

量子プログラミング入門

－前提知識と JavaScript シミュレータによる実践－

Masayuki Otoshi
courseml@gmail.com

Revision History

| Version | Name | Date | Description |
|---------|-----------------|----------|------------------|
| 1.00 | Masayuki Otoshi | 8/1/2015 | Document Created |
| | | | |
| | | | |
| | | | |
| | | | |

Table of Contents

| | |
|--------------------------|----|
| 1. はじめに | 3 |
| 2. 量子計算 | 4 |
| 量子ビット(qubit) | 4 |
| 重ね合わせ状態 | 5 |
| 二次元複素ベクトル表現 | 8 |
| 確率振幅 | 9 |
| n-qubit の計算 | 11 |
| 観測する | 16 |
| エンタングルメント状態（量子もつれ） | 18 |
| 2. 量子論理ゲート | 20 |
| パウリ X ゲート（NOT ゲート） | 20 |
| アダマール(Hadamard)ゲート | 21 |

| | |
|--|----|
| 3. Function..... | 22 |
| U_f ゲート..... | 22 |
| 超並列実行..... | 24 |
| 4. JavaScript シミュレータによる量子プログラミング | 26 |
| クラス図..... | 26 |
| NOT 演算 | 29 |
| テンソル積..... | 31 |
| qubit の観測 | 33 |
| エンタングルメント状態の観測 | 36 |
| パウリ X ゲート..... | 38 |
| アダマールゲート | 39 |
| Function の実行 | 41 |
| 5. Deutsch's algorithm..... | 44 |
| アルゴリズムの概要..... | 44 |
| JavaScript での実装..... | 45 |
| 量子コンピュータでの実装 | 47 |
| 6. おわりに | 52 |

1. はじめに

プログラミングを初めて学んだ頃を思い出して欲しい。おそらく、2進数での数え方やAND、ORなどのビット演算を学習したはずである。特に1+1が1になるというOR演算は、それまでの数学で学んだ常識を覆すインパクトがあり、不思議な印象を受けたことと思う。このように、我々は何か新しいことを学ぶ時、その世界の常識を理解することから始めなければならない。そして、まさに今、新たな常識がプログラミングの世界に押し寄せようとしている。それが量子コンピュータ上でのプログラミングである。

現在のコンピュータ（以降、古典コンピュータと呼ぶ）では、電圧の高低をビットの1,0に割り当てて表現したが、量子コンピュータでは、電子の回転（スピン）方向や光子(Photon)の偏向方向といった量子の状態に対して、ビットを割り当てていく。量子のように極小の世界になると、我々の住む世界の常識が通用しない。例えば、ある一つの電子の回転方向が、ある一瞬において上向きであり、かつ下向きである、という複数の状態が同時に存在可能なのである。これは、あなたが空を見上げつつ地面を見るという、相反する行為を同時に行うことが可能だと言っていることに等しい。

そうした奇妙な量子の現象が量子コンピュータの動作原理となっているため、古典コンピュータでのプログラミングに慣れたプログラマーにとっては、理解し難いかもしれない。また、使われている用語が量子力学から拝借したものが多い点も更にハードルを高めている。

現時点では、量子コンピュータはまだ基礎研究の段階であり、実用的プログラムを書くというレベルではない。しかしながら、量子コンピューティングの基本原理解は既に数理モデルにより明確に定義されており、それに基づいて開発されたシミュレータを用いることで量子プログラミングを体験することが可能である。ただ、量子プログラミングを始めるには、ある程度の量子力学および数学の知識が必要となる。それは、我々が古典コンピュータに対して、2進数や論理演算を学ぶことから始めたようなものだとして欲しい。

とは言え、現時点での量子プログラミングは非常に原始的な操作の羅列に過ぎず、それは、ちょうどアセンブラによるプログラミングを行っているレベルだと思って差し支えない。

今後、より高度で複雑な実用的プログラミングへと発展させるために、我々には量子プログラミングにおける高級言語やフレームワークを考案していく責務があるものと思う。

そこで、本稿では古典コンピュータでのプログラミング経験はあるが、量子コンピュータや量子力学については初めて、という方を対象にしている。量子コンピュータ上でのプログラミングに必要な基礎知識だけでなく、JavaScriptで書かれたシミュレータを用いて具体的なコードとその実行結果も紹介しながら説明していく。

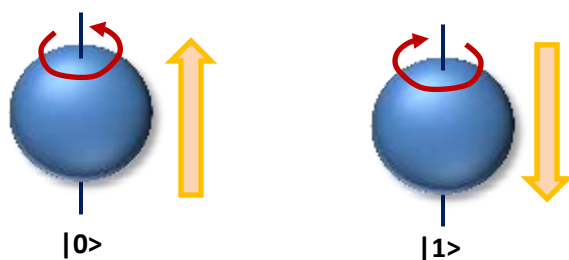
2. 量子計算

量子ビット(qubit)

古典コンピュータでの 1 ビットに相当する単位を、量子コンピュータでは **qubit** という（または、**Qbit**、量子ビットとも言う）。1 ビットは 0 または 1 の値を持つが、1 qubit はそれらに対応する状態 (state) を $|0\rangle$ または $|1\rangle$ で表現する。

