Quantum++
A modern C++ quantum computing library
arXiv:1412.4704

Vlad Gheorghiu

vgheorgh@gmail.com

Institute for Quantum Computing
University of Waterloo
Waterloo, ON, N2L 3G1, Canada

Quantum Programming and Circuits Workshop
June 8, 2015

Introduction

- What is a simulator and why do we care?
- Perform "experiments" on our laptop.
- Test our conjectures, even find new results.
- Understand quantum mechanics better, without the need for fancy (and very expensive) equipment

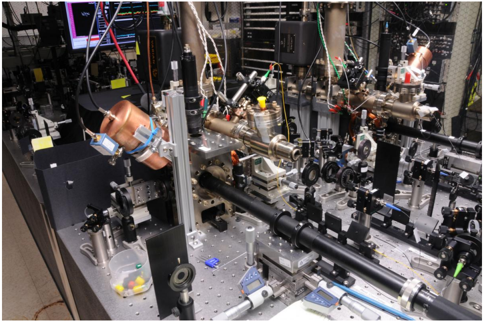


Figure: Courtesy of http://jqi.umd.edu/

Understand this

with this

"Scientists typically develop their own software for these purposes because doing so requires substantial domain-specific knowledge. As a result, recent studies have found that **scientists typically spend 30% or more of their time developing software. However, 90% or more of them are primarily self-taught, and therefore lack exposure to basic software development practices such as writing maintainable code**, using version control and issue trackers, code reviews, unit testing, and task automation." [G. Wilson et al. (2014), *Best Practices for Scientific Computing*, PLoS Biol 12(1), free at `http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3886731/`]

Language	# of quantum-related simulators
C or C++	26
Java	15
GUI-based	12
MATLAB	11
Mathematica	8
Pyhton	3
Exotic (Scheme/Haskell/Lisp/ML)	5

Table: State of affairs, `http://quantiki.org/List_of_QC_simulators`. Most of these simulators are domain-specific. Some need valid licenses ($$$).

- Scientists can write bad code *extremely* easy, even if they put a lot of effort into it.
- Choosing a fast language does not make the code faster.
- It usually end up making it hard to read, hard to understand, and prone to (sometimes subtle, most of the time not so sublte) bugs.
- Goal of a good simulation software: switch the burden of writing "good" code from the end user to the library developer.
- Why C++?
  1. Free
  2. Fast (standard library designed with zero abstraction overhead in mind)
  3. *Extremely* portable (runs everywhere), stable and mature (code 20 years old will most likely run 20 years from now)
  4. Strongly-typed, statically-typed, room for serious compiler optimizations.
  5. Modern: C++11/14 supports lambda functions, closures, smart pointers, multithreading, automatic type deduction etc.

Brief description

- Written in standard C++11.
- Freely available (GPLv3) at `http://vsoftco.github.io/qpp/`.
- Low dependance on external libraries. It only depends on Eigen, `http://eigen.tuxfamily.org/`.
- Easy to install

```
git clone https://github.com/vsoftco/qpp.git
```

- Extensive documentation: both quick starting guide and full API.
- Easy to use.

```
// Quantum++ main library header file
#include <qpp.h>

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

- Compile and run with

```
cd $HOME/qpp; mkdir build; cd build; cmake ..; make; ./qpp
```

- Output:

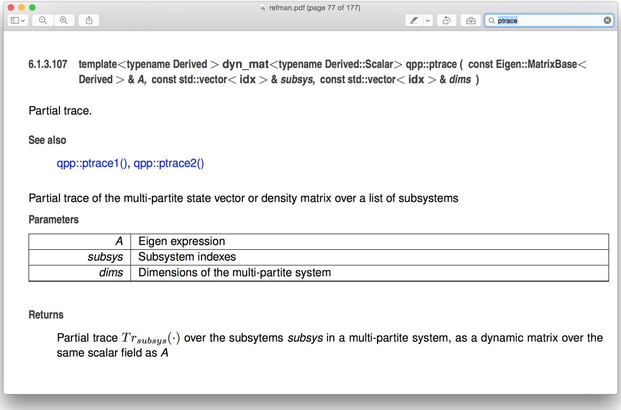
```
>>> Starting Quantum++...
>>> Wed May 20 18:26:03 2015

Hello Quantum++!

>>> Exiting Quantum++...
>>> Wed May 20 18:26:03 2015
```

- Under the hood:

```
  g++ -pedantic -std=c++11 -Wall -Wextra -Weffc++ -fopenmp -O3
-DNDEBUG -DEIGEN_NO_DEBUG -isystem $HOME/eigen -I $HOME/qpp/include
$HOME/qpp/examples/minimal.cpp -o minimal
```

Documentation

- Quick starting guide arXiv:1412.4704
- Full scale documentation, automatically-generated, .pdf and .html (doxygen)



6.1.3.107 template< typename Derived > dyn_mat< typename Derived::Scalar> qpp::ptrace (const Eigen::MatrixBase< Derived > & *A*, const std::vector< idx > & *subsys*, const std::vector< idx > & *dims*)

Partial trace.

See also

qpp::ptrace1(), qpp::ptrace2()

Partial trace of the multi-partite state vector or density matrix over a list of subsystems

Parameters

A	Eigen expression
subsys	Subsystem indexes
dims	Dimensions of the multi-partite system

Returns

Partial trace $Tr_{subsys}(\cdot)$ over the subsytems *subsys* in a multi-partite system, as a dynamic matrix over the same scalar field as *A*

Technicalities

- Can simulate dense quantum computation reasonably fast on 24 qubits (pure states), or 13 qubits (mixed states).[1]
- Contains around 5000 lines of code.
- Header only, no need to compile the library.
- Template-based code, uses expression templates and lazy evaluation.
- Multi-threaded via OpenMP.
- Can simulate arbitrary quantum processing tasks. Not restricted to qubits.
- Intended to use as an API, no GUI.
- “Hard” to use incorrectly.
- Easy to extend.

---

[1] MacBook Pro laptop, 2.5GHz dual-core Intel Core i5 processor, 8GB RAM

- Very few custom data types: complex/real matrices and vectors: `cmat`, `ket`, `bra`.
- Defines a significant collections of quantum information-related functions that operate on such data types.
- Uses a "functional-style" approach: data is immutable (technically passed by constant references), functions do not have side effects (seen as black boxes).
- Example:

