Quantum++
A C++11 quantum computing library

Author: Vlad Gheorghiu
Institute for Quantum Computing, University of Waterloo,
Waterloo, ON, N2L 3G1, Canada
vgheorgh@gmail.com

Version of: November 18, 2017

Abstract: Quantum++ is a modern general-purpose multi-threaded quantum computing library written in C++11 and composed solely of header files. The library is not restricted to qubit systems or specific quantum information processing tasks, being capable of simulating arbitrary quantum processes. The main design factors taken in consideration were the ease of use, portability, and performance. The library's simulation capabilities are only restricted by the amount of available physical memory. On a typical machine (Intel i5 8Gb RAM) Quantum++ can successfully simulate the evolution of 25 qubits in a pure state or of 12 qubits in a mixed state reasonably fast.

Contents

Listings

1 Introduction

Quantum++, available online at `https://github.com/vsoftco/qpp`, is a C++11 general purpose quantum computing library, composed solely of header files. It uses the Eigen 3 linear algebra library and, if available, the OpenMP multi-processing library. For additional Eigen 3 documentation see `http://eigen.tuxfamily.org/dox/`. For a simple Eigen 3 quick ASCII reference see `http://eigen.tuxfamily.org/dox/AsciiQuickReference.txt`.

The simulator defines a large collection of (template) quantum computing related functions and a few useful classes. The main data types are complex vectors and complex matrices, which 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<>`, instantiated accordingly.

Although there are many available quantum computing libraries/simulators written in various programming languages, see [1] for a comprehensive list, I hope what makes Quantum++ different is the ease of use, portability and high performance. The library is not restricted to specific quantum information tasks, but it is intended to be multi-purpose and capable of simulating arbitrary quantum processes. I have chosen the C++ programming language (standard C++11) in implementing the library as it is by now a mature standard, fully (or almost fully) implemented by most important compilers, and highly portable.

In the reminder of this manuscript I describe the main features of the library, "in a nutshell" fashion, via a series of simple examples. I assume that the reader is familiar with the basic concepts of quantum mechanics/quantum information, as I do not provide any introduction to this field. For a comprehensive introduction to the latter see e.g. [2]. This document is not intended to be a comprehensive documentation, but only a brief introduction to the library and its main features. For a detailed reference see the official manual available as a `.pdf` file in `./doc/refman.pdf`. For detailed installation instructions as well as for additional information regarding the library see the main repository page at `https://github.com/vsoftco/qpp`. If you are interesting in contributing, or for any comments or suggestions, please email me at vgheorgh@gmail.com.

2 Installation

To get started with Quantum++, first install the Eigen 3 library from `http://eigen.tuxfamily.org` into your 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 the Eigen 3 library. Next, download the Quantum++ library from `https://github.com/vsoftco/qpp/` and unzip it into the home directory as `$HOME/qpp`. Finally, make sure that your compiler supports C++11 and preferably OpenMP. For a compiler I recommend g++ version 4.8.2 or later or clang version 3.7 or later (previous versions of clang do not support OpenMP). You are now ready to go!

We next build a simple minimal example to test that the installation was successful. Create a directory called `$HOME/testing`, and inside it create the file `minimal.cpp`, with the content listed in Listing 1. A verbatim copy of the above program is also available at `$HOME/qpp/examples/minimal.cpp`.

```
1  // Minimal example
2  // Source: ./examples/minimal.cpp
3  #include <iostream>
4  #include "qpp.h"
5
6  using namespace qpp;
7
8  int main()
9  {
10     std::cout << "Hello Quantum++!\nThis is the |0> state:\n";
11     std::cout << disp(st.z0) << '\n';
12 }
```

Listing 1: Minimal example

Next, compile the file using a C++11 compliant compiler. In the following I assume you use g++, but the building instructions are similar for other compilers. From the directory `$HOME/testing` 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 C++ compiler and operating system. If everything went fine, the above command should build an executable `minimal` in `$HOME/testing`, which can be run by typing `./minimal`. The output should be similar to the following:

```
Hello Quantum++!
This is the |0> state:
1
0
```

Listing 1 output

In line 4 of Listing 1 we include the main header file of the library `qpp.h` This header file includes all other necessary internal Quantum++ header files. In line 11 we display the state $|0\rangle$ represented by the singleton `st.z0` in a nice format using the display manipulator `disp()`.

