Quantum++ for the impatient

Author: Vlad Gheorghiu
vgheorgh@gmail.com

Dated: December 4, 2014

Contents

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 `$HOME/qpp_examples`, and inside it create the file `minimal.cpp`, with the content listed in the listing below. A verbatim copy of the above program is also available at `$HOME/qpp/examples/minimal.cpp`.

```
1 // Minimal example
2 // Source: ./examples/minimal.cpp
3 #include <qpp.h>
4 
5 int main()
6 {
7     std::cout << "Hello Quantum++!" << std::endl;
8 }
```

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 minimal.cpp -o minimal
```

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

```
>>> Starting Quantum++...
>>> Thu Dec  4 15:55:19 2014

Hello Quantum++!

>>> Exiting Quantum++...
>>> Thu Dec  4 15:55:19 2014
```

In line 3 of the program we include the main header file of the library, `qpp.h`. This file includes all other necessary Quantum++ header files, as well as the following C++ standard library files

```
<algorithm>
<chrono>
<cmath>
<complex>
<cstdlib>
<cstring>
<ctime>
<exception>
```

[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

```
<fstream>
<functional>
<initializer_list>
<iomanip>
<iostream>
<iterator>
<limits>
<numeric>
<ostream>
<random>
<sstream>
<stdexcept>
<string>
<tuple>
<type_traits>
<utility>
<vector>
```

and Eigen header files

```
<Eigen/Dense>
<Eigen/SVD>
```

Most of the time, you should be fine including only `qpp.h` in your main project, except when you want to use MATLAB input/output interface, in which case you have to explicitly include the header file `MATLAB/matlab.h`.

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

```
using namespace qpp;
```

in your main `.cpp` file.

3.1 Data types

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

3.2 Constants

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

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 `main()`, as listed in Table 3.

`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

`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

4 Simple examples

All examples of this section are copied verbatim from the directory `./examples` and compiled successfully. For 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 the listing below

```cpp
// Gates and states
// Source: ./examples/gates_states.cpp
#include <qpp.h>
using namespace qpp;

int main()
{
```

`const Init& init = Init::get_instance();`	Library initialization
`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

```
    ket psi = st.z0; // |0> state
    cmat U = gt.X;
    ket result = U * psi;

    std::cout << "The result of applying the bit-flip gate X on |0> is:\n";
    std::cout << disp(result) << std::endl;

    psi = mket({1, 0}); // |10> state
    U = gt.CNOT; // Controlled-NOT
    result = U * psi;

    std::cout << "The result of applying the gate CNOT on |10> is:\n";
    std::cout << disp(result) << std::endl;

    U = randU(2);
    std::cout << "Generating a random one-qubit gate U:\n";
    std::cout << disp(U) << std::endl;

    result = applyCTRL(psi, U, {0}, {1});
    std::cout << "The result of applying the Controlled-U gate on |10> is:\n";
    std::cout << disp(result) << std::endl;
}
```

which may output

```
>>> Starting Quantum++...
>>> Thu Dec  4 15:55:19 2014

The result of applying the bit-flip gate X on |0> is:
     0
1.0000
The result of applying the gate CNOT on |10> is:
     0
     0
     0
1.0000
Generating a random one-qubit gate U:
-0.0288 + 0.6679i  -0.0195 - 0.7434i
 0.6877 - 0.2829i   0.5992 - 0.2965i
The result of applying the Controlled-U gate on |10> is:
               0
               0
```

```
18 -0.0288 + 0.6679i
19  0.6877 - 0.2829i
20
21 >>> Exiting Quantum++...
22 >>> Thu Dec  4 15:55:19 2014
```

In line 4, we bring the namespace `qpp` into the global namespace.

In line 8 we use the singleton `st` to declare `psi` as the zero eigenvector $|0\rangle$ of the Z Pauli operator. In line 9 we assign to the gate `U` the bit flip gate `gt.X`, compute the result of the operation $X|0\rangle$ in line 10, and display the result $|1\rangle$ in lines 12 and 13. In line 13 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.

In line 15 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 16 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 17 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 19 and 20.

Next, in line 22 we generate a random unitary gate via the function `randU()`, then in line 26 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 27 and 28.

4.2 Measurements

Let us now complicate things a bit and introduce measurements. Consider the example in the listing below