余談だが、 $|0\rangle$ は「0 ケット」と読み、qubit の状態が $|0\rangle$ であることを示す。同様に、 $|1\rangle$ は「1 ケット」と読み、状態が $|1\rangle$ であることを表している。このケットという呼び名は **bracket** が由来で、 $|>$ は **bracket** の右半分なので「ket」と呼ばれている。ここでは出てこないが、 $\langle 0|$ という状態もあり、こちらは **bra** (ブラ) と呼ぶ。

ところで、古典コンピュータでの 0 と 1 は電圧の低い（基準値）と高い状態に対応したものだが、qubit の「状態」とは具体的に何なのであろうか？それは、量子コンピュータを構築する実体に依存する。例えば、電子のスピンで表現した場合、電子が右回りしている状態を基準にしたとすると、上向きに回転している状態を $|0\rangle$ 、下向きに回転している状態を $|1\rangle$ と定義できる。また、光子(photon)を用いた場合は、光子を偏向させた向きで表現する。例えば、x 軸方向を向いた状態を $|0\rangle$ 、y 軸方向を向いた状態を $|1\rangle$ と定義できる。



重ね合わせ状態

qubit は $|0\rangle$ 、 $|1\rangle$ と決まった値をとる状態（独立の状態）だけでなく、それらが重ね合わさった状態(superposition state)もとることができる。それら全ての状態を $|\phi\rangle$ で表し、以下の式で定義される。

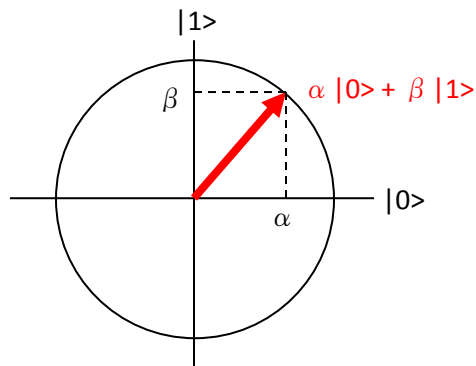
$$|\phi\rangle = \alpha |0\rangle + \beta |1\rangle \quad \text{式(1)}$$

ただし、 α 、 β は**複素数**（実数と虚数で構成され、 $a+bi$ で表される数）であり、以下の条件（**規格化条件**）を満たすものとする。

$$|\alpha|^2 + |\beta|^2 = 1 \quad \text{式(2)}$$

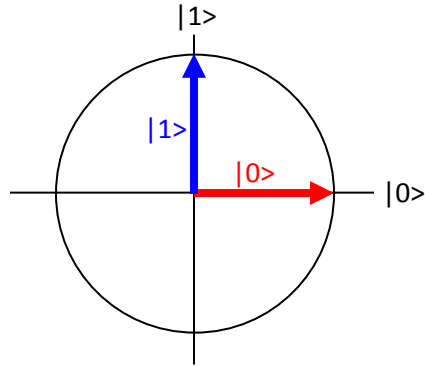
具体的には、 α は $|0\rangle$ の確率振幅(probability amplitude)、 β は $|1\rangle$ の確率振幅を表しており、この qubit を観測すると $|\alpha|^2$ の確率で $|0\rangle$ になり、同様に $|\beta|^2$ の確率で $|1\rangle$ になることを示している。しかし、ひとたび観測すると 0 または 1 の値に収束する。

これらの式が意味する内容を理解することは、非常に重要である。そこで、qubit を平面上の単位ベクトルとして図示してみよう。横軸を $|0\rangle$ 、縦軸を $|1\rangle$ とすると以下のように描かれる。



つまり、qubit の「状態」とは大きさ 1 のベクトルであり、「演算」とはこの単位円の円周上を動き回ること、と言える。

さて、重ね合わせのない独立した状態である $|0\rangle$ や $|1\rangle$ もこれらの式で表現できる。例えば $|0\rangle$ は $\alpha=1, \beta=0$ の状態と言え、同様に、 $|1\rangle$ は $\alpha=0, \beta=1$ の場合を示している。図示すると以下ようになる。



これに対し、重ね合わさった状態はベクトルの向きが水平や垂直でなく、傾いた状態と言える。つまり、式(2)の規格化条件「 $|\alpha|^2 + |\beta|^2 = 1$ 」を満たす α と β であれば、いかような比率でのブレンドも可能なので、重ねあった状態は無限に多数存在しえるのである。ここでは、その一例として $1/2$ の確率で $|0\rangle$ 、残りの $1/2$ の確率で $|1\rangle$ かもしれない状態を示そう。

