卒業論文　　2013年度 (平成25年度)

量子ビットシミュレーションのための C++ライブラリの設計と実装

慶應義塾大学 環境情報学部

氏名：村田 紘司

指導教員

慶應義塾大学 環境情報学部

村井 純

徳田 英幸

楠本 博之

中村 修

高汐 一紀

Rodney D. Van Meter III

植原 啓介

三次 仁

中澤 仁

武田 圭史

平成26 年1 月20 日

卒業論文要旨 - 2013 年度 (平成 25 年度)

量子ビットシミュレーションのための C++ライブラリの設計と実装

量子コンピューティングは現在解決できない問題を解く手段として大いに研究されている．因数分解をとくためのアルゴリズムや，探索アルゴリズム，セキュリティのための研究もされている．情報の分野では，量子を制御出来ることを前提として，量子をどのように操ればどのような計算を行うことが出来るか，といった研究が行われている．量子をシミュレーションする容易さは，そのような研究の効率に直結する．

しかし現在公開されている量子シミュレータには現在多くの問題点がある．メモリの使用量，計算量といったスケーラビリティの問題や，量子シミュレータの実装の概念における問題である．

本研究では実装の自由度を高める工夫や，疎行列や密度行列の利用により，これらの問題点を解決する量子ビットシミュレータのためのライブラリを実装した．また，このライブラリはオープンソースソフトウェアとして公開されている．

このような工夫による結果の一例を挙げる．例えば疎行列を利用すると，有名な shor のアルゴリズムで n-bit の値をシミュレーションする時，理想的にはメモリの使用量を約 $\frac{1}{2^{6n}}$ に減らせる可能性がある．

キーワード
1．量子ビット, 2．量子プログラミング, 3．量子ゲート

慶應義塾大学 環境情報学部

村田 紘司

Abstract of Bachelor's Thesis

Academic Year 2013

Designing and Implementation of C++ Library for Quantum bit Simulation

Quantum computing is being actively researched as a way to solve problems which cannot be solved today, for example, prime factorization, combinatoric search algorithms or security-related applications. In the informatics field, researchers are investigating quantum algorithms, working on the assumption that we can control individual quanta with enough precision. The ease of simulating quantum systems is directly connected to the efficiency of their research.

However, the quantum simulators which are publicly available have many problems. These include problems scalability issues caused by memory consumption and the amount of calculation. Other problems are due to the structure of the quantum simulator implementation.

This thesis describes schemes for raising the flexibility of a simulation and for using sparse matrix and density matrix. By the effect of these, this library improves on available solutions to the above problems. This library has been published as a open source software.

Examples of the results due to such schemes are included. When simulating the factoring of an n-bit number using Shor's algorithm, ideally we can reduce the memory usage by a factor of about 2^{6n} when using a sparse matrix.

Keywords :
1. Quantum Bit, 2. Quantum Programming, 3. Quantum Gate

Keio University, Faculty of Environmental Information

Koji Murata

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　序論

コンピュータは登場以来，高速化，小型化を続け，今やCPUの素子はシリコン原子の大きさに近付きつつあり，遂にムーアの法則[1]の限界に達しようとしている．[2]また，現在のコンピュータは，因数分解を代表とするような，問題の大きさに対して計算量が指数関数的に増える問題は解くことが難しい．コンピュータの速度が上昇しても，計算量を増やすことが容易だからである．

現在のコンピュータでは，1つのbitを電気信号の強さで表している．量子コンピュータは，電気信号の代わりに1つの量子を使って動かされるコンピュータである．量子はニュートン力学では説明できない2つの性質を持つ．量子重ねあわせ，量子もつれの2つである．これにより，現在のコンピュータとは異なる計算能力を持つことが出来る．1994年にShorにより上記の因数分解を効率的に行う量子コンピュータアルゴリズムが発見されている．[3]

2013年8月には，東大大学大学院工学系研究科の古澤明教授らは，1万6000個以上の量子がもつれ合った量子もつれの生成に成功したと発表した．[4]この研究により量子コンピュータの完成は大きく前進したといえるだろう．

このように，量子コンピュータの研究は現在解決できない問題を解く手段として大いに研究されている．情報の分野では，量子ビットを制御出来ることを前提として，量子ビットをどのように操ればどのような計算を行うことが出来るか，といった研究が行われている．つまり量子ビットをシミュレーションする容易さは，そのような研究の効率に直結する．

本研究では，量子ビットのシミュレーションを容易にするライブラリを実装した．

1.2　本研究の目的

本研究では，量子ビットシミュレーションのためのC++ライブラリを構築した．現在公開されている多くの量子シミュレータの問題点を4つあげる．

- 状態ベクトルで量子状態を表現しているが，これでは純粋状態のみシミュレーションでき，混合状態をシミュレーションすることができない．メモリや計算量の問題により混合状態のシミュレーションは難しいが，密度行列による表現も可能にする必要がある．

- シミュレーションの際，データをコピーしている．これは計算が単純になる代わりにメモリの領域を多く使ってしまう．出来る限りデータのコピーは控えるべきである．

- 量子ゲートがライブラリに依存しており，自由にプログラマが量子ゲートを定義，シミュレーションできないシミュレータも多い．これは，シミュレーションの簡略化，効率化がとてもしづらい．

- 量子系が 1 つしかない．これでは量子ネットワーク，量子リピータのシミュレーションが出来ない．プログラマは自由に量子系を定義，操作できなければならない．

本研究は上記 4 つの問題点を解決を目的とする．これらの問題点を解決した量子シミュレータは理想的なものであると考える．

本研究では，このような量子ビットのシミュレーションを容易に実装できる C++ ライブラリを実装する．また，このライブラリはオープンソースライセンスで第三者が利用できるように公開される．

1.3 本研究の成果

本研究では第 1.3 節で述べた目的を達成するため，C++のライブラリである Eigen ライブラリ [6] をもとに，量子ビットシミューレートのための PQS というライブラリを実装した．

これは自由に利用することが出来るよう公開されている．[7]

1.4 本論文の構成

本論文は，7 章から構成される．第 2 章では，本研究の背景となる要素技術を整理する．第 3 章では，本研究の元となった既存研究である Quantum Computing Language についての研究とその問題点を述べ，第 4 章では，本研究によって実装されたライブラリの使い方を解説する．第 5 章では本ライブラリの詳しい実装方法をまとめる．第 6 章では本研究のアイデアと実装したライブラリについて評価し，第 6 賞では本論文の結論と，今後の方針を述べる．

付録では，インストール方法と実際の実装例を解説する．

本研究により実装されたライブラリを利用して量子シミュレータを実装しようとするプログラマは第 4 章と付録 A を参照することを勧める．ライブラリの拡張や変更をしようとするプログラマは加えて第 5 章を参照されたい．

第2章　要素技術

本研究は，量子ビットをシミューレートするためのC++ライブラリの制作を目標とする．本章ではまず，量子，量子ビット，量子プログラミングについて説明する．

2.1　量子

量子とは，量子重ね合わせ，量子もつれといった量子状態を取ることの出来る粒子のことであり，光子，電子を代表例として，炭素原子，水分子等多岐にわたる．

ここでは，量子の持つ最も重要な性質である，量子重ね合わせと量子もつれについて解説する．

2.1.1　量子重ね合わせ

量子重ね合わせとは，量子の持つ最も重要な性質のうちの1つである．量子は2つ以上の状態を同時に取ることが出来る．それは量子ビットとして表現すれば0であり，かつ1でもある状態，ということである．

ニュートン力学において，完全にランダムな事象は存在しない．決定論と呼ばれる考え方である．例えば，コインを1枚投げて手の裏に隠したとき，50%の確率で表，若しくは裏であるが，見て確認する前からどちらであるかは決まっている．

しかし量子力学においては，もしコインが量子であり，コインにおいて量子重ね合わせ状態を作れたと仮定するならば，コインを見て確認するまで表，裏のどちらであるかは決まっていない．見た瞬間に重ね合わせ状態が壊れ，決定する．その決定にはなんの因果もなく，完全なるランダムである．これが量子重ね合わせである．

2.1.2　量子もつれ

量子もつれとは，量子重ね合わせに並んで重要な性質の1つである．引き続きコインを例に挙げると，ニュートン力学ではコインを2枚投げた時2つのコインの表裏は別の事象であり，それぞれの確率同士は影響しあわない．

しかし，仮に2つのコインを量子として，もつれさせることが出来たならば，2つのコインの事象は独立ではなくなる．表表・表裏・裏表・裏裏，4つのそれぞれの事象に対して別の確率を持つことになり，例えば表表の確率が50%，裏裏の確率が50%の系を作った

時，片方のコインを確認して表だった場合，もう片方のコインも表であることが決定する．裏の場合もまたしかりである．

2.2 量子ビット

量子の性質を利用して情報を扱うことを量子コンピューティングという．現在使われているコンピュータ (古典コンピュータ) は，電気信号の電圧の強弱によって 1 つのビットを表現している．量子コンピューティングにおいては，電気信号の代わりに量子を用いる．

例えば光子であれば，偏光によって情報を扱う．縦向き偏光であれば 0，横向き変更であれば 1．電子であれば，時計回りのスピンであれば 0，反時計回りであれば 1，といった具合である．これを量子ビットという．

量子ビットは，純粋状態と混合状態の 2 つの状態を取る．純粋状態では，その系が原理的に持ちうる情報を得られている状態であり，理想的な量子状態と言える．混合状態では，複数の純粋状態が古典的に混ざり合っており，重ね合わせがすでに壊れているような状態も含む．

それぞれの状態を表現するための数学的手法を紹介する．状態ベクトルは純粋状態を表現するためのものであり，密度行列は混合状態を表すためのものである．

2.2.1 状態ベクトル

$$|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \tag{2.1}$$

数式 (2.1) が状態ベクトルの記法である．$|\psi\rangle$ は一般的な量子系を表している．この量子系は 1 つの量子ビットを有しており，$\alpha\beta$ はそれぞれ量子ビットが 0，1 である状態を表している．

$$|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix} \tag{2.2}$$

数式 (2.2) では 2 つの量子ビットをもつ量子系を表している．$\alpha\beta\gamma\delta$ は 2 つの量子ビットがそれぞれ 00，01，10，11 である状態を表している．

2.2.2 密度行列

$$\rho = \begin{pmatrix} \alpha\alpha^* & \alpha\beta^* \\ \beta\alpha^* & \beta\beta^* \end{pmatrix} \tag{2.3}$$

$$\rho = \begin{pmatrix} \alpha\alpha^* & \alpha\beta^* & \alpha\gamma^* & \alpha\delta^* \\ \beta\alpha^* & \beta\beta^* & \beta\gamma^* & \beta\delta^* \\ \gamma\alpha^* & \gamma\beta^* & \gamma\gamma^* & \gamma\delta^* \\ \delta\alpha^* & \delta\beta^* & \delta\gamma^* & \delta\delta^* \end{pmatrix} \tag{2.4}$$

密度行列では混合状態を表すため，1 つの純粋状態を表すこともできる．数式 (2.3)(2,4) はそれぞれ (2.1)(2.2) を密度行列で表したものである．

本研究の実装では，量子ビットを表現するのに，暫定的に密度行列を採用しているが，純粋状態のみのシミューレートをしたい場合はメモリと計算量の節約のため状態ベクトルを採用すべきである．

2.3 疎行列

疎行列とは，要素のほとんどが 0 である行列のことである．これをプログラムで表す場合，多くの 0 である要素に対してメモリを確保しないことでメモリを節約できる．

この時，メモリ上にはデータが順番に並ばないため，値とともに行列のどこの部分に当たる値であるかの情報を保存しておく必要がある．このため，疎行列を想定したプログラムを使った場合，全ての要素が 0 でない最悪の状態では，メモリを多く使用するとともに，演算速度も遅くなる．

最も一般的な疎行列として I を数式 2.5 に示す．

$$I = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.5}$$

この行列を疎行列を想定せず，すべてにメモリを確保した場合，$sizeof(element) \times 6^2$ のメモリが必要になる．対して，想定してプログラムを作った場合，$sizeof(element) \times 6 + sizeof(int) \times 2 \times 6$ が必要なメモリ量となる．このように，実際に多くの要素が 0 だった場合は多くのメモリを節約することができる．

疎行列を想定することでどの程度のメモリを節約できるかを示す例をもう 1 つ挙げる．

Shor のアルゴリズム [3] は最も有名な量子アルゴリズムの 1 つであり，ある任意の値の素因数分解を行うアルゴリズムである．このアルゴリズムの量子回路は多くの量子ビットを利用する．

