi = mket({0, 0});
    cmat U = gt.CNOTab * kron(gt.H, gt.Id2);
    ket result = U * psi; // we have the Bell state (|00>+|11>)/sqrt(2)

    std::cout << "We just produced the Bell state:\n";
    std::cout << disp(result) << std::endl;

    // apply a bit flip on the second qubit
    result = apply(result, gt.X, {1}); // we produced (|01>+|10>)/sqrt(2)
    std::cout << "We produced the Bell state:\n";
    std::cout << disp(result) << std::endl;

    // measure the first qubit in the X basis
    auto m = measure(result, gt.H, {0});
    std::cout << "Measurement result: " << std::get<0>(m);
    std::cout << std::endl << "Probabilities: ";
    std::cout << disp(std::get<1>(m), ", ") << std::endl;
    std::cout << "Resulting states: " << std::endl;
    for (auto && elem : std::get<2>(m))
        std::cout << disp(elem) << std::endl << std::endl;
}
```

Listing 3: Measurements

```
>>> Starting Quantum++...
>>> Tue Dec  2 14:34:07 2014

We just 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++...
>>> Tue Dec  2 14:34:07 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.

4.3 Quantum operations

In Listing 4 we introduce quantum operations: quantum channels, as well as the partial trace and partial transpose operations. The output of this program is

```
>>> Starting Quantum++...
>>> Tue Dec  2 14:34:07 2014

Initial state:
0.5000   0   0   0.5000
     0   0   0        0
     0   0   0        0
0.5000   0   0   0.5000
Eigenvalues of the partial transpose of Bell-0 state are:
-0.5000   0.5000   0.5000   0.5000
Measurement channel with 2 Kraus operators:
1.0000   0
     0   0
    and
0        0
0   1.0000
Superoperator matrix of the channel:
```

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

```
1.0000   0   0        0
     0   0   0        0
     0   0   0        0
     0   0   0   1.0000
Choi matrix of the channel:
1.0000   0   0        0
     0   0   0        0
     0   0   0        0
     0   0   0   1.0000
After applying the measurement channel on the first qubit:
0.5000   0   0        0
     0   0   0        0
     0   0   0        0
     0   0   0   0.5000
After partially tracing down the second subsystem:
0.5000        0
     0   0.5000

>>> Exiting Quantum++...
>>> Tue Dec  2 14:34:07 2014
```

The example should by now be self-explanatory.

In line 7 we define the input state `rho` as the projector onto the Bell state $(|00\rangle + |11\rangle)/\sqrt{2}$, then display it in lines 8 and 9.

In lines 12–14 we partially transpose the first qubit, then display the eigenvalues of the resulting matrix `rhoTA`.

In lines 16–18 we define a quantum channel `Ks` consisting of two Kraus operators: $|0\rangle\langle 0|$ and $|1\rangle\langle 1|$, then display the latter. Note that Quantum++ uses the `std::vector<cmat>` container to store the Kraus operators and define a quantum channel.

In lines 20–24 we display the superoperator matrix as well as the Choi matrix of the channel `Ks`.

Next, in lines 27–29 we apply the channel `Ks` to the first qubit of the input state `rho`, then display the output state `rhoOut`.

Finally, in lines 32–34 we take the partial trace of the output state `rhoOut`, then display the resulting state `rhoA`.

5 Brief description of Quantum++ files

6 Advanced topics

6.1 Exceptions

6.2 Aliasing

6.3 Optimizations

6.4 Extending Quantum++

```cpp
// Quantum operations
// Source: ./examples/ex4.cpp
#include <qpp.h>
using namespace qpp;

int main()
{
    cmat rho = st.pb00; // projector onto the Bell state (|00>+|11>)/sqrt(2)
    std::cout << "Initial state:\n";
    std::cout << disp(rho) << std::endl;

    // partial transpose of first subsystem
    cmat rhoTA = ptranspose(rho, {0});
    std::cout << "Eigenvalues of the partial transpose of Bell-0 state are:\n";
    std::cout << disp(transpose(hevals(rhoTA))) << std::endl;

    std::cout << "Measurement channel with 2 Kraus operators:\n";
    std::vector<cmat> Ks {st.pz0, st.pz1}; // 2 Kraus operators
    std::cout << disp(Ks[0]) << "\n    and \n" << disp(Ks[1]) << std::endl;

    std::cout << "Superoperator matrix of the channel:\n";
    std::cout << disp(super(Ks)) << std::endl;

    std::cout << "Choi matrix of the channel:\n";
    std::cout << disp(choi(Ks)) << std::endl;

    // apply the channel onto the first subsystem
    cmat rhoOut = apply(rho, Ks, {0});
    std::cout << "After applying the measurement channel on the first qubit:\n";
    std::cout << disp(rhoOut) << std::endl;

    // take the partial trace over the second subsystem
    cmat rhoA = ptrace(rhoOut, {1});
    std::cout << "After partially tracing down the second subsystem:\n";
    std::cout << disp(rhoA) << std::endl;
}
```

Listing 4: Quantum operations