まず、確率 $|\alpha|^2$ 、および $|\beta|^2$ がそれぞれ $1/2$ であるので、

$$|\alpha|^2 = |\beta|^2 = \frac{1}{2}$$

となり、 $\alpha = \frac{1}{\sqrt{2}}$ 、 $\beta = \frac{1}{\sqrt{2}}$ であることが分かる。そこで、この $|0\rangle$ と $|1\rangle$ が半々の確率で存在する qubit は以下の式で表される。

$$\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

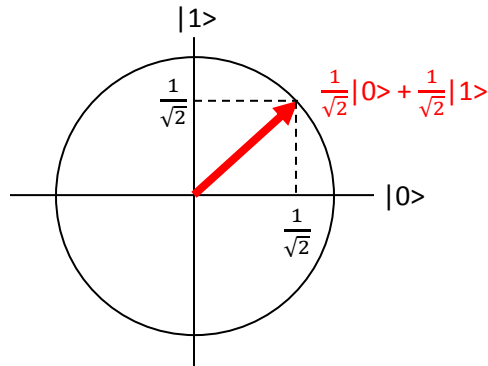
これを先ほどのベクトルの単位円にあてはめると、 $|0\rangle$ 軸方向の成分は「半径 * $\cos \theta$ 」、 $|1\rangle$ 軸方向の成分は「半径 * $\sin \theta$ 」で求められるが、単位円の半径は 1 なので、結局、傾いた角度 θ が分かれば、 α 、 β は以下のように求まる。

$$\alpha = \cos \theta, \quad \beta = \sin \theta$$

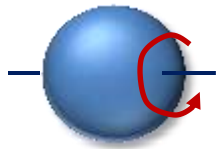
今、 $|0\rangle$ と $|1\rangle$ が $1/2$ ずつの確率で表される状態の α 、 β はどちらも $\frac{1}{\sqrt{2}}$ なので、

$$\cos \theta = \frac{1}{\sqrt{2}}, \sin \theta = \frac{1}{\sqrt{2}}$$

であることが導き出され、 \sin, \cos がともに $\frac{1}{\sqrt{2}}$ となる θ は $\frac{\pi}{4}$ ラジアン、つまり、45 度の方向を指していることが分かる。これは $|0\rangle$ を反時計回りに 45 度回転させた状態である。



この重ね合わせ状態を電子のスピンに置き換えて考えてみた場合、電子はどちらを向いているのだろうか？先に定義したように、ここでも上向き回転を $|0\rangle$ 、下向き回転を $|1\rangle$ とする。上図のようにベクトルは $|0\rangle$ と $|1\rangle$ の中間を向いているので、電子は上向きと下向きの中間、つまり横向きに回転している、と考えるかもしれない。



これが、 $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ なの？

この問いに対する量子力学の答えは、そうかもしれないし、そうでないかもしれない、である。実のところ、電子がどの向きに回転しているかは観測するまで分からない。もしくは、全ての方向を向いて回転している、とも言える。しかし、全く分からないのかというと、そうではなく、だいたいどの方向に向いているか確率で割り出せる。

説明を簡単にするために、ここでは電子は上向き、下向きの二方向にしか回転しないものとしよう。そして、電子の回転方向を十分に多い回数、例えば 1 万回観測したとすると、「 $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ 」で表される状態では、5000 回上向きの状態で観測され、残りの 5000 回は下向きの状態で観測されるのである。

一方、独立状態である $|0\rangle$ の場合は、1 万回観測して、1 万回とも上向きと観測されることを意味している。

二次元複素ベクトル表現

正式には qubit の状態は \mathbb{C}^2 (2-dimensional complex vector space) の単位ベクトルで表される。ここからは、qubit をベクトルで表現し、具体的な演算方法を説明する。

式(1)「 $\alpha |0\rangle + \beta |1\rangle$ 」をベクトル $\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$ と書き表した場合、 $|0\rangle$ および $|1\rangle$ は以下のように定義できる。

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

これは、以下のように変形できることから明らかであろう。

$$\begin{aligned} & \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \\ &= \begin{pmatrix} \alpha \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ \beta \end{pmatrix} \\ &= \alpha \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ &= \alpha |0\rangle + \beta |1\rangle \end{aligned}$$

こうすることで、qubit に対する操作を行列計算として数値モデル化でき、その操作過程や結果も数値演算によりシミュレーションすることが可能となる。

例えば、行列 $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ と $|0\rangle$ の積を計算してみよう。

$$\begin{aligned} & \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} |0\rangle \\ &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 * 1 + 1 * 0 \\ 1 * 1 + 0 * 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ &= |1\rangle \end{aligned}$$

すると $|0\rangle$ が $|1\rangle$ に変換された。同様に $|1\rangle$ に対して掛けると $|0\rangle$ を得るはずである。つまり、この行列 $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ は 1-qubit に対して NOT 操作として作用することが分かる。

確率振幅

ここでは式(1)および(2)で現れた α や β が示す確率振幅について二つの疑問について説明する。特にこの内容を知らなくても以降を読み進むことができるので、興味ない方、すでに理解されている方は読み飛ばして構わない。

(1) 確率振幅の 2 乗が確率になる

式(2)の規格化条件は $|0\rangle$ が出現する確率と $|1\rangle$ が出現する確率を足し合わせれば 1 になることを規定している。部分の確率を合算すれば全体で 1 になるのは当たり前なので、 α 、 β はこの式が成り立つ値しかとりえないと言っているのである。では、なぜ α や β (確率振幅) を 2 乗する必要があるのだろうか？

例えば電子の場合で説明すると、それは電子の振幅と電子の発見確率の関係に依存している。ご存知のように、電子がどこに存在するかは確率的にしか分からない。電子はいつもある一点に存在しているというよりも雲のように空間にぼんやり広がったモノ、つまり電子場として捉えられる。電子場はどこも均一の濃さで広がっているのではなく、場所によって密度が異なる。そして密度が高いところほど電子が発見される確率が高くなることが分かっている。つまり、電子の発見確率は、電子場の密度に比例する。