具体的には，N を素因数分解する場合，N を 2 進数で表現するために必要な bit 数は $\lceil log_2 N \rceil$ であるが，これを n とすると，この値の素因数分解には，値の入力用に $2n$，出力用に n，一時保存用に $2n$ 個の量子ビットを必要とする．

これを密度行列で表すと，2^{10n} もの要素が必要となる．だが，この密度行列の 0 でない値は，値を入力した時点で 2^{4n}，計算することで多少増えていくが，2^{10n} に比べればとても少ない．入力した時点での要素数を比べれば，$\frac{1}{2^{6n}}$ にメモリを節約できていることになり，疎行列の有用性は明らかである．ただしこれは理想的な場合であり，第 6.1.2 節で述べるが，Eigen ライブラリでは $\frac{1}{2^{5n}}$ になることが予想される．

本節の冒頭でも述べたが，疎行列を想定した実装の場合では，0 でない値が多くなると，メモリは無駄に使い，演算速度は遅くなってしまう．量子シミュレーションの中でも，エラーのシミュレーションではこのような状況が起こりやすい．行列の 0 であるべき要素に，0 の代わりに小さな値がノイズとして入ってしまうようなケースである．この場合，すべての 0 であるべき要素が小さな同じ値が入ってしまう場合が多く，このような実装ではデメリットが目立ってしまう．

2.4 量子ゲート

量子コンピューティングを行うためには量子ビットを制御する必要がある．古典コンピュータに NOT，OR，AND 等様々なゲートがあるように，量子ゲートも様々なものがある．

量子ゲートは理論的にはユニタリ行列で表される．1 量子ビットにかけるゲートは 2×2，2 量子ビットにかけるゲートは 4×4 のユニタリ行列になる．これを量子系にかけることで量子ゲートをかけることになる．

ここでは，最も基本的な量子ゲートを 2 つ紹介する．

2.4.1 CNOT ゲート

CNOT ゲートは，最も基本的な量子ゲートの 1 つであり，controled NOT gate の略である．2 つの量子ビットにかけられるゲートであり，2 つの量子ビットはそれぞれコントロールビットとターゲットビットと呼ばれ，コントロールビットが 1 だった場合にターゲットビットに NOT をかけるゲートである．演算の前後を表 2.1 に示す．

表 2.1: CNOT ゲート

演算前		演算後	
コントロール	ターゲット	コントロール	ターゲット
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

また，行列で CNOT ゲートを表したのが数式 2.6 と数式 2.7 である．数式 2.6 では右のビットがコントロールビット，左のビットがターゲットビットとなる．数式 2.7 では逆に左のビットがコントロールビット，右のビットがターゲットビットである．

$$CNOT_{0,1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \tag{2.6}$$

$$CNOT_{1,0} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.7}$$

数式 2.8 には数式 2.7 で示したゲートをかける例を示す．α_n は量子ビットが n である状態を表す．この場合，α_{10} と α_{01} が入れ替わっていることが分かる．

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{pmatrix} = \begin{pmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{11} \\ \alpha_{10} \end{pmatrix} \tag{2.8}$$

2.4.2 Hadamard ゲート

Hadamard ゲートも，最も基本的な量子ゲートの 1 つである．1 つの量子ビットにかける Hadamard ゲートを数式 2.9 に示す．

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{2.9}$$

また，Hadamard ゲートは n 個の量子ビットにかけることができ，その場合は数式 2.8 をテンソル積で n 乗したものが Hadamard ゲートとなる．2 個の量子ビットにかける Hadamard ゲートを数式 2.10 に示す．

$$H^{\otimes 2} = H \otimes H = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \tag{2.10}$$

数式 2.11 では，$|00\rangle$ の状態の 2 個の量子ビットに Hadamard ゲートをかける例を示す．

4 つの量子ビットのステートが同じになっていることがわかる．

$$H^{\otimes 2}|00\rangle = \frac{1}{2}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \end{pmatrix} = \frac{1}{2}\left(|00\rangle + |01\rangle + |10\rangle + |11\rangle\right) \tag{2.11}$$

hadamard ゲートの行列を生成するプログラムを作成し，下記 URL に公開している．
`https://github.com/malt03/hadamard`

2.5 量子プログラミング

2.5.1 量子プログラミング言語

量子コンピュータでは古典コンピュータとはアーキテクチャが根本から異なるため，新しいプログラミング言語が必要である．量子プログラミング言語は，古典コンピュータにおけるプログラミング言語の代わりに，量子コンピュータにおいてプログラミングするための言語である．量子アセンブラとも言える言語が考案されており，シミュレータと共に多数開発されている．

しかし，量子コンピュータにおけるメモリの形態も分からず，また，それぞれの系において複数の量子ビットがもつれ合うため，古典コンピュータと同じような書式での分かりやすい量子アセンブラの開発は難しい．

系を 1 つのクラスとして扱い，そのクラスに対して引数として，量子ゲートとゲートをかける量子ビットの番号を与えることで量子アセンブラは完成するのではないか．そういった考えから本研究は開始された．

本研究は，量子ビットシミュレータのライブラリ制作だが，量子プログラミング言語の基礎的な概念を表しているものにもなっている．

2.5.2 QCL

本研究を始めるにあたって，既存研究として Quantum Computing Language(QCL)[5] を参照した．QCL では，第 2.4.1 節で述べた量子アセンブラ及び QCL を使うための量子ビットシミュレータを開発している．

本研究は，QCL のシミュレータを並列化する試みからスタートしている．第 3 章では，その概要と結果を示す．

第3章　QCL

3.1　概要

第2.4.2節でも述べたが，Quantum Computing Language(QCL)では，量子アセンブラとQCLを使うための量子ビットシミュレータを開発している．本章では，QCLにおける量子ビットシミュレータをOpenMPによる並列化で高速化しようとした試みを紹介する．

ソースコード 3.1: QCLの例

```
1  qureg a[2];
2  qureg b[1];
3  qureg c[1];
4  reset;
5
6  H(a);
7  CNot(c,b);
8  CNot(b,a);
9
10 dump;
```

ソースコード3.1では，量子レジスタを3つ定義し，Hadamardゲートと Control Notゲートをかけるシミュレートを QCLによって実装した例を示している．

ソースコード 3.2: Hadamardゲート

```
1  int n = 1;
2  qureg q[n];
3  H(q);
```

ソースコード3.2では，Hadamardゲートのみをかけた例を示している．まず，このプログラムにおいてnを増やしていったときの速度を測定した．その結果を図3.1に示す．

図3.1を見るとわかるように，29量子ビットまでは想定通り指数関数的に計算時間が増えている．しかし，30量子ビット時において，増え方が想定よりも減り，その後は減少している．これは，コンピュータのメモリをシミュレータが使い果たしたためである．

3.2　並列化

ソースコード3.3は，QCLにおける量子ビットシミュレータを並列化しようとしたソースコードである．

gprofによってopElementary::applyが参照回数も多く，最も時間を使っていることが分かったので，この関数を高速化しようとしている．

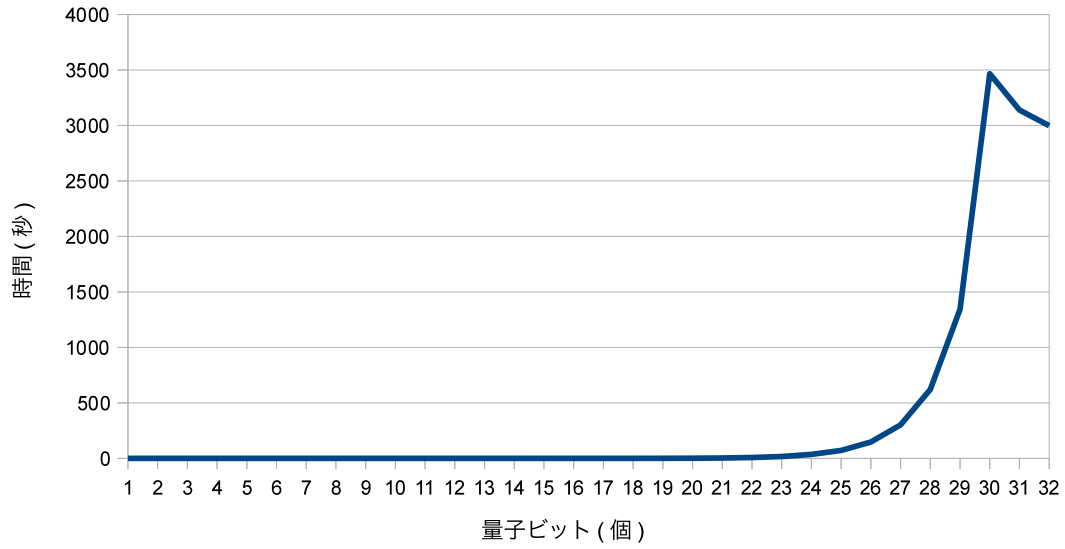

図 3.1: Hadamard ゲート

ソースコード 3.3: QCL シミュレータの並列化

```
/* opElementary */
void opElementary::apply(quState& qs) const {
  int i,n;
  terminfo ti;
  bitvec m;

  if(!qs.mapbits()) return;
  qs.opbegin();
  ti.pqs=&qs;
  m=~qs.mapmask();
  n=qs.baseterms();

#pragma omp parallel for
  for(i=0;i<n;i++) {
  terminfo tmp = ti;
  tmp.mapterm=qs.baseterm(i);
  tmp.frame=tmp.mapterm.vect() & m;
  tmp.mapterm.setvect(qs.map(tmp.mapterm.vect()));
  addterms(tmp);
  };
  qs.opend();
 }
```

3.3 評価

評価は 23 量子ビットに Hadamard ゲートをかけるプログラムを用いた．誤差が少なく，時間もかかり過ぎないと考えたのでこの個数を選んだ．図 3.2 で，スレッド数を増やしていった結果を示す．

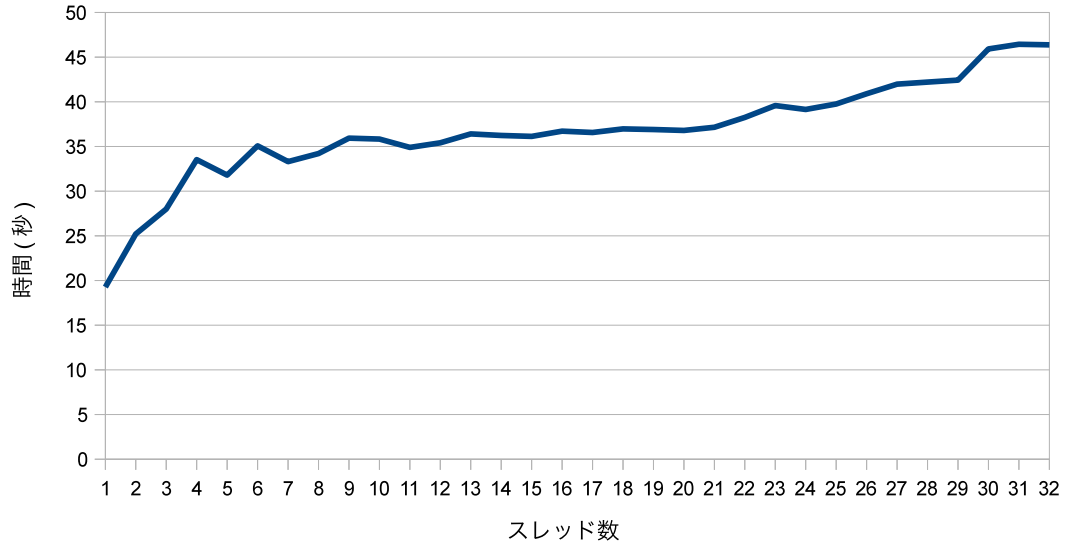

図 3.2: QCL 並列化

スレッドを増やすと計算時間が増えてしまっていることが分かる．今回の実験は失敗に終わったと言える．

3.4 QCL の問題点

3.4.1 並列化の難しさ

第 3.2 節で述べたが，この関数は最も時間を使っているが，最も参照回数も多い．これは，並列化においては不利なことである．なぜなら参照回数が多いということはそれだけ 1 度の参照にかかる時間は少ないからである．