```
1  // Measurements
2  // Source: ./examples/measurements.cpp
3  #include <qpp.h>
4  using namespace qpp;
5
6  int main()
7  {
8      ket psi = mket({0, 0});
9      cmat U = gt.CNOT * kron(gt.H, gt.Id2);
10     ket result = U * psi; // we have the Bell state (|00>+|11>)/sqrt(2)
11
12     std::cout << "We just produced the Bell state:\n";
13     std::cout << disp(result) << std::endl;
14
15     // apply a bit flip on the second qubit
16     result = apply(result, gt.X, {1}); // we produced (|01>+|10>)/sqrt(2)
17     std::cout << "We produced the Bell state:\n";
18     std::cout << disp(result) << std::endl;
19
20     // measure the first qubit in the X basis
21      auto m = measure(result, gt.H, {0});
22      std::cout << "Measurement result: " << std::get<0>(m);
23      std::cout << std::endl << "Probabilities: ";
24      std::cout << disp(std::get<1>(m), ", ") << std::endl;
25      std::cout << "Resulting states: " << std::endl;
26      for (auto && elem : std::get<2>(m))
27          std::cout << disp(elem) << std::endl << std::endl;
28  }
```

which outputs

```
>>> Starting Quantum++...
>>> Thu Dec  4 15:55:19 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: 0
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 Dec  4 15:55:19 2014
```

In line 9, 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, then we left-multiply it by the Controlled-NOT gate. In line 10 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 12 and 13.

In line 16 we use the function `apply()` to apply the gate X on the second qubit[2] 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 17 and 18 we display the newly created Bell state.

In line 21 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

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

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 22–27 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 the listing below we introduce quantum operations: quantum channels, as well as the partial trace and partial transpose operations.

```cpp
// Quantum operations
// Source: ./examples/quantum_operations.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;

    // compute the von-Neumann entropy
    double ent = entropy(rhoA);
39      std::cout << "Entropy: " << ent << std::endl;
40  }
```

The output of this program is

```
>>> Starting Quantum++...
>>> Thu Dec  4 15:55:19 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:
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
Entropy: 1.0000

>>> Exiting Quantum++...
>>> Thu Dec  4 15:55:19 2014
```

The example should by now be self-explanatory.

In line 8 we define the input state `rho` as the projector onto the Bell state $(|00\rangle + |11\rangle)/\sqrt{2}$, then display it in lines 9 and 10.

In lines 13–15 we partially transpose the first qubit, then display the eigenvalues of the resulting matrix `rhoTA`.

In lines 17–19 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 21–25 we display the superoperator matrix as well as the Choi matrix of the channel `Ks`.

Next, in lines 28–30 we apply the channel `Ks` to the first qubit of the input state `rho`, then display the output state `rhoOut`.

In lines 33–35 we take the partial trace of the output state `rhoOut`, then display the resulting state `rhoA`.

Finally, in lines 38 and 39 we compute the von-Neumann entropy of the resulting state and display it.

4.4 Timing

For convenience, Quantum++ provides a timer class defined in `./classes/timer.h`. It uses internally a `std::steady_clock`. The following program demonstrate its usage.

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

int main()
{
    std::cout << std::setprecision(8); // increase the default output precision

    // get the first codeword from Shor's [[9,1,3]] code
    ket c0 = codes.codeword(Codes::Type::NINE_QUBIT_SHOR, 0);

    Timer t; // declare and start a timer
    std::vector<idx> perm {1, 2, 3, 4, 5, 6, 7, 8, 0}; // declare a permutation
    ket c0perm = syspermute(c0, perm); // permute the system
    t.toc(); // stops the timer
    std::cout << "Permuting subsystems according to " << disp(perm, ", ");
    std::cout << "\nIt took " << t << " seconds to permute the subsytems.\n";

    t.tic(); // restart the timer
    std::cout << "Inverse permutation: ";
    std::cout << disp(invperm(perm), ", ") << std::endl;
    ket c0invperm = syspermute(c0perm, invperm(perm)); // permute again
    std::cout << "It took " << t.toc();
    std::cout << " seconds to un-permute the subsystems.\n";

    std::cout << "Norm difference: " << norm(c0invperm - c0) << std::endl;
}
```

A possible output of this program is

```
>>> Starting Quantum++...
>>> Thu Dec  4 15:55:19 2014

Permuting subsystems according to [1, 2, 3, 4, 5, 6, 7, 8, 0]
It took 0.00011800 seconds to permute the subsytems.
Inverse permutation: [8, 0, 1, 2, 3, 4, 5, 6, 7]
It took 0.00011400 seconds to un-permute the subsystems.
Norm difference: 0.00000000