そして、電子場の密度は、電子の振幅の 2 乗に比例することも分かっている。したがって、電子の発見確率は、電子の振幅の 2 乗にも比例するのである。

$$\begin{aligned}\text{電子の発見確率} &\propto \text{電子場の密度} \\ \text{電子場の密度} &\propto \text{電子の振幅の 2 乗}\end{aligned}$$

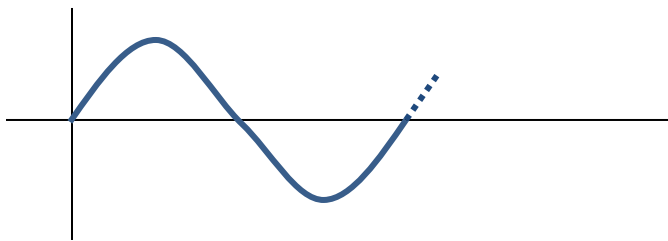
よって

$$\text{電子の発見確率} \propto |\text{振幅}|^2$$

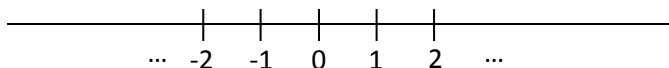
(2) 確率振幅は複素数の値をとる

確率振幅を示す α 、 β は実数を取る場合もあるが、基本的には複素数で表される。なぜ複素数なのだろうか？その理由は複素数を用いて数理モデル化すると、数々の実験で得られた結果と合致するからではあるが、ここではもう少し詳しく考察してみたい。

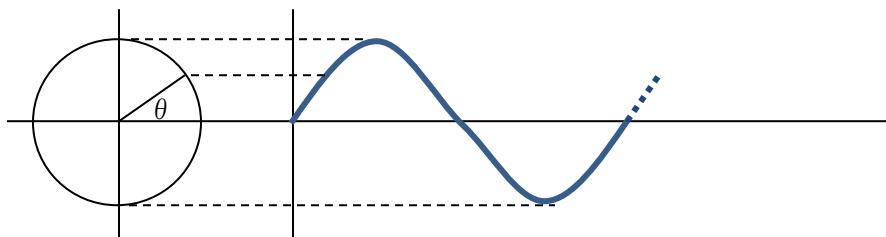
電子や光子といった量子が存在する場所が確率的にしか分からないという現象は、量子が粒子としての性質と同時に、波としての性質も持ち合わせており、干渉現象が生じるからである。



波は上図のような周期性を持つが、実数だけではこの周期性を表現できない。実数は一直線上を埋め尽くす点の集合でしかない。



これに対し、複素数は複素平面で表されることから分かるように二次元の広がりを持ち、今、ある qubit の状態を大きさ 1 のベクトルとすれば、その qubit への演算は大きさ 1 を保ったまま、ベクトルを回転させることを意味する。



例えば、 $1 * i = i$ という演算は x 軸方向を向いた $1(1 + 0i)$ を反時計回りに 90 度回転させた結果 $i(0 + 1i)$ となったと捉えることもできる。この性質が波の持つ周期運動とマッチし、量子の挙動を数式で表す上で都合がいいのである。

n-qubit の計算

ここまでは、計算を行うレジスタとして 1 ビットの qubit を対象にして考察したが、ここでは複数の qubit を組み合わせた多量子ビットの例として、2-qubit での計算を行う。

古典コンピュータで bit が複数桁になる場合、例えば 2 ビットでは 01 と書くが、qubit の場合は各々の状態の組み合わせで表す。例えば、 $|0\rangle$ と $|1\rangle$ の組み合わせた 2-qubit は以下のように記述する。

$$|0\rangle \otimes |1\rangle$$

または、演算記号を省略して単に

$$|01\rangle$$

と書くことも多い。また、本稿では使用しないが、ケット内の数列の桁が増えた場合、10 進数で表すこともある。

$$|101\rangle = |5\rangle$$

なお、 \otimes の記号は、 $|0\rangle$ と $|1\rangle$ の直積（テンソル積 Tensor Product）をとることを意味している。そこで、テンソル積について復習しておこう。ここでは qubit を行列で表現しているので、行列に対してのテンソル積について説明する。

今、下記の値を持つ 2×2 行列 A、B が与えられたとしよう。

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

このとき、A と B のテンソル積は以下の行列で定義される。

$$A \otimes B = \begin{pmatrix} a_{11} * b_{11} & a_{11} * b_{12} & a_{12} * b_{11} & a_{12} * b_{12} \\ a_{11} * b_{21} & a_{11} * b_{22} & a_{12} * b_{21} & a_{12} * b_{22} \\ a_{21} * b_{11} & a_{21} * b_{12} & a_{22} * b_{11} & a_{22} * b_{12} \\ a_{21} * b_{21} & a_{21} * b_{22} & a_{22} * b_{21} & a_{22} * b_{22} \end{pmatrix}$$