3 Data types, constants and global objects

All header files of Quantum++ are located inside the `./include` directory. All functions, classes and global objects defined by the library belong to the `namespace qpp`. To avoid additional typing, I will omit the prefix `qpp::` in the rest of this document. I recommend to use `using namespace qpp;` in your main `.cpp` file.

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

3.1 Data types

The most important data types are defined in the header file `types.h`. We list them in Table 1.

`idx`	Index (non-negative integer), alias for `std::size_t`
`bigint`	Big integer, alias for `long long int`
`cplx`	Complex number, alias for `std::complex<double>`
`cmat`	Complex dynamic matrix, alias for `Eigen::MatrixXcd`
`dmat`	Double dynamic matrix, 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 idx maxn = 64;`	Maximum number of allowed qu(d)its (subsystems)
`constexpr double pi = 3.1415...;`	π
`constexpr double ee = 2.7182...;`	e, base of natural logarithms
`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 double infty = ...;`	Used to denote infinity in double precision
`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 Singleton classes and their global instances

Some useful classes are defined as singletons and their instances are globally available, being initialized at runtime in the header file `qpp.h`, before `main()`. They are listed in Table 3.

4 Simple examples

All of the examples of this section are copied verbatim from the directory `./examples` and are fully compilable. For convenience, the location of the source file is displayed in the first line of each example as a C++ comment. The examples are simple and demonstrate the main features of Quantum++. They cover

`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_thread_local_instance();`	Random devices/generators/engines

Table 3: Global singleton classes and instances

only a small part of library functions, but enough to get the interested user started. For an extensive reference of all library functions, including various overloads, the user should consult the complete reference `./doc/refman.pdf`. See the rest of the examples (not discussed in this document) in `./examples/`. for more comprehensive code snippets.

4.1 Gates and states

Let us introduce the main objects used by Quantum++: gates, states and basic operations. Consider the code in Listing 2.

```
// Gates and states
// Source: ./examples/gates_states.cpp
#include <iostream>
#include "qpp.h"

using namespace qpp;

int main()
{
    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) << '\n';

    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) << '\n';

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

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

```
31 }
```

Listing 2: Gates and states

A possible output is:

```
>> The result of applying the bit-flip gate X on |0> is:
0
1
>> The result of applying the gate CNOT on |10> is:
0
0
0
1
>> Generating a random one-qubit gate U:
 -0.53824 - 0.140356i  -0.467118 - 0.687313i
-0.533027 + 0.637557i  0.549513 - 0.0862425i
>> The result of applying the CTRL-U gate on |10> is:
                    0
                    0
 -0.53824 - 0.140356i
-0.533027 + 0.637557i
```

Listing 2 output

In line 4 of Listing 2 we bring the namespace `qpp` into the global namespace.

In line 10 we use the `States` singleton `st` to declare `psi` as the zero eigenvector $|0\rangle$ of the Z Pauli operator. In line 11 we use the `Gates` singleton `gt` and assign to `U` the bit flip gate `gt.X`. In line 12 we compute the result of the operation $X|0\rangle$, and display the result $|1\rangle$ in lines 14 and 15. In line 15 we use the display 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 17 we reassign to `psi` the state $|10\rangle$ via the function `mket()`. We could have also used the Eigen 3 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 18 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 19 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 21 and 22.

Next, in line 24 we generate a random unitary gate via the function `randU()`, then in line 28 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 29 and 30.

4.2 Measurements

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

```
1 // Measurements
2 // Source: ./examples/measurements.cpp
3 #include <iostream>
4 #include <tuple>
```

```cpp
#include "qpp.h"

using namespace qpp;

int main()
{
    ket psi = mket({0, 0});
    cmat U = gt.CNOT * 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) << '\n';

    // 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) << '\n';

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

Listing 3: Measurements

A possible output is:

```
>> We just produced the Bell state:
0.707107
       0
       0
0.707107
>> We produced the Bell state:
       0
0.707107
0.707107
       0
>> Measurement result: 1
>> Probabilities: [0.5, 0.5]
>> Resulting states:
0.707107
0.707107

-0.707107
 0.707107
```

Listing 3 output

In line 12 of Listing 3 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 13 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 15 and 16.

In line 19 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 20 and 21 we display the newly created Bell state.