並列化は理想的には，それぞれのスレッドが長く計算し，スレッド同士は干渉しあわない．今回のテストでは，並列化は失敗した．QCL はもともと並列化を想定しておらず，もっとも時間のかかっている関数では，並列化してもお互いに干渉し，計算時間も短いからである．

並列化するためには初めから並列化を想定して実装しておく必要がある．

3.4.2 メモリの使用量

QCL では，演算をする際多くのデータをコピーしている．これは演算を簡単にするためであるが，そのために多くのメモリを使用している．

また，疎行列を想定しており，キャッシュを使ったプログラムを構築しようとしているが，キャッシュとデータが 1 対 1 対応しているためにキャッシュによる検索スピードの向上というメリットを生かせておらず，結果としてメモリもあまり節約できていないという結果になってしまっている．

3.4.3 密度行列

状態ベクトルによるシミュレーションになっており，密度行列を使うことができない．これはメモリの節約にはなるものの，シミュレーション出来る量子状態に限りが出来てしまう．

3.4.4 量子ゲート

予め実装されている量子ゲートはとても使いやすく実装されているのだが，自分で作成した自由な量子ゲートをかけるシミュレーションが出来ない．これは，シミュレーションの自由度を著しく下げる結果となっている．

3.4.5 量子系の定義

QCL では，全ての量子ビットが同じ系にあるように定義されている．つまり，1 つの量子ビットを定義した場合 $|0\rangle$ となり，さらに 1 つ定義すると $|00\rangle$，さらに定義すると $|000\rangle$ となる．

このように，別の系を定義できずに 1 つの系の中でしかシミュレーションができない．これは，量子ネットワーク，量子リピータのように，複数の系が関係するようなをシミュレーションするときに問題となる．

第4章　ライブラリの使い方

本章では，PQS の使い方を解説する．

4.1　実装されたクラス

表 4.1: 実装されたクラス

qSystem Class	量子系クラス
qubit Class	量子ビットクラス
qOperator Class	量子ゲートクラス
cmplx Class	コンプレックスクラス
sMatrix Class	疎行列クラス
T Class	Triplet クラス

ユーザーが使うクラスは表 4.1 に示された 6 つである．量子ビットクラスは量子系クラスのコンストラクタに渡すためのクラスなので，コンストラクタしか持たない．

コンプレックスクラスは，complex<double> を typedef したものであり，Triplet クラスは Triplet<cmplx> を typedef したものである．また，量子ゲートクラス，疎行列クラスは，Eigen ライブラリ [6] の SparseMatrix<cmplx> を typedef したものである．

4.2　メンバー関数

量子系クラスが持つメンバー関数を表 4.2，4.3 に，量子ビットクラスが持つメンバー関数を表 4.4 に示す．また，その他の使用できる関数を表 4.5 に示す．

表 4.2: 量子系クラス 1

qSystem()
デフォルトコンストラクタ．配列を定義するときに使用する． この後引数付きのコンストラクタで再定義しなければならない．
qSystem(const int qubit_num)
コンストラクタ．qubit_num に指定された数の量子ビットを持った 量子系を作成する．すべての量子ビットは 0 になる．
qSystem(const int qubit_num, const qubit qubits[])
コンストラクタ．qubit_num に指定された数の量子ビットを持った 量子系を作成する．量子ビットの状態は qubits[] で指定された状態になる．
qSystem(const qSystem* source)
コンストラクタ．source で指定された qSystem をコピーする．
qSystem(const sMatrix* value)
コンストラクタ．value で指定された行列の状態を持った量子系を作成する．
const sMatrix* getMatrix()
量子系の状態を表す行列を返す．
qSystem* calculate(qOperator& q_operator)
q_operator で指定されたゲートを量子系にかける． ope の大きさは量子系が持つ n 個の量子ビットに対して $2^n \times 2^n$ でなければならない．this を返す．
qSystem* calculate(qOperator& q_operator, int* positions, int q_operator_len)
q_operator で指定されたゲートを positions ポインタで指定された配列が 示す場所にかける．q_operator_len でゲートをかける量子ビットの個数を 指定する．それは positions の長さと同じ値になる． positions は数値が昇順にソートされていなければならない．this を返す．
float getProbOf(int index, int a, bool isX=false)
index で指定された場所の値が a である確率を返す． isX を true に指定すると X 軸で測定する．
float* getProbs(int index, float* answer, bool isX=false)
index で指定された場所の値の確率をすべて answer に入れる． isX を true に指定すると X 軸で測定する． answer のポインタを返す．
qSystem* measureTo(int index, int a, bool isX=false)
index で指定された場所の値を a の値になるように測定する． isX を true に指定すると X 軸で測定する． x になる確率が 100%の測定．this を返す．

表 4.3: 量子系クラス 2

qSystem* measurement(int index, bool isX=false)
index で指定された場所を実際の確率にそって測定する． isX を true に指定すると X 軸で測定する．this を返す．
qSystem* measurement(bool isX=false)
量子系を全て実際の確率にそって測定する． isX を true に指定すると X 軸で測定する．this を返す．
const qSystem* tensor(qSystem* answer, const qSystem& psi)
psi に指定された量子系とのテンソル積を answer に入れ込む． answer のポインタを返す．

表 4.4: 量子ビットクラス

qubit()
デフォルトコンストラクタ．配列を定義するときに使用する． この後引数付きのコンストラクタで再定義しなければならない．
qubit(const cmplx a, const cmplx b)
コンストラクタ．1 つの量子ビットの状態を表す状態ベクトルの $\|0\rangle$ の状態を表す値を a に，$\|1\rangle$ の 状態を表す値を b に指定する．

表 4.5: その他の関数

ostream& operator<<(ostream& os, const qSystem psi)
os に psi で指定された量子ゲートクラスの密度行列を文字列として 追加して返す．
qOperator* tensor(qOperator* answer, qOperator& x, int x_len, qOperator& y, int y_len)
x と y のオペレーターのテンソル積を answer のポインタに入れる． 正方行列同士のテンソル積のみに対応している． x_len，y_len はそれぞれ x と y の行，列の長さを指定する． answer のポインタを返す．

4.3 量子系の概念

PQS を利用するにあたって最も理解すべき概念が，量子系の概念である．PQS において量子系はひとつのクラスとして表現される．プログラマは量子系を任意の数作成できる．

量子系が持つ量子ビットの個数はコンストラクタによって初期化されなければならない．配列を利用するために空のコンストラクタは用意されているが，その後必ず量子ビットの個数を new 演算子を使ってコンストラクタによって指定する．

また，量子ビットのそれぞれの状態の初期化は主に qubit Class の配列をコンストラクタに引数として渡すことで行われる．密度行列や，qSystem Class をそのまま引数として渡す事もできる．引数を渡さず，量子ビットの個数のみを渡した場合はすべて 0 として初期化される．

量子ビットの個数は初期化後の変更は許されず，必要とする場合初期化の時に 0 を指定した量子ビットを初めから追加しておく必要がある．

また，qubit Class のコンストラクタには，1 つの量子ビットの状態を表した状態ベクトルをそのまま渡すことで初期化する．

ソースコード 4.1 において量子系クラスの初期化，及び標準出力への出力の実装例を示す．

ソースコード 4.1: 量子系初期化の実装例

```
1  #include <iostream>
2  #include "pqs"
3
4  using namespace std;
5  using namespace pqs;
6
7  int main(){
8    qubit value[] = {
9      qubit(sqrt(cmplx(0.5,0.0)), sqrt(cmplx(0.5,0.0))),
10     qubit(sqrt(cmplx(0.3,0.0)), sqrt(cmplx(0.7,0.0))),
11     qubit(sqrt(cmplx(0.3,0.0)), sqrt(cmplx(0.7,0.0))),
12   };
13   qSystem psi(3,value);
14
15   cout << psi << endl;
16 }
```

4.3.1 量子ゲートの操作

量子ゲートはプログラマが自由に定義できるように実装されている．qOperator Class の初期化では行列の大きさを渡す．値のセットは Triplet Class の vector を専用の関数に渡すことによってなされる．これは，Eigen ライブラリの疎ベクトル用の実装によっている．

これにより qOperator の初期化は疎ベクトルを利用するために少し長くなる．プログラマは量子ゲートを定義する専用のヘッダを作成することが推奨される．

実際の計算は qOperator Class とゲートをかける量子ビットを指定した値を，qSystem Class の calculate 関数に渡す．

calculate 関数の引数は，第 1 引数から順に，量子ゲート，ゲートをかける量子ビットを示した配列のポインタ，第 2 引数の長さ，をそれぞれ指定する．

ソースコード 4.2-4.7 にそれぞれ 1 量子ビット，2 量子ビット，3 量子ビットの場合の実装例とその出力結果を示す．

1 量子ビットでは数式 2.9 で示した Hadamard ゲートを，2 量子ビットでは数式 2.7 で示した CNOT ゲートを，3 量子ビットでは Toffoli ゲートという CNOT ゲートに更にコントロールビットが追加されたゲートをシミュレーションする例である．

ソースコード 4.2: 量子ゲートの実装例 1Hadamard

```
#include <iostream>
#include <vector>
#include "pqs.h"

using namespace std;
using namespace pqs;

int main(){
  // first, create the matrix for the Hadamard operator
  double r2 = 1.0/sqrt(2.0);
  vector<T> tripletList;
  tripletList.push_back(T(0,0,cmplx(r2,0.0)));
  tripletList.push_back(T(1,0,cmplx(r2,0.0)));
  tripletList.push_back(T(0,1,cmplx(r2,0.0)));
  tripletList.push_back(T(1,1,cmplx(-r2,0.0)));
  qOperator H(2,2);
  H.setFromTriplets(tripletList.begin(), tripletList.end());

  // create a system with three qubits
  qubit value[] = {
    qubit(sqrt(cmplx(1.0,0.0)), sqrt(cmplx(0.0,0.0))),
    qubit(sqrt(cmplx(1.0,0.0)), sqrt(cmplx(0.0,0.0))),
    qubit(sqrt(cmplx(1.0,0.0)), sqrt(cmplx(0.0,0.0))),
  };
  qSystem psi(3,value);

  // pos[] defines which a qubit we will execute the H on
  int pos[] = {1};

  cout << "before:\n" << psi << endl;
  psi.calculate(H, pos, 1);
  cout << "after:\n" << psi << endl;
}
```

ソースコード 4.3: 量子ゲートの実装例 1 の出力結果

```
before:
Nonzero entries:
((1,0),0)

Outer pointers:
0 1 1 1 1 1 1 1  $

(1,0) 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0

after:
Nonzero entries:
((0.5,0),0) ((0.5,0),2) ((0.5,0),0) ((0.5,0),2)

Outer pointers:
0 2 2 4 4 4 4 4  $

(0.5,0) 0 (0.5,0) 0 0 0 0 0
0 0 0 0 0 0 0 0
(0.5,0) 0 (0.5,0) 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

ソースコード 4.4: 量子ゲートの実装例 2 CNOT

```
#include <iostream>
#include <vector>
#include "pqs.h"

using namespace std;
using namespace pqs;

int main(){
  // first, create the matrix for the CNOT operator
  vector<T> tripletList;
  tripletList.push_back(T(0,0,cmplx(1.0,0.0)));
  tripletList.push_back(T(1,1,cmplx(1.0,0.0)));
  tripletList.push_back(T(3,2,cmplx(1.0,0.0)));
  tripletList.push_back(T(2,3,cmplx(1.0,0.0)));
  qOperator CNOT(4,4);
  CNOT.setFromTriplets(tripletList.begin(), tripletList.end());

  // create a system with three qubits
  qubit value[] = {
    qubit(sqrt(cmplx(0.5,0.0)), sqrt(cmplx(0.5,0.0))),
    qubit(sqrt(cmplx(1.0,0.0)), sqrt(cmplx(0.0,0.0))),
    qubit(sqrt(cmplx(0.7,0.0)), sqrt(cmplx(0.3,0.0))),
  };
  qSystem psi(3,value);

  // pos[] defines which two qubits we will execute the CNOT on
  int pos[] = {0,2};

  cout << "before:\n" << psi << endl;
  psi.calculate(CNOT, pos, 2);
  cout << "after:\n" << psi << endl;
}
```

ソースコード 4.5: 量子ゲートの実装例 2 の出力結果

