Quantum++ for the impatient

Author: Vlad Gheorghiu
vgheorgh@gmail.com

Date: November 20, 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

```cpp
#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, constants, states and gates

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.

The most important data types are defined in `types.h`[2] (inside the `include` folder of the Quantum++ source distribution). Complex matrices are typedef-ed as

[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 (like `<iostream>`, `<cmath>` etc), are automatically included in `qpp.h`, so the user should only include `qpp.h` in the main `.cpp` file.

```
cmat
```

and are an alias for `Eigen::MatrixXcd`. State vectors are typedef-ed as

```
ket (bra)
```

and are aliases for `Eigen::VectorXcd` and `Eigen::RowVectorXcd`, respectively. Matrices with real entries are typedef-ed as

```
dmat
```

and are an alias for `Eigen::MatrixXd`. Finally, complex numbers are typedef-ed as

```
cplx
```

and are an alias for `std::complex<double>`.

Quantum++ uses `std::size_t` (and not `int`) as the type for index-like quantities, like matrix sizes, measurement results etc.

Some important constants, such as π, base of natural logarithm e, square root on -1 i etc. are defined in the file `constants.h`. The square root of -1 is a C++11 user-defined literal.

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.

3.2 Simple examples

3.2.1 Gates and states

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

```
 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

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.

3.2.2 Measurements

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

```
     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.

3.2.3 Qudit quantum teleportation

We are now ready to consider a more complex real-world example: quantum teleportation of a qudit of dimension $D \geq 2$. Consider the example in Listing 4. A possible output of this program is

```
>>> Starting Quantum++...
>>> Thu Nov 20 16:39:09 2014

**** Qudit teleportation, D = 3 ****
>> Initial state:
 0.5359 + 0.2412i
-0.0339 + 0.5856i
 0.3062 - 0.4656i
>> Alice's measurement result: 7 -> [2 1]
>> Alice's measurement probabilities: [0.1111, 0.1111, 0.1111, 0.1111,
0.1111, 0.1111, 0.1111, 0.1111, 0.1111]
>> Bob must apply the correction operator Z^2 X^2
>> Bob's final state (after correction):
         0.3454    0.1231 - 0.3220i   0.0518 + 0.3234i
0.1231 + 0.3220i             0.3441   -0.2831 + 0.1635i
0.0518 - 0.3234i  -0.2831 - 0.1635i            0.3105
>> Norm difference: 0.0000

>>> Exiting Quantum++...
>>> Thu Nov 20 16:39:10 2014
```

In lines 9–12 we define the qudit generalization of a Bell state, namely the state $(\sum_{i=0}^{D-1} |i\rangle|i\rangle)/\sqrt{D}$, where D is the size of the system (a qutrit in our case) and was previously defined in line 6. Note the usage of another overload of `\mket` in line 11. In line 15, we construct the circuit that measures into the generalized Bell basis. Next, in line 18 we generate a random ket of dimension D (3 in our case), then display it.

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

3.2.4 Grover's searching algorithm

The code in Listing 5 implements Grover's searching algorithm on a database of size 2^n ($n = 4$ in our example). A possible output of this program is

```
>>> Starting Quantum++...
>>> Thu Nov 20 17:07:11 2014

**** Grover on n = 4 qubits ****
>> Database size: 16
>> Marked state: 11 -> [1 0 1 1]
>> We run 4 queries
>> Probability of the marked state: 0.5817
>> Probability of all results: [0.0279, 0.0279, 0.0279, 0.0279,
0.0279, 0.0279, 0.0279, 0.0279,
0.0279, 0.0279, 0.0279, 0.5817,
0.0279, 0.0279, 0.0279, 0.0279]
>> Let's sample...
>> Hooray, we obtained the correct result: 11 -> [1 0 1 1]
>> It took 0.0023 seconds to simulate Grover on 4 qubits.