In line 24 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 tuple of 3 elements

```
std::tuple<qpp::idx, std::vector<double>, std::vector<qpp::cmat>>
```

The first element represents the measurement result, the second the possible output probabilities and the third the output output states. Instead of using this long type definition, we use the new C++11 `auto` keyword to define the type of the result `measured` of `measure()`. In lines 25–30 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.

```
1  // Quantum operations
2  // Source: ./examples/quantum_operations.cpp
3  #include <iostream>
4  #include <vector>
5  #include "qpp.h"
6
7  using namespace qpp;
8
9  int main()
10 {
11     cmat rho = st.pb00; // projector onto the Bell state (|00> + |11>) / sqrt(2)
12     std::cout << ">> Initial state:\n";
13     std::cout << disp(rho) << '\n';
14
15     // partial transpose of first subsystem
16     cmat rhoTA = ptranspose(rho, {0});
17     std::cout << ">> Eigenvalues of the partial transpose "
18               << "of Bell-0 state are:\n";
19     std::cout << disp(transpose(hevals(rhoTA))) << '\n';
20
21     std::cout << ">> Measurement channel with 2 Kraus operators:\n";
22     std::vector<cmat> Ks{st.pz0, st.pz1}; // 2 Kraus operators
```

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

```
23      std::cout << disp(Ks[0]) << "\nand\n" << disp(Ks[1]) << '\n';
24
25      std::cout << ">> Superoperator matrix of the channel:\n";
26      std::cout << disp(kraus2super(Ks)) << '\n';
27
28      std::cout << ">> Choi matrix of the channel:\n";
29      std::cout << disp(kraus2choi(Ks)) << '\n';
30
31      // apply the channel onto the first subsystem
32      cmat rhoOut = apply(rho, Ks, {0});
33      std::cout << ">> After applying the measurement channel "
34                << "on the first qubit:\n";
35      std::cout << disp(rhoOut) << '\n';
36
37      // take the partial trace over the second subsystem
38      cmat rhoA = ptrace(rhoOut, {1});
39      std::cout << ">> After partially tracing down the second subsystem:\n";
40      std::cout << disp(rhoA) << '\n';
41
42      // compute the von-Neumann entropy
43      double ent = entropy(rhoA);
44      std::cout << ">> Entropy: " << ent << '\n';
45  }
```

Listing 4: Quantum operations

The output of this program is:

```
>> Initial state:
0.5   0   0   0.5
  0   0   0     0
  0   0   0     0
0.5   0   0   0.5
>> Eigenvalues of the partial transpose of Bell-0 state are:
-0.5   0.5   0.5   0.5
>> Measurement channel with 2 Kraus operators:
1   0
0   0
and
0   0
0   1
>> Superoperator matrix of the channel:
1   0   0   0
0   0   0   0
0   0   0   0
0   0   0   1
>> Choi matrix of the channel:
1   0   0   0
0   0   0   0
0   0   0   0
0   0   0   1
>> After applying the measurement channel on the first qubit:
0.5   0   0     0
```

```
  0   0   0     0
  0   0   0     0
  0   0   0   0.5
>> After partially tracing down the second subsystem:
0.5     0
  0   0.5
>> Entropy: 1
```

Listing 4 output

The example should by now be self-explanatory. In line 11 of Listing 4 we define the input state `rho` as the projector onto the Bell state $(|00\rangle + |11\rangle)/\sqrt{2}$, then display it in lines 12 and 13.

In lines 16–19 we partially transpose the first qubit, then display the eigenvalues of the resulting matrix `rhoTA`.

In lines 21–23 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 25–29 we display the superoperator matrix as well as the Choi matrix of the channel `Ks`.

Next, in lines 32–35 we apply the channel `Ks` to the first qubit of the input state `rho`, then display the output state `rhoOut`.

In lines 38–40 we take the partial trace of the output state `rhoOut`, then display the resulting state `rhoA`.

Finally, in lines 43 and 44 we compute the von-Neumann entropy of the resulting state and display it.