```
before:
Nonzero entries:
((0.35,0),0) ((0.229129,0),1) ((0.35,0),4) ((0.229129,0),5) ((0.229129,0),0) ((0.15,0),1)
     ((0.229129,0),4) ((0.15,0),5) ((0.35,0),0) ((0.229129,0),1) ((0.35,0),4)
    ((0.229129,0),5) ((0.229129,0),0) ((0.15,0),1) ((0.229129,0),4) ((0.15,0),5)

Outer pointers:
0 4 8 8 8 12 16 16  $

(0.35,0) (0.229129,0) 0 0 (0.35,0) (0.229129,0) 0 0
(0.229129,0) (0.15,0) 0 0 (0.229129,0) (0.15,0) 0 0
```

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
(0.35,0) (0.229129,0) 0 0 (0.35,0) (0.229129,0) 0 0
(0.229129,0) (0.15,0) 0 0 (0.229129,0) (0.15,0) 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

after:
Nonzero entries:
((0.35,0),0) ((0.229129,0),1) ((0.229129,0),4) ((0.35,0),5) ((0.229129,0),0) ((0.15,0),1)
     ((0.15,0),4) ((0.229129,0),5) ((0.229129,0),0) ((0.15,0),1) ((0.15,0),4)
    ((0.229129,0),5) ((0.35,0),0) ((0.229129,0),1) ((0.229129,0),4) ((0.35,0),5)

Outer pointers:
0 4 8 8 8 12 16 16  $

(0.35,0) (0.229129,0) 0 0 (0.229129,0) (0.35,0) 0 0
(0.229129,0) (0.15,0) 0 0 (0.15,0) (0.229129,0) 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
(0.229129,0) (0.15,0) 0 0 (0.15,0) (0.229129,0) 0 0
(0.35,0) (0.229129,0) 0 0 (0.229129,0) (0.35,0) 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

ソースコード 4.6: 量子ゲートの実装例 3 Toffoli

```
#include <iostream>
#include <vector>
#include "pqs.h"

using namespace std;
using namespace pqs;

int main(){
  // first, create the matrix for the Toffoli operator
  vector<T> tripletList;
  tripletList.push_back(T(0,0,cmplx(1.0,0.0)));
  tripletList.push_back(T(1,1,cmplx(1.0,0.0)));
  tripletList.push_back(T(2,2,cmplx(1.0,0.0)));
  tripletList.push_back(T(3,3,cmplx(1.0,0.0)));
  tripletList.push_back(T(4,4,cmplx(1.0,0.0)));
  tripletList.push_back(T(5,5,cmplx(1.0,0.0)));
  tripletList.push_back(T(6,7,cmplx(1.0,0.0)));
  tripletList.push_back(T(7,6,cmplx(1.0,0.0)));
  qOperator Toffoli(8,8);
  Toffoli.setFromTriplets(tripletList.begin(), tripletList.end());

  // create a system with three qubits
  qubit value[] = {
    qubit(sqrt(cmplx(0.5,0.0)), sqrt(cmplx(0.5,0.0))),
    qubit(sqrt(cmplx(0.0,0.0)), sqrt(cmplx(1.0,0.0))),
    qubit(sqrt(cmplx(0.3,0.0)), sqrt(cmplx(0.7,0.0))),
  };
  qSystem psi(3,value);

  // pos[] defines which three qubits we will execute the Toffoli on
  int pos[] = {0,1,2};

  cout << "before:\n" << psi << endl;
  psi.calculate(Toffoli, pos, 3);
  cout << "after:\n" << psi << endl;
}
```

ソースコード 4.7: 量子ゲートの実装例 3 の出力結果

```
before:
Nonzero entries:
((0.15,0),2) ((0.229129,0),3) ((0.15,0),6) ((0.229129,0),7) ((0.229129,0),2) ((0.35,0),3)
     ((0.229129,0),6) ((0.35,0),7) ((0.15,0),2) ((0.229129,0),3) ((0.15,0),6)
    ((0.229129,0),7) ((0.229129,0),2) ((0.35,0),3) ((0.229129,0),6) ((0.35,0),7)

Outer pointers:
0 0 0 4 8 8 8 12  $

0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 (0.15,0) (0.229129,0) 0 0 (0.15,0) (0.229129,0)
0 0 (0.229129,0) (0.35,0) 0 0 (0.229129,0) (0.35,0)
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 (0.15,0) (0.229129,0) 0 0 (0.15,0) (0.229129,0)
0 0 (0.229129,0) (0.35,0) 0 0 (0.229129,0) (0.35,0)

after:
Nonzero entries:
((0.15,0),2) ((0.229129,0),3) ((0.229129,0),6) ((0.15,0),7) ((0.229129,0),2) ((0.35,0),3)
     ((0.35,0),6) ((0.229129,0),7) ((0.229129,0),2) ((0.35,0),3) ((0.35,0),6)
    ((0.229129,0),7) ((0.15,0),2) ((0.229129,0),3) ((0.229129,0),6) ((0.15,0),7)

Outer pointers:
0 0 0 4 8 8 8 12  $

0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 (0.15,0) (0.229129,0) 0 0 (0.229129,0) (0.15,0)
0 0 (0.229129,0) (0.35,0) 0 0 (0.35,0) (0.229129,0)
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 (0.229129,0) (0.35,0) 0 0 (0.35,0) (0.229129,0)
0 0 (0.15,0) (0.229129,0) 0 0 (0.229129,0) (0.15,0)
```

第5章　実装

5.1　動作環境

本研究による実装の動作確認がされた環境を表 4.1 に示す.

表 5.1: 動作環境

	環境 1	環境 2
OS	Mac OS X	Cent OS
コンパイラ	clang 4.1	gcc 4.6.0
sizeof(int)	8	8
sizeof(complex<double>)	16	16
Eigen Library	Eigen 3.1.4	Eigen 3.1.4

5.2　設計要件

本実装に必要な機能を以下にまとめる．まず，量子ビットシミュレータとして必要な最低限の機能は下記の通りである．　まず，機能 1 により量子系を定義し，機能 2 によって

表 5.2: 本研究の実装におけるシステム要件一覧

要件	目的
量子系クラス	量子系の定義と複数量子系のシミュレーション
量子ビットクラス	量子系の定義
量子ゲートクラス	量子系のシミュレーション
計算関数	量子系のシミュレーション

定義した量子ビットを定義した量子系に追加する．その後，予め機能 3 により定義された量子ゲートクラスを用いて機能 4 により計算する．以上の機能を達成すれば，こうして量子ビットのシミュレートが可能である．

5.3 Eigen ライブラリ

本実装では Eigen ライブラリの SparseMatrix クラスを使用している．ここでは，Eigen ライブラリの疎行列の実装について解説する．

5.3.1 ストレージの構造

Eigen ライブラリは CompressedStorage というクラスを持つ．これが Eigen ライブラリの核となるクラスである．ここにデータを入れ込む．CompressedStorage は 1 次元のデータ構造であり，可変長配列に近い．

CompressedStorage は下記 4 つのデータを持つ．

- m_values
- m_indices
- m_size
- m_allocatedSize

まず，m_values は，データの実際の内容である．Template で定義されているため，任意のデータ型を用いることができる．

m_indices は，m_values の値が入っている場所である．m_indices は昇順にソートされている．表 5.2 は，float 型のデータを CompressedStorage に入れた例である．

表 5.3: CompressedStorage 例

m_indices	m_values
0	30.5
2	21.23
12	-1.93
13	0.0
21	94.21

m_size は m_values の論理的なサイズである．データ型は size_t である．表 5.2 の例では，21 以上のプログラマが定義した数になる．

m_allocatedSize はデータを確保した物理的なサイズである．データ型は m_size と同じく size_t であり，表 5.2 の例では，5 となる．

このクラスは，値を入れる，値を間に入れてそれより上を 1 つずつずらす，reallocate する以外には機能を持たない．

5.3.2 疎ベクトルクラス

疎行列クラスを見る前に，より容易な疎ベクトルクラスについて解説する．

疎ベクトルクラスは，CompressedStorage クラスをほぼそのまま使っている．ただ，$1 \times n$ のベクトルも，$n \times 1$ のベクトルも使うことができるように設計されている．コンストラクタは SparseVector(Index rows, Index cols) のように 2 つの引数を持つ物も用意されているが，引数のうちの片方は 1 でなければならない．

出力する関数を含めて operator が使えることと，IsColVector というモジュールによって横向きのベクトルか縦向きのベクトルかを管理されている以外は，CompressedStorage に関数を通してそのまま値を渡しているだけだが，これによって CompressedStorage の実用的な使い方を見ることができる．

5.3.3 疎行列クラス

疎行列クラスは疎ベクトルクラスをそのまま更に配列化したものではない．疎行列クラスが持つデータを下記に示す．

- m_outerSize
- m_innerSize
- m_outerIndex
- m_innerNonZeros
- m_data

m_data は CompressedStorage クラスであり，データの内容を持つ．CompressedStorage は 1 つしか持たず，2 次元にしているわけではない．

疎行列クラスは IsRowMajor というモジュールで縦に長い行列か，横に長い行列かを管理している．これによって，横に長い行列のときに縦に割ってしまってメモリを無駄に使うことが無いように設計してある．

outer が，行列の長い方の要素を示し，inner が短い方の要素を示している．つまり，m_outerSize は，IsRowMejor が true であれば rows() の返り値となり，cols() の返り値は m_innerSize になる．

ここでは，要素の値を返す関数である coeff 関数を使って更に解説する．

ソースコード 5.1: coeff 関数

```
1 inline Scalar coeff(Index row, Index col) const
2 {
3     const Index outer = IsRowMajor ? row : col;
4     const Index inner = IsRowMajor ? col : row;
5     Index end = m_innerNonZeros ? m_outerIndex[outer] + m_innerNonZeros[outer] :
          m_outerIndex[outer+1];
6     return m_data.atInRange(m_outerIndex[outer], end, inner);
7 }
```

この関数は，row，col の引数を渡すと，その場所の値を返す関数である．3，4 行目では row と col がそれぞれ outer と inner のどちらかを決めている．

m_innerNonZeros は，その outer の値の入っている数を記録している．もし値がなければ null ポインターになっている．m_outerIndex はその outer の値がある初めの場所の index を記録している．これを m_data のメンバー関数に渡すことで，場所を計算して値を返している．

例として数式 5.1 のような 5×5 の行列を考える．

$$\begin{pmatrix} 0 & 3 & 0 & 0 & 0 \\ 22 & 0 & 0 & 0 & 17 \\ 7 & 5 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 8 \end{pmatrix} \tag{5.1}$$

この時のそれぞれのデータの内容を表 5.4 に示す．

表 5.4: CompressedStorage 例

データ名	内容
m_outerIndex	0, 2, 4, 5, 6
m_innerIndex	1, 2, 0, 2, 4, 2, 1, 4
m_data	22, 7, 3, 5, 14, 1, 17, 8
IsRowMajor	false

5.4 calculate 関数の実装

本ライブラリにおいて最も難解な関数が calculate 関数である．ソースコード 5.2 に calculate 関数，ソースコード 5.3，5.4 にそれぞれ，calculate 関数を作成する上で重要な関数である makingQOperator 関数と calcIndexNum 関数を示す．

5.4.1 calculate 関数

ソースコード 5.2: calculate 関数

```
1 qSystem* qSystem::calculate(qOperator& q_operator, int* positions, int q_operator_len){
2   int system_len = log(q_system->cols()) / log(2);
3   qOperator last_operator(q_system->cols(),q_system->cols());
4   makingQOperator(&last_operator, q_operator, positions, q_operator_len, system_len);
5
6   return calculate(last_operator);
7 }
```

calclucate関数はq_operatorとposition，そしてq_operator_lenを引数に取る．それぞれゲートとゲートをかける量子ビットの位置，ゲートをかける量子ビットの数を指定する．

2行目で定義しているsystem_lenは量子系が持つ量子ビットの数である．last_operatorは，最終的に量子系にかけるべきゲートを作成し，代入するための変数である．これをソースコード5.3に示すmakingQOperator関数に渡す．

5.4.2 makingQOperator関数

ソースコード 5.3: makingQOperator関数