```

```
>>> Exiting Quantum++...
>>> Thu Dec  4 15:55:19 2014
```

4.5 Input/output

We now introduce the input/output functions of Quantum++, as well as the input/output interfacing with MATLAB. The program below saves a matrix in both Quantum++ internal format as well as in MATLAB format, then loads it back and tests that the norm difference between the saved/loaded matrix is zero.

```
// Input/output
// Source: ./examples/input_output.cpp
#include <qpp.h>
#include <MATLAB/matlab.h>
using namespace qpp;

int main()
{
    // Quantum++ native input/output
    cmat rho = randrho(256); // an 8 qubit density operator
    save(rho, "rho.dat"); // save it
    cmat loaded_rho = load<cmat>("rho.dat"); // load it back
    // display the difference in norm, should be 0
    std::cout << "Norm difference load/save: ";
    std::cout << norm(loaded_rho - rho) << std::endl;

    // interfacing with MATLAB
    saveMATLABmatrix(rho, "rho.mat", "rho", "w");
    loaded_rho = loadMATLABmatrix<cmat>("rho.mat", "rho");
    // display the difference in norm, should be 0
    std::cout << "Norm difference MATLAB load/save: ";
    std::cout << norm(loaded_rho - rho) << std::endl;
}
```

The output of this program is

```
>>> Starting Quantum++...
>>> Thu Dec  4 15:55:19 2014

Norm difference load/save: 0.0000
Norm difference MATLAB load/save: 0.0000

>>> Exiting Quantum++...
>>> Thu Dec  4 15:55:19 2014
```

5 Brief description of Quantum++ file structure

A brief description of the Quantum++ file structure is presented below. The directories and their brief descriptions are emphasized using **bold fonts**. The main header file qpp.h is emphasized in red fonts.

```
./
└─ doc/ ........................................................ Documentation
   └─ html/ ............................................... HTML documentation
```

```
    │   └── index.html ........ Main HTML documentation file
    ├── quick.pdf ........ Quick reference guide
    └── refman.pdf ........ Complete reference
├── examples/ ........ Usage examples
│   └── ex*.cpp ........ Example source files
├── include/ ........ Header files
│   ├── MATLAB/ ........ MATLAB support
│   │   └── matlab.h ........ Input/output interfacing with MATLAB
│   ├── classes/ ........ Class definitions
│   │   ├── codes.h ........ Quantum error correcting codes
│   │   ├── exception.h ........ Exceptions
│   │   ├── gates.h ........ Quantum gates
│   │   ├── init.h ........ Initialization
│   │   ├── random_devices.h ........ Random devices
│   │   ├── states.h ........ Quantum states
│   │   └── timer.h ........ Timing
│   ├── experimental/ ........ Experimental/test functions/classes, do not use or modify
│   │   └── test.h ........ Experimental/test functions/classes
│   ├── internal/ ........ Internal implementation details, do not use/modify
│   │   ├── classes/ ........ Internal class definitions
│   │   │   ├── iomanip.h ........ Input/output manipulators
│   │   │   └── singleton.h ........ Singleton pattern via CRTP
│   │   └── util.h ........ Internal utility functions
│   ├── constants.h ........ Constants
│   ├── entanglement.h ........ Entanglement functions
│   ├── entropies.h ........ Entropy functions
│   ├── functions.h ........ Generic quantum computing functions
│   ├── input_output.h ........ Input/output functions
│   ├── instruments.h ........ Measurement functions
│   ├── number_theory.h ........ Number theory functions
│   ├── operations.h ........ Quantum operation functions
│   ├── qpp.h ........ Quantum++'s main header file, includes all other necessary headers
│   ├── random.h ........ Randomness-related functions
│   └── types.h ........ Type aliases
├── CMakeLists.txt ........ cmake configuration file, builds ./examples/example.cpp
├── COPYING ........ GNU General Public License version 3
├── README.md ........ Building instructions
├── RELEASE.md ........ Release notes
├── VERSION ........ Version number
└── run_OSX_MATLAB ........ Script for running with MATLAB support under OS X
```

6 Advanced topics

6.1 Exceptions

Exceptions...

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

```

```cpp
int main()
{
    cmat rho = randrho(16); // 4 qubits (subsystems)
    try
    {
        double mInfo = qmutualinfo(rho, {0}, {4}); // throws qpp::Exception
        std::cout << "Mutual information between first and last subsystem: ";
        std::cout << mInfo << std::endl;
    }
    catch (const std::exception& e)
    {
        std::cout << "Exception caught: " << e.what() << std::endl;
    }
}
```

The output of this program is

```
>>> Starting Quantum++...
>>> Thu Dec  4 15:55:19 2014

Exception caught: IN qpp::qmutualinfo(): Subsystems mismatch dimensions!

>>> Exiting Quantum++...
>>> Thu Dec  4 15:55:19 2014
```

6.2 Aliasing

6.3 Optimizations

6.4 Extending Quantum++