>>> Exiting Quantum++...
>>> Thu Nov 20 17:07:11 2014
```

4 Brief description of Quantum++ files

5 Advanced topics

5.1 Exceptions

5.2 Aliasing

5.3 Optimizations

5.4 Extending Quantum++

```cpp
#include "qpp.h"
using namespace qpp;

int main()
{
    std::size_t D = 3; // size of the system
    std::cout << "**** Qudit teleportation, D = " << D << " ****" << std::endl;

    ket mes_AB = ket::Zero(D * D); // maximally entangled state resource
    for (std::size_t i = 0; i < D; ++i)
        mes_AB += mket({i, i}, D);
    mes_AB /= std::sqrt((double) D);

    // circuit that measures in the qudit Bell basis
    cmat Bell_aA = adjoint(gt.CTRL(gt.Xd(D), {0}, {1}, 2, D)
            * kron(gt.Fd(D), gt.Id(D)));

    ket psi_a = randket(D); // random state as input on a
    std::cout << ">> Initial state:" << std::endl;
    std::cout << disp(psi_a) << std::endl;

    ket input_aAB = kron(psi_a, mes_AB); // joint input state aAB
    // output before measurement
    ket output_aAB = apply(input_aAB, Bell_aA, {0, 1}, D);

    // measure on aA
    auto measured_aA = measure(output_aAB, gt.Id(D * D), {0, 1}, D);
    std::size_t m = std::get<0>(measured_aA); // measurement result

    auto midx = n2multiidx(m, {D, D});
    std::cout << ">> Alice's measurement result: ";
    std::cout << m << " -> " << disp(midx, " ") << std::endl;
    std::cout << ">> Alice's measurement probabilities: ";
    std::cout << disp(std::get<1>(measured_aA), ", ") << std::endl;

    // conditional result on B before correction
    cmat output_m_B = std::get<2>(measured_aA)[m];
    cmat correction_B = powm(gt.Zd(D), midx[0]) *
            powm(adjoint(gt.Xd(D)), midx[1]); // correction operator
    // apply correction on B
    std::cout << ">> Bob must apply the correction operator Z^" << midx[0]
            << " X^" << D - midx[1] << std::endl;
    cmat rho_B = correction_B * output_m_B * adjoint(correction_B);

    std::cout << ">> Bob's final state (after correction): " << std::endl;
    std::cout << disp(rho_B) << std::endl;

    // verification
    std::cout << ">> Norm difference: " << norm(rho_B - prj(psi_a)) << std::endl;
}
```

Listing 4: Qudit quantum teleportation

```
#include "qpp.h"
using namespace qpp;

int main()
{
    Timer t; // start a timer

    std::size_t n = 4; // number of qubits
    std::cout << "**** Grover on n = " << n << " qubits ****" << std::endl;

    std::vector<std::size_t> dims(n, 2); // local dimensions
    std::size_t N = std::round(std::pow(2, n)); // number of elements in the database
    std::cout << ">> Database size: " << N << std::endl;

    std::size_t marked = randidx(0, N - 1); // mark an element randomly
    std::cout << ">> Marked state: " << marked << " -> ";
    std::cout << disp(n2multiidx(marked, dims), " ") << std::endl;

    ket psi = mket(n2multiidx(0, dims)); // computational |0>^\otimes n
    // apply H^\otimes n, no aliasing
    psi = (kronpow(gt.H, n) * psi).eval();
    cmat G = 2 * prj(psi) - gt.Id(N); // Diffusion operator

    // number of queries
    std::size_t nqueries = std::ceil(pi * std::sqrt((double) N) / 4.);
    std::cout << ">> We run " << nqueries << " queries" << std::endl;
    for (std::size_t i = 0; i < nqueries; ++i)
    {
        psi(marked) = -psi(marked); // apply the oracle first, no aliasing
        psi = (G * psi).eval(); // then the diffusion operator, no aliasing
    }

    auto measured = measure(psi, gt.Id(N)); // measure in the computational basis
    std::cout << ">> Probability of the marked state: "
            << std::get<1>(measured)[marked] << std::endl;
    std::cout << ">> Probability of all results: ";
    std::cout << disp(std::get<1>(measured), ", ") << std::endl;

    std::cout << ">> Let's sample..." << std::endl; // sample
    std::size_t result = std::get<0>(measured);
    if (result == marked)
        std::cout << ">> Hooray, we obtained the correct result: ";
    else
        std::cout << ">> Not there yet... we obtained: ";
    std::cout << result << " -> ";
    std::cout << disp(n2multiidx(result, dims), " ") << std::endl;

    // stop the timer and display it
    std::cout << ">> It took " << t.toc() << " seconds to simulate Grover on "
            << n << " qubits." << std::endl;
}
```

Listing 5: Grover's searching algorithm