```
1  qOperator* makingQOperator(qOperator* answer, qOperator& ope0, int* pos, int len0, int
       len_ans){
2    for(int i=1;i<len0;i++) pos[i] -= i;
3    for(int i=0;i<1<<(len_ans-len0);i++){
4      for(int j=0;j<1<<len0;j++){
5        int index_col = calcIndexNum(pos, len0, i, j, len_ans);
6        for(int k=0;k<1<<len0;k++){
7          if(ope0.coeff(k,j) != cmplx(0.0,0.0)){
8            int index_row = calcIndexNum(pos, len0, i, k, len_ans);
9            answer->insert(index_row, index_col) = ope0.coeff(k,j);
10         }
11       }
12     }
13   }
14   return answer;
15 }
```

makingQOperator関数の機能は，任意の数の量子ビットにかけるゲートと，そのゲートをかける量子ビットの位置，という2つの情報から，全体にかけるゲートの行列を生成することである．例えば，数式2.7のCNOTゲートと0番目と2番目の量子ビットにゲートをかける，という情報から，数式5.2のゲートを生成する．

$$CNOT_{0,2} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{5.2}$$

makingQOperator関数は，answer，ope0，pos，len0，len_ansの5つの引数を取る．これらは，出来たゲートを入れるための変数，元のゲート，ゲートをかける量子ビットの位置，ゲートをかける量子ビットの数，出来たゲートがかけられる量子ビットの数をそれぞれ示す．

2行目では，量子ビットの順番と保存されている順番が逆向きであるため，順番を反転させている．3，4，6行目で，設定すべき全ての場所を網羅するようにfor文を書いている．

5，8行目ではcalcIndexNum関数を呼び出している．この関数は，元のゲートをかける位置，元のゲートの長さ，ゲートにおける計算に係る量子ビットの位置，量子系全体の量子ビットの個数，という4つの情報から，完成する量子ゲートのどこに値を代入すればよいかを計算する機能を持つ．

これにより，9行目でanswerの導き出された位置に元のゲートの値を代入することが可能になっている．7行目は，元のゲートの要素が0だった場合に0の挿入をして疎行列クラスに無駄なメモリの確保をしないようにしている．

5.4.3 calcIndexNum関数

ソースコード 5.4: calcIndexNum関数

```
1  int calcIndexNum(int* pos, int len0, int noncalc_num, int calc_num, int len_ans){
2    int bit_len = len_ans - len0;
3    int answers[len0+1];
4    int answer = 0;
5    for(int i=0;i<len0;i++){
6      answers[i] = noncalc_num;
7      for(int j=1;j<=i;j++) answers[i] -= answers[i-j];
8      answers[i] = (answers[i] >> (bit_len-pos[i])) << (bit_len-pos[i]);
9    }
10   answers[len0] = noncalc_num;
11
12   for(int i=0;i<len0;i++)   answers[len0] -= answers[i];
13   for(int i=0;i<len0+1;i++) answer += (answers[i] << (len0-i));
14   for(int i=0;i<len0;i++)   answer += ((calc_num & (1 << (len0-i-1))) >> (len0-i-1)) << (
         bit_len-pos[i]+(len0-i-1));
15   return answer;
16 }
```

calcIndexNum関数の機能は，第5.4.2節で述べたとおりである．ゲートにおける計算に係る量子ビットの位置の情報を元に，ビット操作で1つのビットの位置を取り出すことで値を代入するべき位置を計算している．

具体的には，数式5.3，数式5.4の2つの数式と，[2，4，8]という配列から，数式5.4で示されている値を導き出す関数である．数式5.5の，上にチェックが付いている値が，数式5.3から来た値であり，他の値が数式5.4から来た値である．

$$|101\rangle \tag{5.3}$$

$$|100110\rangle \tag{5.4}$$

$$\left|10\check{1}0\check{0}110\check{1}\right\rangle \tag{5.5}$$

ここではわかりやすいように2進法で表記しているが，実際のプログラムでは数式5.2は5，数式5.4は38，数式5.5は333としてint型で保存されている．

5.5 測定に関連した関数

測定は，式 5.6，5.7 に示した行列と，その値になる確率を指定した量子ビットにかけることで行われる．

$$Zmeasurement = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \tag{5.6}$$

$$Xmeasurement = -\frac{1}{2}\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \frac{1}{2}\begin{pmatrix} 1 & -1 \\ -1 & -1 \end{pmatrix} \tag{5.7}$$

確率は getProbs 関数での機能で求められる．これは，指定された位置の量子ビットと，量子系全体の長さを第 5.4.3 節で解説した calcIndexNum 関数に渡すことで計算された index をもとに量子系の中の値から求められている．

量子系のすべての量子ビットを測定する measurement 関数では，(0,0)(1,1)(2,2)...(n,n) の値を別の量子系を一時的に用意してコピーし，戻している．これにより，1 つずつ測定する measurement 関数を呼び出すよりも高速な測定が可能である．

第6章　評価

6.1　定性評価

本研究で構築したシステムの評価は定性評価で行う．表 6.1 に表 5.2 で示したシステム要件とその評価をまとめる．

表 6.1: 本研究の実装におけるシステム要件と評価

	要件	評価	備考
要件 1	量子系クラス	◯	qSystem
要件 2	量子ビットクラス	◯	qubit
要件 3	量子ゲートクラス	◯	qOperator