```
cmat rhoA = ptrace(rhoAB, {1}); // partial trace
```

- Advantages: easy to test, easy to use in a multi-processing environment, highly optimized (move semantics, RVO).

Gates and states

```
// Source: ./examples/gates_states.cpp
#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) << 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;
}
```

Gates and states output

```
>>> Starting Quantum++...
>>> Wed May 20 18:26:01 2015

>> 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:
 0.3272 - 0.2006i  -0.6030 + 0.6993i
-0.6858 + 0.6183i  -0.1933 + 0.3316i
>> The result of applying the Controlled-U gate on |10> is:
               0
               0
 0.3272 - 0.2006i
-0.6858 + 0.6183i

>>> Exiting Quantum++...
>>> Wed May 20 18:26:01 2015
```

Measurements

```cpp
// Source: ./examples/measurements.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) << 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&& it : std::get<2>(m))
        std::cout << disp(it) << std::endl;
}
```

Measurements output

```
>>> Starting Quantum++...
>>> Wed May 20 18:26:03 2015

>> 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++...
>>> Wed May 20 18:26:03 2015
```

Dense coding

```
// Qudit dense coding
// Source: ./examples/dense_coding.cpp
#include <qpp.h>
using namespace qpp;
using std::cout;
using std::endl;

int main()
{
    idx D = 3; // size of the system
    cout << ">> Qudit dense coding, D = " << D << endl;

    ket mes_AB = ket::Zero(D * D); // maximally entangled state resource
    for (idx 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_AB = adjoint(gt.CTRL(gt.Xd(D), {0}, {1}, 2, D)
                           * kron(gt.Fd(D), gt.Id(D)));

    // equal probabilities of choosing a message
    idx m_A = randidx(0, D * D - 1);
    auto midx = n2multiidx(m_A, {D, D});
    cout << ">> Alice sent: " << m_A << " -> ";
    cout << disp(midx, " ") << endl;
```

Dense coding (cont.)

```
    // Alice's operation
    cmat U_A = powm(gt.Zd(D), midx[0]) * powm(adjoint(gt.Xd(D)), midx[1]);

    // Alice encodes the message
    ket psi_AB = apply(mes_AB, U_A, {0}, D);

    // Bob measures the joint system in the qudit Bell basis
    psi_AB = apply(psi_AB, Bell_AB, {0, 1}, D);

    auto measured = measure(psi_AB, gt.Id(D * D));
    cout << ">> Bob's measurement probabilities: ";
    cout << disp(std::get<1>(measured), ", ") << endl;

    // Bob samples according to the measurement probabilities
    idx m_B = std::get<0>(measured);
    cout << ">> Bob received: ";
    cout << m_B << " -> " << disp(n2multiidx(m_B, {D, D}), " ") << endl;
}
```

Dense coding output

```
>>> Starting Quantum++...
>>> Wed May 20 18:26:00 2015

>> Qudit dense coding, D = 3
>> Alice sent: 0 -> [0 0]
>> Bob's measurement probabilities: [1.0000, 0.0000, 0.0000, 0.0000,
                             0.0000, 0.0000, 0.0000, 0.0000, 0.0000]
>> Bob received: 0 -> [0 0]

>>> Exiting Quantum++...
>>> Wed May 20 18:26:00 2015
```

Teleportation

```
// Qudit teleporation
// Source: ./examples/teleportation.cpp
#include <qpp.h>
using namespace qpp;
using std::cout;
using std::endl;

int main()
{
    idx D = 3; // size of the system
    cout << ">> Qudit teleportation, D = " << D << endl;

    ket mes_AB = ket::Zero(D * D); // maximally entangled state resource
    for (idx 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 qudit state
    cout << ">> Initial state:" << endl;
    cout << disp(psi_a) << 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);
```

Teleportation (cont.)

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

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

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

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

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

Teleportation output

```
>>> Starting Quantum++...
>>> Wed May 20 18:26:03 2015

>> Qudit teleportation, D = 3
>> Initial state:
-0.0173 - 0.5525i
-0.4593 + 0.2473i
-0.6483 - 0.0448i
>> Alice’s measurement result: 3 -> [1 0]
>> 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^1 X^3
>> Bob’s final state (after correction):
          0.3055   -0.1287 + 0.2580i  0.0360 + 0.3574i
-0.1287 - 0.2580i            0.2721   0.2867 - 0.1809i
 0.0360 - 0.3574i   0.2867 + 0.1809i           0.4223
>> Norm difference: 0.0000

>>> Exiting Quantum++...
>>> Wed May 20 18:26:03 2015
```

Future directions

- Extensive unit testing.
- Rigorous comparisons with similar software.
- Better integration with third party software (currently supports only basic input/output interfacing with MATLAB).
- Additional modules for more "specialized" tasks: stabilizer states, circuit synthesizing, etc.
- Further optimization (make use of sparseness when possible)
- Perhaps a GUI interface (Qt, wxWidgets, JavaFX/Swing or anything that works and is portable).

Thank you!

NSERC
CRSNG

Industry Canada Industrie Canada
Canada