4.4 Timing

To facilitate simple timing tasks, Quantum++ provides a `Timer` class that uses internally a `std::steady_clock`. The program in Listing 5 demonstrate its usage.

```
1  // Timing
2  // Source: ./examples/timing.cpp
3  #include <iomanip>
4  #include <iostream>
5  #include <vector>
6  #include "qpp.h"
7
8  using namespace qpp;
9
10 int main()
11 {
12     std::cout << std::setprecision(8); // increase the default output precision
13
14     // get the first codeword from Shor's [[9,1,3]] code
15     ket c0 = codes.codeword(Codes::Type::NINE_QUBIT_SHOR, 0);
16
17     Timer<> t; // declare and start a timer
18     std::vector<idx> perm = randperm(9); // declare a random permutation
19     ket c0perm = syspermute(c0, perm); // permute the system
20     t.toc(); // stops the timer
21     std::cout << ">> Permuting subsystems according to " << disp(perm, ", ");
22     std::cout << "\n>> It took " << t << " seconds to permute the subsytems.\n";
23
24     t.tic(); // restart the timer
25     std::cout << ">> Inverse permutation: ";
```

```
26      std::cout << disp(invperm(perm), ", ") << '\n';
27      ket c0invperm = syspermute(c0perm, invperm(perm)); // permute again
28      std::cout << ">> It took " << t.toc();
29      std::cout << " seconds to un-permute the subsystems.\n";
30
31      std::cout << ">> Norm difference: " << norm(c0invperm - c0) << '\n';
32  }
```

Listing 5: Timing

A possible output of this program is:

```
>> Permuting subsystems according to [5, 4, 0, 2, 8, 1, 6, 7, 3]
>> It took 0.000544 seconds to permute the subsytems.
>> Inverse permutation: [2, 5, 3, 8, 1, 0, 6, 7, 4]
>> It took 0.000286 seconds to un-permute the subsystems.
>> Norm difference: 0
```

Listing 5 output

In line 12 of Listing 5 we change the default output precision from 4 to 8 decimals after the delimiter.

In line 15 we use the `Codes` singleton `codes` to retrieve in `c0` the first codeword of the Shor's $[[9, 1, 3]]$ quantum error correcting code.

In line 17 we declare an instance `timer` of the class `Timer`. In line 18 we declare a random permutation `perm` via the function `randperm()`. In line 19 we permute the codeword according to the permutation `perm` using the function `syspermute()` and store the result . In line 20 we stop the timer. In line 21 we display the permutation, using an overloaded form of the `disp()` manipulator for C++ standard library containers. The latter takes a `std::string` as its second parameter to specify the delimiter between the elements of the container. In line 22 we display the elapsed time using the `ostream operator<<()` operator overload for `Timer` objects.

Next, in line 24 we reset the timer, then display the inverse permutation of `perm` in lines 25 and 26. In line 27 we permute the already permuted state `c0perm` according to the inverse permutation of `perm`, and store the result in `c0invperm`. In lines 28 and 29 we display the elapsed time. Note that in line 28 we used directly `t.toc()` in the stream insertion operator, since, for convenience, the member function `Timer::toc()` returns a `const Timer&`.

Finally, in line 31, we verify that by permuting and permuting again using the inverse permutation we recover the initial codeword, i.e. the norm difference has to be zero.

4.5 Input/output

We now introduce the input/output functions of Quantum++, as well as the input/output interfacing with MATLAB. The program in Listing 6 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.

```
1  // Input/output
2  // Source: ./examples/input_output.cpp
3  #include <iostream>
4  #include "qpp.h"
5  #include "MATLAB/matlab.h" // must be explicitly included
6
7  using namespace qpp;
8
9  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) << '\n';

    // interfacing with MATLAB
    saveMATLAB(rho, "rho.mat", "rho", "w");
    loaded_rho = loadMATLAB<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) << '\n';
}
```

Listing 6: Input/output

The output of this program is:

```
>> Norm difference load/save: 0
>> Norm difference MATLAB load/save: 0
```

Listing 6 output

Note that in order to use the MATLAB input/output interface support, you need to explicitly include the header file `MATLAB/matlab.h`, and you also need to have MATLAB or MATLAB compiler installed, otherwise the program fails to compile. See the file `./README.md` for extensive details about compiling with MATLAB support.