要件 4	計算関数	◯	qSystem.calculate()

6.1.1　要件 1:量子系クラス

量子系クラスは qSystem という名前のクラスとして実装した．目的通り，複数の量子系の定義と，その量子系同士をテンソル積にかけることが可能になっている．これにより，量子ネットワーク，量子リピータのシミュレーションが可能である．また，密度行列を用いることにより混合状態のシミュレーションも可能になっている．

6.1.2　要件 2:量子ビットクラス

量子ビットクラスは qubit という名前のクラスとして実装した．目的通り，量子系の定義の引数として使えるクラスが実装されている．

6.1.3　要件 3:量子ゲートクラス

量子ゲートクラスは qOperator という名前のクラスとして実装した．目的通り量子系のシミュレーションをするときの計算関数の引数として使えるクラスが実装されている．また，自由にプログラマがゲートを定義できるようになっている．

6.1.4 要件 4:計算関数

計算関数は量子系クラスのメンバー関数 calculate() として実装した．目的通り，量子系のシミュレーションを行うことができる．また，プログラマが作成したゲートを任意の量子ビットにかけることが可能になっている．

6.2 既存のシステムとの速度比較

PQS を利用して Hadamard ゲートに n 個の量子ビットをかける実験を行った．図 3.1 で示した QCL での実験と速度比較する．その結果を図 6.1，図 6.2 に示す．

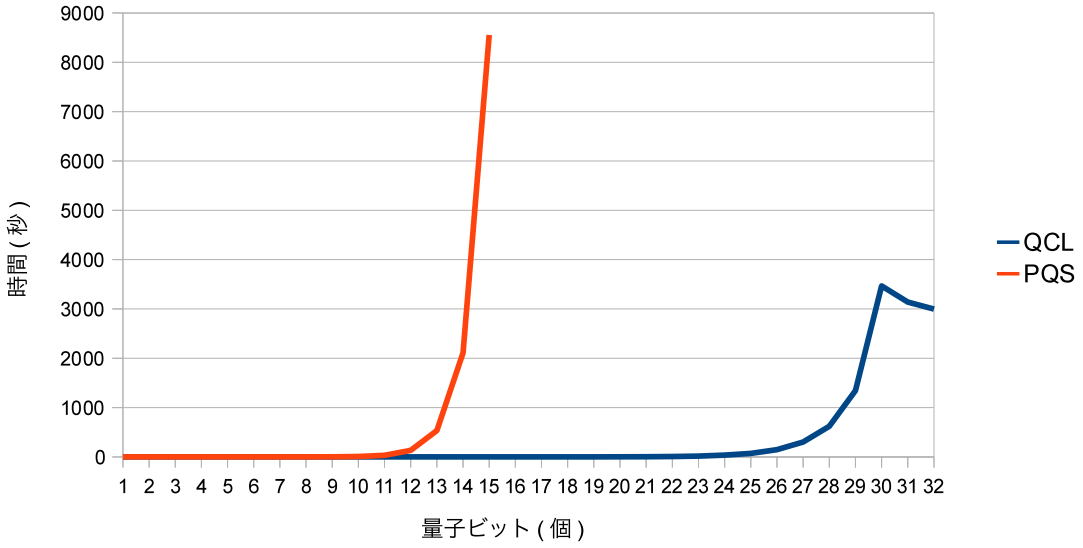

図 6.1: Hadamard ゲートのシミュレーションによる速度比較

PQS では密度行列を使っているため，計算量の増え方は n 個の量子ビットに対して 4^n に比例する．QCL では状態ベクトルを使用しているため，計算量の増え方は 2^n に比例する．このため，速度では状態ベクトルを使っている多くの量子シミュレータには PQS によるシミュレーションは速度において大きく劣る．

また，$|0\rangle$ に対してすべての量子ビットに Hadamard ゲートをかけるシミュレーションは，数式 2.11 に示す通り，全ての要素が 0 ではなくなる．このようなシミュレーションは疎行列を想定した実装のプログラムである PQS にとっては最悪のケースである．これは稀なケースであるが，このようなシミュレーションには PQS は向かない．

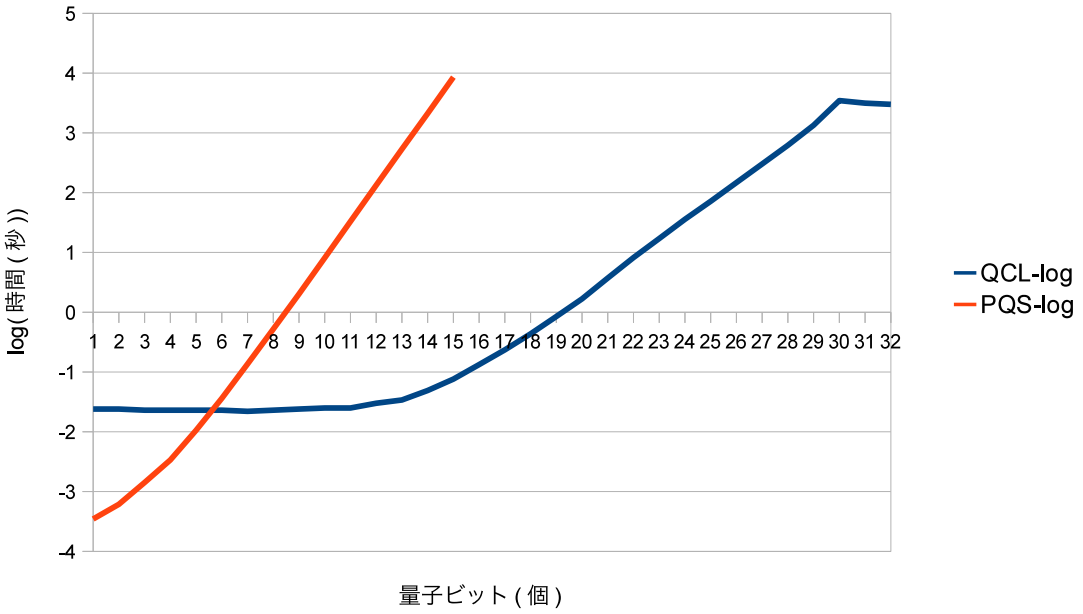

図 6.2: Hadamard ゲートのシミュレーションによる速度比較 (対数)

6.3 スケーラビリティ評価

6.3.1 疎行列を想定しない場合との比較

本節では，最良の状態の状態の時に，疎行列を前提の実装することでどの程度メモリを節約できているかを評価する．数式 6.1 のような，全ての量子ビットが $|0\rangle$ の状態である量子状態を作り，物理的なメモリをどの程度使用しているかを計測した．

$$|0\rangle = \begin{pmatrix} 1 & 0 & 0 & & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & & 0 \\ & \vdots & & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{6.1}$$

密度行列の要素は複素数なので，それぞれの要素は double 型である．double 型は 8byte なので，8×8 で 16byte である．つまり，疎行列で実装しなかった場合の論理的に必要なメモリ量は $16 \times 2^2 n$ である．

そこで，実際に使用したメモリ量と $16 \times 2^2 n$ を比較したグラフを図 6.3 に示す．

図 6.3 では，差が大きくなりすぎたため，対数表示したものを図 6.4 に示す．このように，大量のメモリを節約できていることがわかる．

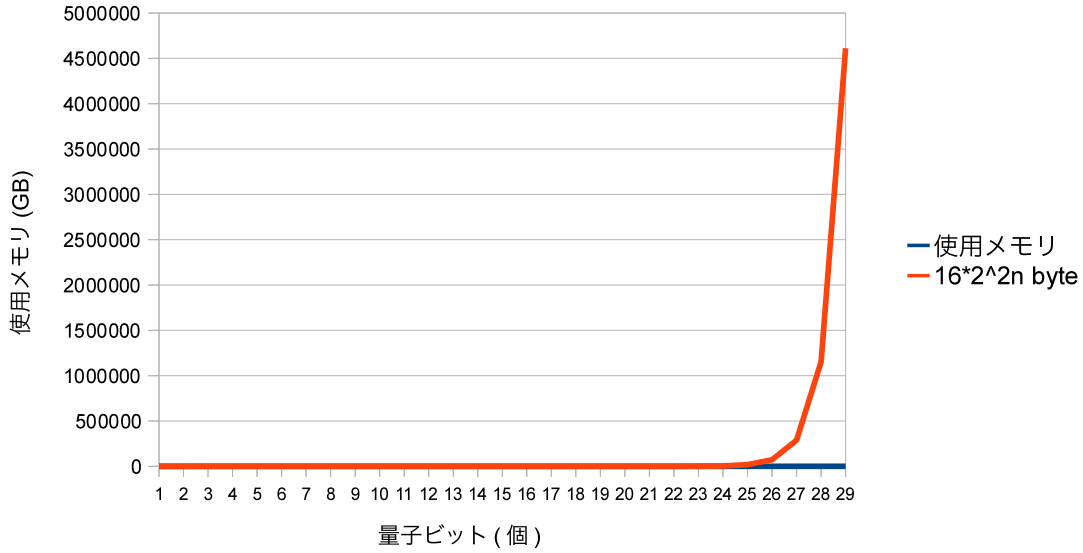

図 6.3: $|0\rangle$ の作成によるスケーラビリティ評価

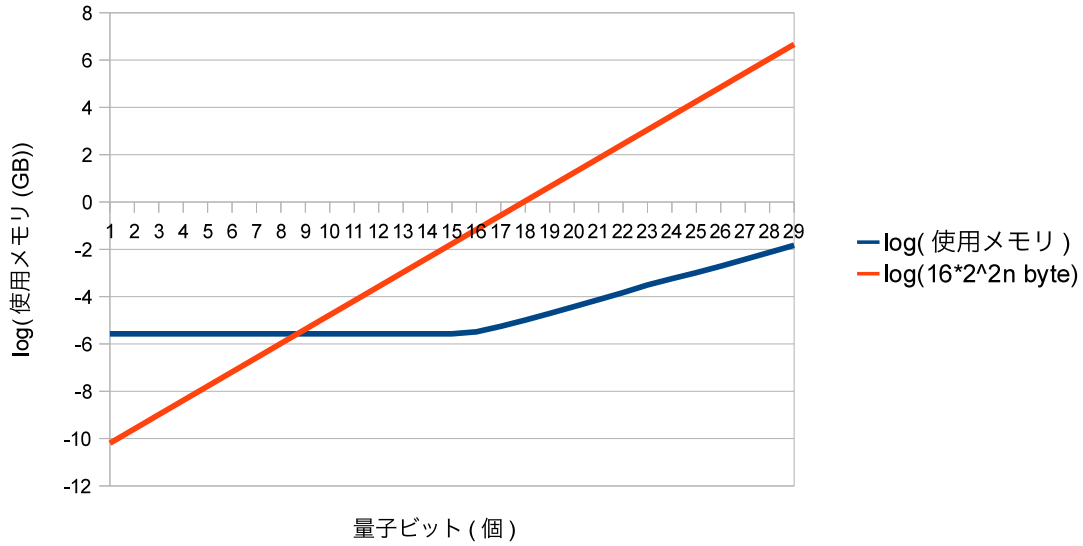

図 6.4: $|0\rangle$ の作成によるスケーラビリティ評価 (対数)

図 6.4 で量子ビットが 16 個になるまで使用メモリ量に変化がないのは，プロセス全体が使用しているメモリを計測しているためである．また，その後使用メモリ量が増えているのは，1 つの値を代入したとき，第 5.3.3 節で述べた m_outerIndex を 1 列分の確保して

しまうためである．このため，2^n に比例してメモリ使用量が増えてしまっている．

疎行列を想定した理想的なクラスを実装した場合，今回の例ではメモリ使用量は増えないべきであり，スケーラビリティは 0 でない要素数が増えない限り無限大になるのが正しい．

今回はその理想には劣る実装になった．

6.3.2 疎行列クラスのデータ圧縮の評価

量子ゲートには，量子系の持つ 0 でない要素の数を変えないものが多くある．例としては，数式 2.6，2.7 で挙げた CNOT ゲート，そして，数式 6.2，6.3，6.4 に示すパウリ行列，数式 6.5 に示すスワップゲート，数式 6.6 に示す Toffoli ゲート等，多くのものが挙げられる．

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{6.2}$$

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \tag{6.3}$$

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{6.4}$$

$$SWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{6.5}$$

$$Toffoli = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{6.6}$$

このように，全ての行，列に 1 つの 0 ではない要素がある場合，要素の数は変わらず，理想的な疎行列の実装ではメモリの使用量が変わることはない．本節では Eigen ライブラリの実装がどの程度理想的かを評価する．

評価方法として，付録 B.4 でも紹介している足し算のための回路を評価用に変更して，足し算の回路にかける前後のメモリ使用量を getrusage 関数によって測定した．前後でメモリの使用量が変わらなければ理想的な実装であると言える．

もし理想的な実装であった場合，shor のアルゴリズムでも，初めにいくつかの量子ビットに Hadamard ゲートをかけたあとの主な計算では，X ゲート，CNOT ゲート，Toffoli ゲートと，数式 6.7 で示すゲートしか使わないため，shor のアルゴリズムでもとても有用な実装である．

$$Rotate = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{\frac{-i\pi}{2^k}} \end{pmatrix} \tag{6.7}$$

ソースコード 6.1: 足し算前後のメモリ使用量の測定