A の各要素について B の全ての要素を掛けていく。そのため、A が $k \times l$ 行列で B が $m \times n$ 行列の時、A と B のテンソル積は $km \times ln$ 行列になる。この例の場合、 2×2 行列を与えたので、そのテンソル積後の行列は 4×4 行列となった。また、結合法則も成り立つ。

$$(A \otimes B) \otimes C = A \otimes (B \otimes C)$$

それでは、話を複数 qubits での計算に戻そう。1-qubit をベクトルで表現できたように、2-qubit もベクトル表現可能である。では、どのように表現するのだろうか？ $|0\rangle \otimes |1\rangle$ をベクトルで計算してみよう。

$$|0\rangle \otimes |1\rangle$$

$$= \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 * 0 \\ 1 * 1 \\ 0 * 0 \\ 0 * 1 \end{pmatrix}$$

$$= \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

この様に 2-qubit は 4 行のベクトルで表現される（2 行 1 列と 2 行 1 列のテンソル積なので、その積は 4 行 1 列のベクトルとなる）。なお、この 4 行 1 列のベクトルは、上から $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ の 4 つの状態を表している。そして、今、このベクトルは上から 2 番目のみ 1 なので、 $|01\rangle$ という状態であることが分かる。

$$\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \begin{array}{l} |00\rangle = 0 \\ |01\rangle = 1 \\ |10\rangle = 2 \\ |11\rangle = 3 \end{array}$$

他の 2-qubit 列についても同様に計算すると以下のようになり、これが 2-qubit 版ベクトル表現である。

$$|00\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$|01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

$$|10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

$$|11\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

ついでに式(1)と式(2)の 2-qubit 版も以下に定義しておく。

$$|\psi\rangle = c_0|00\rangle + c_1|01\rangle + c_2|10\rangle + c_3|11\rangle \quad \text{式(3)}$$

ただし、 c_n は複素数であり、以下の規格化条件を満たすものとする。

$$|c_0|^2 + |c_1|^2 + |c_2|^2 + |c_3|^2 = 1 \quad \text{式(4)}$$

それでは、重ね合わせの状態にある qubit が 2 ビットあった場合は、どうなるだろうか？
 ここでは、 $|0\rangle$ と $|1\rangle$ が $1/2$ の確率で重ね合わさっている状態を考えてみる。

$$\begin{aligned}
 & \left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) \otimes \left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) \\
 &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\
 &= \left(\frac{1}{\sqrt{2}} * \frac{1}{\sqrt{2}}\right)(|0\rangle + |1\rangle \otimes |0\rangle + |1\rangle) \\
 &= \frac{1}{2} \left(\begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right) \otimes \left(\begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right) \\
 &= \frac{1}{2} \left(\begin{pmatrix} 1 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right) \\
 &= \frac{1}{2} \begin{pmatrix} 1 * 1 \\ 1 * 1 \\ 1 * 1 \\ 1 * 1 \end{pmatrix} \\
 &= \frac{1}{2} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \\
 &= \frac{1}{2} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \frac{1}{2} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} + \frac{1}{2} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \frac{1}{2} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \\
 &= \frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle
 \end{aligned}$$

となり、 $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$ までのすべての組み合わせが同確率 ($1/4$) で存在する状態を表すことが分かる。各状態の係数 (確率振幅) が $1/2$ ということは

$$\left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2 = 4 * \frac{1}{4} = 1$$

となり、式(4)の規格化条件も満たしていることに注意されたい。

ちなみに、上記では計算過程を詳しく示すために、ベクトルに変換して計算したが、ケット記号のまま計算しても同じ結果が得られる。

$$\begin{aligned}
 & \left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) \otimes \left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) \\
 &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\
 &= \left(\frac{1}{\sqrt{2}} * \frac{1}{\sqrt{2}}\right)(|0\rangle + |1\rangle \otimes |0\rangle + |1\rangle \otimes |1\rangle) \\
 &= \frac{1}{2}(|0\rangle \otimes |0\rangle + |0\rangle \otimes |1\rangle + |1\rangle \otimes |0\rangle + |1\rangle \otimes |1\rangle) \\
 &= \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) \\
 &= \frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle
 \end{aligned}$$

この 2-qubit での計算で注目すべき点は、重ね合わせ状態の 1-qubit では $|0\rangle$ と $|1\rangle$ の 2 通りの状態を表現したが、2-qubit に拡張すると $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$ の 4 通りが表現できたところである。同様に 3-qubit にすると $|000\rangle$, $|001\rangle$, $|010\rangle$, $|011\rangle$, $|100\rangle$, $|101\rangle$, $|110\rangle$, $|111\rangle$ の 8 通りが表現できる。つまり、ビット数が n 倍になると、同時に表現できる情報量は 2^n 倍になるのである。

古典コンピュータの 2 ビットも 00, 01, 10, 11 の 4 通りを表現できると思うかもしれないが、それはデータの組み合わせとして 4 通りあるだけで、その 2 ビットがある瞬間に持てる値はその 4 通りの中の 1 つだけである。しかし、量子コンピュータではその全てを同時に保持でき、一度に並列計算できる点で大きく異なる。これこそが組み合わせ問題を解く際に大量計算を一気にこなす秘密なのである。