4.6 Exceptions

Most Quantum++ functions throw exceptions in the case of unrecoverable errors, such as out-of-range input parameters, input/output errors etc. The exceptions are handled via the class `Exception`, derived from `std::exception`. The exception types are hard-coded inside the strongly-typed enumeration (enum class) `Exception::Type`. If you want to add more exceptions, augment the enumeration `Exception::Type` and also modify accordingly the member function `Exception::construct_exception_msg_()`, which constructs the exception message displayed via the overridden virtual function `Exception::what()`. Listing 7 illustrates the basics of exception handling in Quantum++.

```
// Exceptions
// Source: ./examples/exceptions.cpp
#include <exception>
#include <iostream>
#include "qpp.h"

using namespace qpp;

int main()
{
    cmat rho = randrho(16); // 4 qubits (subsystems)
    try
    {
        // the line below throws qpp::exception::SubsysMismatchDims
```

```
15          double mInfo = qmutualinfo(rho, {0}, {4});
16          std::cout << ">> Mutual information between first and last subsystem: ";
17          std::cout << mInfo << '\n';
18      }
19      catch (const std::exception& e)
20      {
21          std::cout << ">> Exception caught: " << e.what() << '\n';
22      }
23  }
```

Listing 7: Exceptions

The output of this program is:

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

Listing 7 output

In line 11 of Listing 7 we define a random density matrix on four qubits (dimension 16). In line 15, we compute the mutual information between the first and the 5-th subsystem. Line 15 throws an exception of type `qpp::exception::SubsysMismatchDim` exception, as there are only four systems. We next catch the exception in line 19 via the `std::exception` standard exception base class. We could have also used the Quantum++ exception base class `qpp::exception::Exception`, however using the `std::exception` allows the catching of other exceptions, not just of the type `Exception`. Finally, in line 21 we display the corresponding exception message.

5 Brief description of Quantum++ file structure

A brief description of the Quantum++ file structure is presented in Figure 1. The directories and their brief descriptions are emphasized using **bold fonts**. The main header file qpp.h is emphasized in red fonts.

6 Advanced topics

6.1 Aliasing

Aliasing occurs whenever the same Eigen 3 matrix/vector appears on both sides of the assignment operator, and happens because of Eigen 3's lazy evaluation system. Examples that exhibit aliasing:

```
mat = 2 * mat;
```

or

```
mat = mat.transpose();
```

Aliasing *does not* occur in statements like

```
mat = f(mat);
```

where `f()` returns by value. Aliasing produces in general unexpected results, and should be avoided at all costs.

Whereas the first line produces aliasing, it is not dangerous, since the assignment is done in a one-to-one manner, i.e. each element (i, j) on the left hand side of the assignment operator is solely a function of the the *same* (i, j) element on the right hand side, i.e. $mat(i, j) = f(mat(i, j)), \forall i, j$. The problem appears whenever coefficients are being combined and overlap, such as in the second example, where $mat(i, j) = mat(j, i), \forall i, j$. To avoid aliasing, use the member function `eval()` to transform the right hand side object into a temporary, such as

```
./
├── VisualStudio/ ..... Visual Studio 2017 solution
├── 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
│   │   ├── idisplay.h ..... Interface for stream extraction
│   │   ├── 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
│   │   └── experimental.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++ main header file, includes all other necessary headers
│   ├── random.h ..... Randomness functions
│   ├── statistics.h ..... Statistics functions
│   ├── traits.h ..... Type traits
│   └── types.h ..... Type aliases
├── unit_tests/ ..... Unit testing with Google Mock/Test (included in lib)
├── CHANGES ..... Modifications added by each version
├── CMakeLists.txt ..... cmake configuration file, builds ./examples/minimal.cpp and the unit tests
├── CODE_OF_CONDUCT.md ..... Code of conduct
├── CONTRIBUTING.md ..... Contributing rules
├── COPYING ..... GNU General Public License version 3
├── README.md ..... Building instructions
├── RELEASE.md ..... Release notes
├── VERSION ..... Version number
└── run_mac_MATLAB ..... Script for running with MATLAB support under OS X
```

Figure 1: Quantum++ file structure

```
mat = 2 * mat.eval();
```

In general, aliasing can not be detected at compile time, but can be detected at runtime whenever the compile flag `EIGEN_NO_DEBUG` is not set. Quantum++ does not set this flag in debug mode. I highly recommend to first compile your program in debug mode to detect aliasing run-time assertions, as well as other possible issues that may have escaped you, such as assigning to a matrix another matrix of different dimension etc.

For more details about aliasing, see the official Eigen 3 documentation at http://eigen.tuxfamily.org/dox/group__TopicAliasing.html.