```
#include <sys/time.h>
#include <sys/resource.h>
#include <iostream>
#include <vector>
#include "pqs.h"
#include "gate.h"

#define N 5

using namespace std;
using namespace pqs;

struct rusage r;

int pos_cnot[2];
int pos_toffoli[3];
qSystem psi;

void toffoli(int a, int b, int c){
  int pos[3] = {a, b, c};
  psi.calculate(Toffoli, pos, 3);
}

void toffoli(){
  psi.calculate(Toffoli, pos_toffoli, 3);
}

void cnot(int a, int b){
  int pos[2] = {a, b};
  psi.calculate(CNOT, pos, 2);
}

void cnot(){
  psi.calculate(CNOT, pos_cnot, 2);
}

void h(int a){
  int pos[1] = {a};
  psi.calculate(H, pos, 1);
}

int main(){
  makeGate();

  qubit value[3*N];
  for(int i=0;i<3*N;i++) value[i] = qubit(sqrt(cmplx(1.0,0.0)), sqrt(cmplx(0.0,0.0)));
  psi = qSystem(3*N,value);

  for(int i=0;i<N;i++){
    h(i*3);
  }
```

```
52
53    getrusage(RUSAGE_SELF, &r);
54    cout << r.ru_maxrss << endl;
55
56    for(int i=0;i<N;i++){
57      for(int j=0;j<3;j++) pos_toffoli[j] = i * 3 + j;
58      toffoli();
59    }
60    for(int i=1;i<N;i++){
61      for(int j=0;j<2;j++) pos_cnot[j] = i * 3 + j;
62      cnot();
63    }
64    for(int i=1;i<N;i++){
65      toffoli(-1+i*3, 1+i*3, 2+i*3);
66    }
67    cnot(N*3-4,N*3-2);
68    for(int i=N-2;i>0;i--){
69      toffoli(-1+i*3, 1+i*3, 2+i*3);
70      cnot(i*3, 1+i*3);
71      toffoli(i*3, 1+i*3, 2+i*3);
72      cnot(-1+i*3, 1+i*3);
73      cnot(i*3, 1+i*3);
74    }
75    toffoli(0,1,2);
76    cnot(0,1);
77
78    getrusage(RUSAGE_SELF, &r);
79    cout << r.ru_maxrss << endl;
80  }
```

ソースコード 6.1 は，実際に使用したコードである．define されている N の値を変更することで計測を行った．この結果を表に示す．

表 6.2: 足し算前後のメモリ使用量の測定

N	量子ビット (個)	計算前使用メモリ (kB)	計算後使用メモリ (kB)	時間 (秒)
4	12	2780	2780	1.3
5	15	6500	6500	13.5
6	18	44120	44120	133.31
7	21	336836	336836	1281.88

量子ビットの個数は N に対して $N\times3$ になっている．詳しくは付録 B.4 を参照されたい．さて，評価目的の計算前と計算後の使用メモリは，1kB の違いもなく同じであることがわかった．これは，データが移動した時に移動元が 0 になったあと確保したメモリを削除していることの証明であり，その点において Eigen ライブラリでは理想的な実装であると言える．

本説の目的とはずれるが，メモリの使用量が量子ビットの個数に対して 2^n ずつ，この場合では，下にさがるたびに 8 倍になっていることが確認できる．時間については 10 倍ずつ増えている．これは Hadamard ゲートの計算をはじめに行っていることも関わっていると思われるが，さらなる考察が必要である．

6.4 まとめ

本章ではシステム要件と照らし合わせて，本研究で構築したシステムの定性評価，QCL との速度比較，そしてスケーラビリティの評価を行った．

定性評価により，本研究が目指したライブラリのシステム要件を満たしていることを確認した．

速度評価では，密度行列を使用しているため，0 でない要素が多い場合は QCL に比べて大きく劣ることが分かった．ただ，これは密度行列を使用する以上はやむを得ないことである．

スケーラビリティ評価では 0 の要素が多いような量子状態のシミュレーションでは疎行列により，メモリを多く節約できていることが分かった．また，疎行列の圧縮については理想的な実装を持っていることが分かった．しかし，データ構造については理想的な実装とは言えず，改善点があることも分かった．

第7章　結論

7.1　まとめ

量子状態のシミュレーションのためのライブラリでは，プログラマがいかに容易に，いかに自由に実装できるかが重要である．

本研究では，状態ベクトルで表した簡単な量子ビットを引数に取ることで，容易に疎行列を利用した量子系を定義することができる．量子系同士はテンソル積にかけることが出来，複数の量子系を使った量子ネットワーク，量子リピータのシミュレーションも容易である．

また，量子ゲートを Eigen ライブラリの SparseMatrix クラスで自由に定義でき，そのまま量子系のクラスに引数として渡すだけで量子ビットのシミュレーションが可能である．

本研究の成果である PQS ライブラリはオープンソースで公開されている．[7] 詳しい使い方は第 4 章を，インストール方法は付録 A を参照されたい．

7.2　今後の展望

7.2.1　新しい量子プログラミング言語の開発

本研究は，多くの量子シミュレータを作成するプログラマのためにある．本ライブラリを使用して，多くの量子シミュレータが容易に実装され，それにより量子プログラミング言語の開発の研究が進むことが本研究の最終的な目標である．

量子プログラミング言語の開発は，実際に量子コンピュータが完成した後のための最も重要な研究の１つである．

7.2.2　疎行列クラスのデフォルト値の自動変更

第 2.3 節で述べたように，疎行列クラスはエラーのシミュレーションでは，デメリットが大きくなってしまう．0 であるべき値の代わりに小さな同じ値が入ってしまうからである．このような状況は，量子ビットシミュレーションではエラーのシミュレーション以外でも起こりやすい．

このように，0 の値がある程度減ってしまった時に，行列の中で最も多い値を検索し，その値を 0 の代わりのデフォルト値として自動的に設定することで効率的なシミュレーションが可能である．

このような機能の実装は，量子ビットシミュレーションを更に効率的にする．

7.3 今後の課題

7.3.1 状態ベクトルの実装

本研究では，量子状態を密度行列で表現している．密度行列は状態ベクトルに対して，混合状態を表現できる点で優れているが，速度やメモリ使用量の点において劣っている．

純粋状態のみをシミュレーションしたいプログラマのために，状態ベクトルと密度行列のどちらを使うかを選択できるようにすることは必須であると考えている．

7.3.2 データ構造の変更

Eigen ライブラリの疎行列クラスでは，index を使ったデータ構造を利用している．これは，1 つの行に連続したデータが入る行列では効率が良いが，量子シミュレーションにおける多くの場合は，この状況には当てはまらない．

また，このデータ構造のために，大きな行列を作成した場合，1 つの値を代入しただけで膨大なメモリを無駄にしてしまう．

本研究では，今後独自の木構造による疎行列クラスを作成する予定である．行列用とベクトル用の 2 種類を作成し，それぞれ 4 本木構造，2 本木構造にする．これにより，理想的にメモリを消費しない，検索の高速な疎行列クラスが完成すると考えている．

量子シミュレータにおいて多くの場合，はじめに (0,0) に値を代入するため，図状況に応じて上方向にも枝を伸ばすことの出来る双方向 4 本木構造になる予定である．

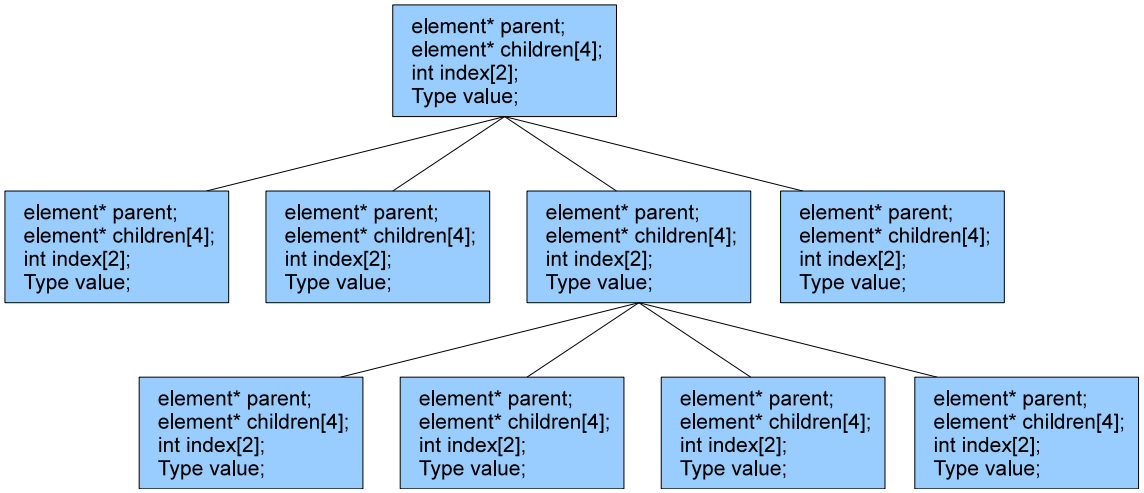

図 7.1: 双方向 4 本木構造

index が 2 つの値を持つのは，2×2 の行列であるためであり，下記 4 つの値のパターンによって枝の場所は決定される．

- $x_0 \leq x_1 \wedge y_0 \leq y_1$
- $x_0 \leq x_1 \wedge y_0 > y_1$
- $x_0 > x_1 \wedge y_0 \leq y_1$
- $x_0 > x_1 \wedge y_0 > y_1$

この構造は，1 つの要素が必要とするメモリ量は多いが，1 つの要素を小さくしようとして結果として必要のない要素のためにメモリを確保したり，計算量の効率が悪いデータ構造にするよりは結果的に優れていると考えられる．

理論的な行列の大きさは大きく，要素の少ない疎行列であればあるほどこのデータ構造は優秀になる．

7.3.3 速度の改善

本実装で Eigen ライブラリを採用した理由は，並列化が可能ないくつかの行列計算のライブラリの中で一番高速だったからである．しかし，実際に実装してみると，並列化は OpenMP をほとんど使わずに自前で実装してあり，かつ少なくとも疎行列のライブラリでは並列化されていなかった．

事前の調査では並列化せずに速度を測定し，また，あまり大きい問題にはしていなかったためである．

Eigen ライブラリのコードを変更することで行列計算を並列化しようとしたがメモリリークが発生してしまい成功しなかった．

今後は新しいデータ構造の実装とともに並列化することで速度を改善する予定である．

7.3.4 出力方法の改善

現在，Eigen ライブラリの¡¡オペレーターを使って出力しているため，大きい行列の出力をすると，可読性が著しく下がる．純粋状態である場合は，0 の部分を排除した上で 10 進数と共に出力する，といったような対処が必要である．

謝辞

本論文の執筆にあたり，多くの方々に本当にお世話になりました．まず，ご指導いただいた慶應義塾大学環境情報学部長村井純博士，同学部教授徳田英幸博士，同学部准教授楠本博之博士，同学部教授中村修博士，同学部准教授高汐一紀博士，同学部准教授 Rodney D. Van Meter III 博士，同学部准教授植原啓介博士，同学部准教授三次仁博士，同学部准教授中澤仁博士，同学部教授武田圭史，博士同大学大学院政策・メディア研究科特認講師斉藤賢爾博士に感謝致します．この場をお借りしてお礼を申し上げます．

Rodney D. Van Meter III 博士に重ねて感謝致します．博士には私が研究室に入室した当初からご指導ご助言して頂き，最後まで温かく見守ってくださいました．

Oxford 大学リサーチアシスタント Clare Horsman 博士に感謝致します．量子プログラミング言語の研究のきっかけを作ってくださいました．

本論文の執筆にあたって指導を頂いた楠本博之博士に重ねて感謝致します．叱咤激励していただき，多くの協力を頂きました．

本研究に際して指導を頂いた慶應義塾大学政策メディア研究科 特任講師 鈴木茂哉博士に感謝致します．豊富な知識でサポートしていただきました．

本研究に際して英語の指導にあたって頂いた立命館大学生命科学部生物工学科山中司准教授に感謝致します．私の拙い英語を指導していただきました．

慶應義塾大学政策メディア研究科博士課程永山翔太氏に重ねて深謝致します．数々のレトルト食品で私の空腹を救ってくださいました．

研究室で活動を共にしました，慶應義塾大学政策メディア研究科修士課程石崎佳織女史，同大学学士課程福山翔一朗氏，Nguyen Trung Duc 氏，水谷伊織氏，細川毅騎氏，松尾賢明氏，Maxim LeFleur 氏，Michael Wiegner 氏，Nhop-anon Thairungroj 氏に感謝致します．

そして，徳田・村井・楠本・中村・高汐・重近・バンミーター・植原・三次・中澤合同研究プロジェクトの皆様に感謝致します。

サークル活動で寝食を共にした，黒須隆宏氏，渡邉光洋氏，浅田奈穂子女史，海上知哉氏，庄司すみれ女史，安形憲一氏，渋谷航星氏，内海翔太氏，浜田瑞希女史に感謝致します．大学生活を支えてくれました．

最後に，22 年間育てていただいた，父 泰朗，加えて，要旨の翻訳に協力もしていただいた，母 祐子，そして，彼女のおかげで今の私があると言っても良い，姉 玲に感謝致します.

以上を持って謝辞といたしいます.

2014 年 1 月
村田 紘司

参考文献

[1] Moore, Gordon E. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[2] Schaller, R.R. Moore's law: past, present and future. *IEEE*, 34(6), 1997.

[3] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. *IEEE*, 1994.

[4] S. Takeda, T. Mizuta, M. Fuwa, P. van Loock and A. Furusawa. Deterministic quantum teleportation of photonic quantum bits by a hybrid technique. *Nature*, 500(315-318), 2013.

[5] Ömer, Bernhard. Classical Concepts in Quantum Programming. *International Journal of Theoretical Physics*, 44-7(943-955), 2005.

[6] http://eigen.tuxfamily.org/

[7] https://github.com/malt03/pqs

[8] Vlatko Vedral, Adriano Barenco and Artur Ekert. Quantum networks for elementary arithmetic operations. *Phys. Rev. A*, 54(147-153), 1996.

付 録A　PQSのインストール方法

A.1　Eigen ライブラリ

Eigen ライブラリをインストールしていない場合は，まず下記 URL を参照してインストールする．本ライブラリは Eigen3.1.4 を利用している．http://eigen.tuxfamily.org/

cmake を使ってインストールすることを推奨するが，解凍してそのまま include してもよい．その場合は pqs.h を編集してインクルードファイルを変更するか，リンクする必要がある．

A.2　PQS のインストール

任意のディレクトリで

ソースコード A.1: PQS のインストール

```
% git clone git@github.com:malt03/pqs.git
% cd pqs
% ./configure
% make
% sudo make install
```

とすることでインストールすることができる．

付 録B　PQSによる実装例

B.1　Hadamardゲート

ソースコード B.1: Hadamard ゲート-実装

```
#include <iostream>
#include <vector>
#include "pqs.h"

using namespace std;
using namespace pqs;

int main(){
  // first, create the matrix for the Hadamard operator
  double r2 = 1.0/sqrt(2.0);
  vector<T> tripletList;
  tripletList.push_back(T(0,0,cmplx(r2,0.0)));
  tripletList.push_back(T(1,0,cmplx(r2,0.0)));
  tripletList.push_back(T(0,1,cmplx(r2,0.0)));
  tripletList.push_back(T(1,1,cmplx(-r2,0.0)));
  qOperator H(2,2);
  H.setFromTriplets(tripletList.begin(), tripletList.end());

  // create a system with three qubits
  qubit value[] = {
    qubit(sqrt(cmplx(1.0,0.0)), sqrt(cmplx(0.0,0.0))),
    qubit(sqrt(cmplx(1.0,0.0)), sqrt(cmplx(0.0,0.0))),
    qubit(sqrt(cmplx(1.0,0.0)), sqrt(cmplx(0.0,0.0))),
  };
  qSystem psi(3,value);

  // pos[] defines which a qubit we will execute the H on
  int pos[] = {1};

  cout << "before:\n" << psi << endl;
  psi.calculate(H, pos, 1);
  cout << "after:\n" << psi << endl;
}
```

ソースコード B.2: Hadamard ゲート-出力

```
before:
Nonzero entries:
((1,0),0)

Outer pointers:
0 1 1 1 1 1 1 1  $

(1,0) 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

after:
Nonzero entries:
((0.5,0),0) ((0.5,0),2) ((0.5,0),0) ((0.5,0),2)

Outer pointers:
0 2 2 4 4 4 4 4  $

(0.5,0) 0 (0.5,0) 0 0 0 0 0
0 0 0 0 0 0 0 0
(0.5,0) 0 (0.5,0) 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

B.2 CNOTゲート

ソースコード B.3: CNOT ゲート-実装

```
#include <iostream>
#include <vector>
#include "pqs.h"

using namespace std;
using namespace pqs;

int main(){
  // first, create the matrix for the CNOT operator
  vector<T> tripletList;
  tripletList.push_back(T(0,0,cmplx(1.0,0.0)));
  tripletList.push_back(T(1,1,cmplx(1.0,0.0)));
  tripletList.push_back(T(3,2,cmplx(1.0,0.0)));
  tripletList.push_back(T(2,3,cmplx(1.0,0.0)));
  qOperator CNOT(4,4);
  CNOT.setFromTriplets(tripletList.begin(), tripletList.end());

  // create a system with three qubits
  qubit value[] = {
    qubit(sqrt(cmplx(0.5,0.0)), sqrt(cmplx(0.5,0.0))),
    qubit(sqrt(cmplx(1.0,0.0)), sqrt(cmplx(0.0,0.0))),
    qubit(sqrt(cmplx(0.7,0.0)), sqrt(cmplx(0.3,0.0))),
  };
  qSystem psi(3,value);

  // pos[] defines which two qubits we will execute the CNOT on
  int pos[] = {0,2};

  cout << "before:\n" << psi << endl;
  psi.calculate(CNOT, pos, 2);
  cout << "after:\n" << psi << endl;
}
```