観測する

「観測する」とは、qubit の値が 0 なのか 1 なのかを調べることである。古典コンピュータではビット情報を参照する行為にわざわざ「観測」などという大げさな言葉は使わない。値が 0 のビットは常に 0 だし、値が 1 のビットは常に 1 だからである。しかし、量子の世界では観測する前と後とでは状態が変わるため、観測という行為をいつ行うかが演算結果に影響を及ぼす。

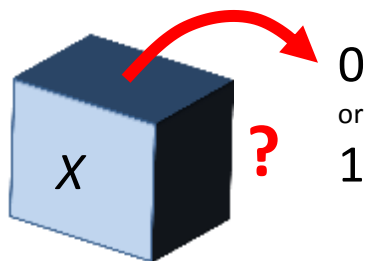
独立状態である qubit を観測する場合は、古典コンピュータ同様に一意に決定される。

$|0\rangle$ を観測すると、0 が得られる。同様に、 $|1\rangle$ を観測すると、1 が得られる。

しかし、問題は重ね合わさった状態の時に起きる。今、ある 1-qubit が以下の式で表される状態（確率 $1/2$ で重ね合わさった状態）だとして。

$$\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

この qubit を観測すると値は 0 になるだろうか？それとも 1 になるだろうか？この段階ではどちらとも断定できない。ただ言えるのは確率 $1/2$ で 0、確率 $1/2$ で 1 となるだろうということである。実際、観測を実行すると、0 が返却されたり、1 が返却されたりする。ただし、ひとたび観測すると、重ね合わせの状態は独立状態へと変わり、それ以降は同一 qubit を何度観測しても同じ値を返却するようになる。



この観測するまでは 2 つの状態が同時に存在し、そのどちらであるかは確率的にしか分からない。しかし、観測するとどちらか一方の状態に収束し、それ以降もその状態のままでい続けるという奇妙な概念は量子の世界から導入されたものである。物理学の世界では長年、量子（例えば、原子、電子、光子など）は粒なのか、それとも波なのかが議論されているが、現在では、その両方の性質を持つ、とされている。そして、観測しない場合、量子は波としての性質を示すが、ひとたび観測すると粒子としての性質を示す。この奇妙な振る舞いの正体はまだ解明されていないが、この量子の振る舞いが発見された以降の百年以上の間、多くの物理学者が実験を重ねてきて、皆同じ結果を得ている。つまり、何故そ

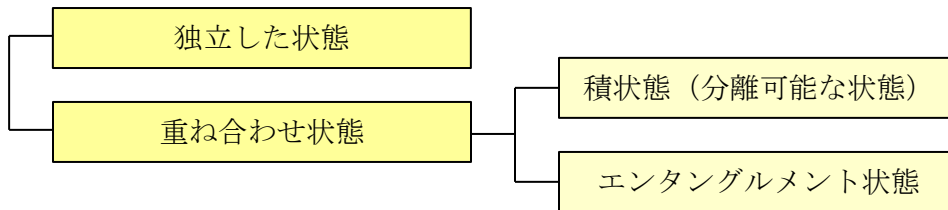
うなるのかは分からないが、「こうすれば、こうなる」的な操作は把握でき、かつ、同じ結果を再現できるため、この現象を量子コンピュータに応用できるのだ。

しかしながら、計算結果が **0** なのか **1** なのか決定しないというのでは困ってしまう。そこで、通常のプログラミングでは並列計算を行っている間だけ重ね合わせの状態を維持し、結果を求める前に、何らかの変換操作で一意に決定できる値へと収束させる工夫を行っている。

または、式(1)で係数 α 、 β で表した確率振幅を増大させ、取り出したい値の振幅だけ増大させておき、観測するというも行われたりする。

エンタングルメント状態（量子もつれ）

これまで、qubit の取り得る状態として独立した状態と重ね合わさった状態の二種類を紹介した。独立状態とは $|0\rangle$, $|1\rangle$ のように特定の値しか含まない状態であった。そして、 $|0\rangle$ と $|1\rangle$ の重なり合った状態を重ね合わせ状態といい、 $|0\rangle$ と $|1\rangle$ が同時に存在する状態を指した。この重ね合わせ状態は、更に二つの状態に分類できる。それは分離可能な状態（積状態; product state）とエンタングルメント状態（量子もつれ; entangled state）である。



エンタングルメント状態とは、qubit の構成要素が絡み合って分離できない状態を指す。例えば、以下に示す 2-qubit の状態を見てみよう。

$$\frac{1}{\sqrt{2}}(|00\rangle + |01\rangle)$$

係数（確率振幅）が $\frac{1}{\sqrt{2}}$ なので、この qubit を観測すると $1/2$ の確率で $|00\rangle$ 、残り $1/2$ の確率で $|01\rangle$ が得られるはずである。まず、左ビットを観測すると常に $|0\rangle$ が得られる。これは両方の状態の左ビットが 0 だから、どちらが選ばれても 0 となるからである。次に、右側のビットが $|0\rangle$ であるか $|1\rangle$ であるかは、その右ビットを観測するまで分からない。つまり、左ビットは右ビットと何ら関係なく値が決まる。確かに、上記式は以下のようにも変形できる。

$$\frac{1}{\sqrt{2}}(|00\rangle + |01\rangle)$$

$$= \frac{1}{\sqrt{2}}(|0\rangle \otimes |0\rangle + |0\rangle \otimes |1\rangle)$$

$$= \frac{1}{\sqrt{2}}(|0\rangle \otimes (|0\rangle + |1\rangle))$$

この式は、左ビットが常に $|0\rangle$ となり、それとは無関係に右ビットは $|0\rangle$ または $|1\rangle$ になることを示している。このようにテンソル積で分離可能な状態はエンタングルメントな状態とは言わない。