6.2 Type deduction via `auto`

Avoid the usage of `auto` when working with Eigen 3 expressions, e.g. avoid writing code like

```
auto mat = A * B + C;
```

but write instead

```
cmat mat = A * B + C;
```

or

```
auto mat = (A * B + C).eval();
```

to force evaluation, as otherwise you may get unexpected results. The "problem" lies in the Eigen 3 lazy evaluation system and reference binding, see e.g. http://stackoverflow.com/q/26705446/3093378 for more details. In short, the reference to the internal data represented by the expression `A * B + C` is dangling at the end of the `auto mat = A * B + C;` statement.

6.3 Optimizations

Whenever testing your application, I recommend compiling in debug mode, as Eigen 3 run-time assertions can provide extremely helpful feedback on potential issues. Whenever the code is production-ready, you should *always* compile with optimization flags turned on, such as `-O3` (for g++) and `-DEIGEN_NO_DEBUG`. You should also turn on the OpenMP (if available) multi-processing flag (`-fopenmp` for g++), as it enables multi-core/multi-processing with shared memory. Eigen 3 uses multi-processing when available, e.g. in matrix multiplication. Quantum++ also uses multi-processing in computationally-intensive functions.

Since most Quantum++ functions return by value, in assignments of the form

```
mat = f(another_mat);
```

there is an additional copy assignment operator when assigning the temporary returned by `f()` back to `mat`. As far as I know, this extra copy operation is not elided. Unfortunately, Eigen 3 does not yet support move semantics, which would have got rid of this additional assignment via the corresponding move assignment operator. If in the future Eigen 3 will support move semantics, the additional assignment operator will be "free", and you won't have to modify any existing code to enable the optimization; the Eigen 3 move assignment operator should take care of it for you.

Note that in a line of the form

```
cmat mat = f(another_mat);
```

most compilers perform return value optimization (RVO), i.e. the temporary on the right hand side is constructed directly inside the object `mat`, the copy constructor being elided.

6.4 Extending Quantum++

Most Quantum++ operate on Eigen 3 matrices/vectors, and return either a matrix or a scalar. In principle, you may be tempted to write a new function such as

```
cmat f(const cmat& A){...}
```

The problem with the approach above is that Eigen 3 uses *expression templates* as the type of each expression, i.e. different expressions have in general different types, see the official Eigen 3 documentation at http://eigen.tuxfamily.org/dox/TopicFunctionTakingEigenTypes.html for more details. The correct way to write a generic function that is guaranteed to work with any matrix expression is to make your function template and declare the input parameter as `Eigen::MatrixBase<Derived>`, where `Derived` is the template parameter. For example, the Quantum++ `transpose()` function is defined as

```
1  template<typename Derived>
2  dyn_mat<typename Derived::Scalar>
3  transpose(const Eigen::MatrixBase<Derived>& A)
4  {
5      const dyn_mat<typename Derived::Scalar>& rA = A.derived();
6
7      // check zero-size
8      if (!internal::check_nonzero_size(rA))
9          throw Exception("qpp::transpose()", Exception::Type::ZERO_SIZE);
10
11     return rA.transpose();
12 }
```

It takes an Eigen 3 matrix expression, line 3, and returns a dynamic matrix over the scalar field of the expression, line 2. In line 5 we implicitly convert the input expression `A` to a dynamic matrix `rA` over the same scalar field as the expression, via binding to a `const` reference, therefore paying no copying cost. We then use `rA` instead of the original expression `A` in the rest of the function. Note that most of the time it is OK to use the original expression, however there are some cases where you may get a compile time error if the expression is not explicitly casted to a matrix. For consistency, I use this reference binding trick in the code of all Quantum++ functions.

As you may have already seen, Quantum++ consists mainly of a collection of functions and few classes. There is no complicated class hierarchy, and you can regard the Quantum++ API as a medium-level API. You may extend it to incorporate graphical input, e.g. use a graphical library such as Qt, or build a more sophisticated library on top of it. I recommend to read the source code and make yourself familiar with the library before deciding to extend it. You should also check the complete reference manual `./doc/refman.pdf` for an extensive documentation of all functions and classes. I hope you find Quantum++ useful and wish you a happy usage!

Acknowledgements

I acknowledge financial support from Industry Canada and from the Natural Sciences and Engineering Research Council of Canada (NSERC). I thank Kassem Kalach for carefully reading this manuscript and providing useful suggestions.

References

[1] List of QC simulators, available online at http://www.quantiki.org/wiki/List_of_QC_simulators.

[2] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, 5th edition, 2000.