ソースコード B.4: CNOT ゲート-出力

```
before:
Nonzero entries:
((0.35,0),0) ((0.229129,0),1) ((0.35,0),4) ((0.229129,0),5) ((0.229129,0),0) ((0.15,0),1)
     ((0.229129,0),4) ((0.15,0),5) ((0.35,0),0) ((0.229129,0),1) ((0.35,0),4)
    ((0.229129,0),5) ((0.229129,0),0) ((0.15,0),1) ((0.229129,0),4) ((0.15,0),5)
```

```
4
5  Outer pointers:
6  0 4 8 8 8 12 16 16  $
7
8  (0.35,0) (0.229129,0) 0 0 (0.35,0) (0.229129,0) 0 0
9  (0.229129,0) (0.15,0) 0 0 (0.229129,0) (0.15,0) 0 0
10 0 0 0 0 0 0 0 0
11 0 0 0 0 0 0 0 0
12 (0.35,0) (0.229129,0) 0 0 (0.35,0) (0.229129,0) 0 0
13 (0.229129,0) (0.15,0) 0 0 (0.229129,0) (0.15,0) 0 0
14 0 0 0 0 0 0 0 0
15 0 0 0 0 0 0 0 0
16
17 after:
18 Nonzero entries:
19 ((0.35,0),0) ((0.229129,0),1) ((0.229129,0),4) ((0.35,0),5) ((0.229129,0),0) ((0.15,0),1)
       ((0.15,0),4) ((0.229129,0),5) ((0.229129,0),0) ((0.15,0),1) ((0.15,0),4)
     ((0.229129,0),5) ((0.35,0),0) ((0.229129,0),1) ((0.229129,0),4) ((0.35,0),5)
20
21 Outer pointers:
22 0 4 8 8 8 12 16 16  $
23
24 (0.35,0) (0.229129,0) 0 0 (0.229129,0) (0.35,0) 0 0
25 (0.229129,0) (0.15,0) 0 0 (0.15,0) (0.229129,0) 0 0
26 0 0 0 0 0 0 0 0
27 0 0 0 0 0 0 0 0
28 (0.229129,0) (0.15,0) 0 0 (0.15,0) (0.229129,0) 0 0
29 (0.35,0) (0.229129,0) 0 0 (0.229129,0) (0.35,0) 0 0
30 0 0 0 0 0 0 0 0
31 0 0 0 0 0 0 0 0
```

B.3 Toffoliゲート

ソースコード B.5: Toffoli ゲート-実装

```
1  #include <iostream>
2  #include <vector>
3  #include "pqs.h"
4
5  using namespace std;
6  using namespace pqs;
7
8  int main(){
9    // first, create the matrix for the Toffoli operator
10   vector<T> tripletList;
11   tripletList.push_back(T(0,0,cmplx(1.0,0.0)));
12   tripletList.push_back(T(1,1,cmplx(1.0,0.0)));
13   tripletList.push_back(T(2,2,cmplx(1.0,0.0)));
14   tripletList.push_back(T(3,3,cmplx(1.0,0.0)));
15   tripletList.push_back(T(4,4,cmplx(1.0,0.0)));
16   tripletList.push_back(T(5,5,cmplx(1.0,0.0)));
17   tripletList.push_back(T(6,7,cmplx(1.0,0.0)));
18   tripletList.push_back(T(7,6,cmplx(1.0,0.0)));
19   qOperator Toffoli(8,8);
20   Toffoli.setFromTriplets(tripletList.begin(), tripletList.end());
21
22   // create a system with three qubits
23   qubit value[] = {
24     qubit(sqrt(cmplx(0.5,0.0)), sqrt(cmplx(0.5,0.0))),
25     qubit(sqrt(cmplx(0.0,0.0)), sqrt(cmplx(1.0,0.0))),
26     qubit(sqrt(cmplx(0.3,0.0)), sqrt(cmplx(0.7,0.0))),
27   };
28   qSystem psi(3,value);
29
```

```
30    // pos[] defines which three qubits we will execute the Toffoli on
31    int pos[] = {0,1,2};
32
33    cout << "before:\n" << psi << endl;
34    psi.calculate(Toffoli, pos, 3);
35    cout << "after:\n" << psi << endl;
36  }
```

ソースコード B.6: Toffoli ゲート-出力

```
1   before:
2   Nonzero entries:
3   ((0.15,0),2) ((0.229129,0),3) ((0.15,0),6) ((0.229129,0),7) ((0.229129,0),2) ((0.35,0),3)
        ((0.229129,0),6) ((0.35,0),7) ((0.15,0),2) ((0.229129,0),3) ((0.15,0),6)
      ((0.229129,0),7) ((0.229129,0),2) ((0.35,0),3) ((0.229129,0),6) ((0.35,0),7)
4
5   Outer pointers:
6   0 0 0 4 8 8 8 12  $
7
8   0 0 0 0 0 0 0 0
9   0 0 0 0 0 0 0 0
10  0 0 (0.15,0) (0.229129,0) 0 0 (0.15,0) (0.229129,0)
11  0 0 (0.229129,0) (0.35,0) 0 0 (0.229129,0) (0.35,0)
12  0 0 0 0 0 0 0 0
13  0 0 0 0 0 0 0 0
14  0 0 (0.15,0) (0.229129,0) 0 0 (0.15,0) (0.229129,0)
15  0 0 (0.229129,0) (0.35,0) 0 0 (0.229129,0) (0.35,0)
16
17  after:
18  Nonzero entries:
19  ((0.15,0),2) ((0.229129,0),3) ((0.229129,0),6) ((0.15,0),7) ((0.229129,0),2) ((0.35,0),3)
        ((0.35,0),6) ((0.229129,0),7) ((0.229129,0),2) ((0.35,0),3) ((0.35,0),6)
      ((0.229129,0),7) ((0.15,0),2) ((0.229129,0),3) ((0.229129,0),6) ((0.15,0),7)
20
21  Outer pointers:
22  0 0 0 4 8 8 8 12  $
23
24  0 0 0 0 0 0 0 0
25  0 0 0 0 0 0 0 0
26  0 0 (0.15,0) (0.229129,0) 0 0 (0.229129,0) (0.15,0)
27  0 0 (0.229129,0) (0.35,0) 0 0 (0.35,0) (0.229129,0)
28  0 0 0 0 0 0 0 0
29  0 0 0 0 0 0 0 0
30  0 0 (0.229129,0) (0.35,0) 0 0 (0.35,0) (0.229129,0)
31  0 0 (0.15,0) (0.229129,0) 0 0 (0.229129,0) (0.15,0)
```

B.4 足し算のための回路

図 B.1 は，足し算のための回路である．[8] 数式 B.1 の状態の量子系を入力すると，数式 B.2 の状態の量子系が出力される．

$$|A, B, 0\rangle \tag{B.1}$$

$$|A, A + B, 0\rangle \tag{B.2}$$

A_n，B_n はそれぞれ A，B を 2 進法で表した時の値であり，C_n は temporary 変数である．一番下の K は繰り上がりに使われる．

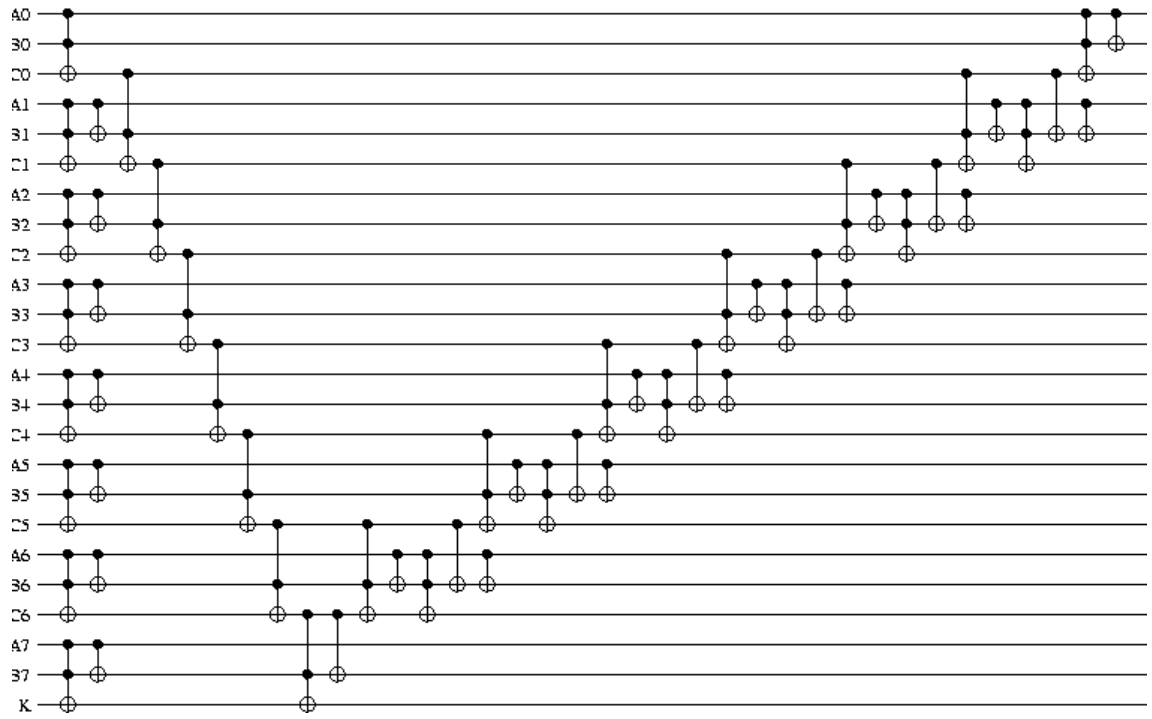

図 B.1: 足し算のための回路

このままだとシミュレーションするには大きいので，3×3の9量子ビットを使って4bit+4bitの足し算のシミュレーションを行う．第B.2，B.3節で定義したCNOTゲートとToffoliゲートはgate.hに定義されているmakeGate関数を呼び出すことで利用できるようになるものとする．

今回の例では，5+6を行う．これを量子ビットに落としこむと数式B.3になる．

$$|100010110\rangle \tag{B.3}$$

実際のプログラムをソースコードB.7に示す．また，その出力の一番上の一部をソースコードB.8に示す．

ソースコード B.7: 足し算のための回路-実装

```
#include <iostream>
#include <vector>
#include "pqs.h"
#include "gate.h"

using namespace std;
using namespace pqs;

int pos_cnot[2];
int pos_toffoli[3];
qSystem psi;

void toffoli(int a, int b, int c){
  int pos[3] = {a, b, c};
  psi.calculate(Toffoli, pos, 3);
}
```

```
17
18 void toffoli(){
19   psi.calculate(Toffoli, pos_toffoli, 3);
20 }
21
22 void cnot(int a, int b){
23   int pos[2] = {a, b};
24   psi.calculate(CNOT, pos, 2);
25 }
26
27 void cnot(){
28   psi.calculate(CNOT, pos_cnot, 2);
29 }
30
31 int main(){
32   makeGate();
33
34   qubit value[] = {
35     qubit(sqrt(cmplx(0.0,0.0)), sqrt(cmplx(1.0,0.0))), //A0
36     qubit(sqrt(cmplx(1.0,0.0)), sqrt(cmplx(0.0,0.0))), //B0
37     qubit(sqrt(cmplx(1.0,0.0)), sqrt(cmplx(0.0,0.0))), //C0
38     qubit(sqrt(cmplx(1.0,0.0)), sqrt(cmplx(0.0,0.0))), //A1
39     qubit(sqrt(cmplx(0.0,0.0)), sqrt(cmplx(1.0,0.0))), //B1
40     qubit(sqrt(cmplx(1.0,0.0)), sqrt(cmplx(0.0,0.0))), //C1
41     qubit(sqrt(cmplx(0.0,0.0)), sqrt(cmplx(1.0,0.0))), //A2
42     qubit(sqrt(cmplx(0.0,0.0)), sqrt(cmplx(1.0,0.0))), //B2
43     qubit(sqrt(cmplx(1.0,0.0)), sqrt(cmplx(0.0,0.0))), //K
44   };
45   psi = qSystem(9,value);
46
47   for(int i=0;i<3;i++){
48     for(int j=0;j<3;j++) pos_toffoli[j] = i * 3 + j;
49     toffoli();
50   }
51   for(int i=1;i<=2;i++){
52     for(int j=0;j<2;j++) pos_cnot[j] = i * 3 + j;
53     cnot();
54   }
55   toffoli(2,4,5);
56   toffoli(5,7,8);
57   cnot(5,7);
58   toffoli(2,4,5);
59   cnot(3,4);
60   toffoli(3,4,5);
61   cnot(2,4);
62   toffoli(0,1,2);
63   cnot(3,4);
64   cnot(0,1);
65
66   cout << psi << endl;
67 }
```

ソースコード B.8: 足し算のための回路-出力

```
1 Nonzero entries:
2 ((1,0),405)
```

405 は 2 進法に直すと数式 B.4 になる．これの A+B に当たるのはチェックが付いている部分である．これを取り出すと数式 B.5 であり，量子ビットの列は後ろから読むので 2 進法で 1011，つまり 10 進法での 11 となる．

$$\left|1\check{1}00\check{1}01\check{0}\check{1}\right\rangle \tag{B.4}$$

$$|1101\rangle \tag{B.5}$$

$|A\rangle = \sum_{j=0}^{2^n} |j\rangle$