次の例として、以下の 2-qubit の状態を見てほしい。

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

先の例とよく似ているが、右側のケットが 11 になっている点が違う。

この左ビットを観測すると 1/2 の確率で |0>、残り 1/2 の確率で |1>となることが分かる。そして重要なのは、一度左ビットの値を得れば、右ビットを観測しなくてもその値を知ることができる。つまり、左ビットが |0>ならば、右ビットも |0>であり、左ビットが |1>ならば右ビットも |1>となる。先の節でも述べたとおり、一度観測するとそのビットは |0>または |1>に収束するため、このような結果を得るのである。

ここで、この 2-qubit の左ビットを観測して値 0 を得たとしよう。その瞬間、左ビットの値は 0 に確定され、1 をとる可能性は 0 となる。つまり、右の項 |11>は状態として、もはや存在し得ないのである。そのため、|00>だけが qubit 内に残り、その確率振幅は 1 となる（|00>しか存在しないので、以降、確率 1 で |00>が観測される）。

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

- qubit 内に |00>と |01>の二つの状態がある
- 左ビットを観測して、0 を得た！（|11>は消える）
- qubit 内の状態は「|00>」だけとなる

同様に以下の 2-qubit もエンタングルメント状態となっている。

$$\frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$$

この場合は、右ビットは常に左ビットの反転となる値をとる。つまり、左ビットで 0 が観測されれば右ビットは 1 となり、左ビットで 1 が観測されれば右ビットは 0 となる。

これらの 2-qubit は分離可能な状態のようにテンソル積で分けることができない（分離可能でないため）。

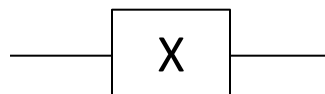
この状態を電子の場合で説明すると、観測によって電子が上向きに回転していることが確定すると、もう一方の電子は瞬時に下向きに確定される。これら二つの電子の間にある種の保存則が働き、後者の電子は前者の電子の回転を打ち消す向きに回転しなければいけないためである。

2. 量子論理ゲート

古典コンピュータで論理演算を行う素子をゲートと呼び、NOT, AND, OR, XOR, NAND ゲートを組み合わせて複雑な計算を行うように、量子コンピュータでもゲートがある。ここでは量子論理ゲートの基本概念（ユニタリゲート、制御 NOT ゲート）の説明は省略し、実際の演算で多用するゲートの紹介のみに留めたい。

パウリ X ゲート (NOT ゲート)

1-qubit に対するゲートで、入力ビットを反転し、出力ビットとする。つまり、入力が $|0\rangle$ なら $|1\rangle$ を出力し、入力が $|1\rangle$ なら $|0\rangle$ を出力する。いわゆる NOT ゲートのことであり、回路図では以下の図で表される。



パウリという名前はパウリのスピン行列と同じ働きをすることから由来している。パウリゲートにはパウリ X の他にパウリ Y、パウリ Z ゲートもあるが、ここではパウリ X のみ取り上げる。

パウリ X は以下の行列で表される。

$$\text{パウリ X} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

それでは、この行列を使って、実際に演算してみよう。

$$X|0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \cdot 1 + 1 \cdot 0 \\ 1 \cdot 1 + 0 \cdot 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle$$

$$X|1\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \cdot 0 + 1 \cdot 1 \\ 1 \cdot 0 + 0 \cdot 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle$$

期待通り、どちらのビット入力に対しても反転ビットが得られた。電子で言うと、上向きのスピンを下向きのスピンに変えるような操作を指す。

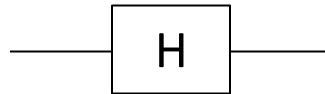
アダマール(Hadamard)ゲート

1-qubit に対するゲートで、重ね合わさった qubit を出力する。 $|0\rangle$ または $|1\rangle$ の入力に対して、以下のような値を出力する。

$$|0\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

$$|1\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

回路図では以下のように表される。



そして、演算においては以下の行列 H で表される。

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

ここでも行列 H を利用して演算してみよう。

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \cdot 1 + 1 \cdot 0 \\ 1 \cdot 1 - 1 \cdot 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \left(\begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right) = \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle)$$

$$H|1\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \cdot 0 + 1 \cdot 1 \\ 1 \cdot 0 - 1 \cdot 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \frac{1}{\sqrt{2}} \left(\begin{pmatrix} 1 \\ 0 \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right) = \frac{1}{\sqrt{2}} (|0\rangle - |1\rangle)$$

このように、 $|0\rangle$ または $|1\rangle$ といった値の決まった qubit の入力に対し、 $|0\rangle$ と $|1\rangle$ が半々の確率で存在する重ね合わさった値を出力している。そのため、多くの量子アルゴリズムで、並列計算を行う前に計算量を増やす目的でアダマールゲートを呼び出す操作が見られる。光子の場合で説明すると、 x 偏光 (x 軸方向に偏向された光子) を反時計回りに 45 度回転させ、 y 偏光 (y 軸方向に偏向された光子) を右下 45 度の向きに回転させる操作にあたる。このように重ね合わさった状態を作り出すと、ビット列が一体になったかのように振る舞い、一つの qubit に対する操作で他方のビットも同時に操作できるようになる。これこそが、量子コンピュータは単なる並列処理でなく、並列を超えた「超並列処理」と呼ばれる所以である。

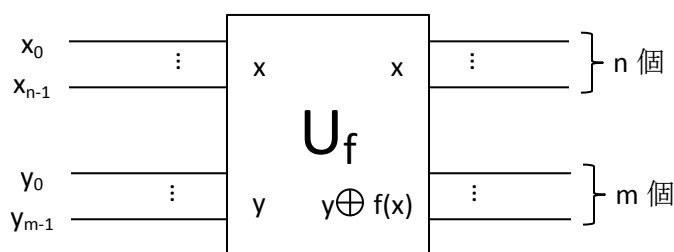
3. Function

量子論理ゲートは、システム側であらかじめ用意された **built-in** 関数のようなもので、どのようなアルゴリズムを実装するにしても、よく使われる基本的なビット操作を提供している。しかし、特定のアルゴリズムを実装する場合、そのような基本操作の組み合わせだけでなく、独自のロジックを組み込んだ **custom** 関数が必要になることが多い。

本章では、そうしたアプリケーション固有のビット操作を行う関数の作成と実行方法について説明する。

U_f ゲート

まず、関数を実行するための論理ゲート（ここでは U_f と記述する）を考える。以下に U_f ゲートの一般的な図を示す。



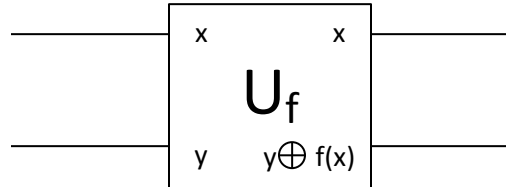
このゲートは入力に n 個の x ($x_0, x_1, x_2, \dots, x_{n-1}$) と m 個の y ($y_0, y_1, y_2, \dots, y_{m-1}$) を受け取る。そして、 x は何も変換せずにそのまま出力し、 y は関数 $f(x)$ を実行した結果と y の元の値と加算し、その一桁目のみを出力する（つまり桁あふれした部分は捨てる）。ここでは、 \oplus の記号は 1 ビットの加算を意味している。なお、ここで単に x, y と記した場合、それは要素 $x_0, x_1, x_2, \dots, x_{n-1}$ を含む集合としての x, y を意味することに注意されたい。

U_f ゲート自体はシステム側から提供され、関数 $f(x)$ がプログラマーによって作成するカスタム関数だと理解してもよい。

ところで、古典コンピュータの概念で考えると、出力側に x は不要なのでは？と疑問に思うかもしれない。確かに変換しないのだからわざわざゲートを通して出力する必要はない。しかし、量子論理ゲートでは入力数と出力数が同じなければならないという制約がある。それは、量子計算では二つの基底状態 ($|0\rangle$ と $|1\rangle$) をもつ 2 次元複素空間内での操作と等価であるため、出力側から入力側へ逆方向にも戻れなければならないためである。このような変換をユニタリー変換 (Unitary transformation) といい、複素平面上で大きさ 1 のベ

クトルを保ったまま方向を変える変換と理解して欲しい（単位円の円周上をグルグル回っている様子を想像してもらえると理解しやすいかもしれない）。

ここで具体例を示そう。問題を簡単にするため、ここでは入力 qubit は x と y の 2 ビットだけだとしてよう。



そして、入力ビット x を反転する関数 $f(x)$ を作成したとする。その関数は以下のように実装できる。

```
function f(x) {  
  return x == 0 ? 1 : 0;  
}
```

この他にも実装方法はあるだろうが、要は、 $x = \{0, 1\}$ であれば、 0 が渡されれば 1 を返し、 1 が渡されれば 0 を返す関数である。

さて、今、 U_f ゲートに $|00\rangle$ が渡されたとしてよう。左ビットから x, y に割り当てるものとした場合、入力値は以下ようになる。

$x = 0,$
 $y = 0$

そして、出力は以下ようになる。

$x = 0,$
 $y = 0 \oplus f(0) = 1$

つまり、 $|00\rangle$ の入力に対して $|01\rangle$ が出力された ($y=0$ が $y=1$ に変換された)。

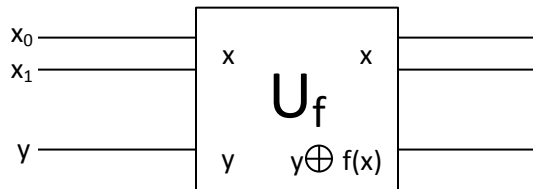
同様に $|01\rangle$ に対しては $|00\rangle$ 、 $|10\rangle$ に対しては $|10\rangle$ 、 $|11\rangle$ に対しては $|11\rangle$ となる。

これより U_f はビット反転関数 $f(x)$ とともに操作すると、入力 x (左ビット) が $|1\rangle$ の場合は何も変更しないが、 $|0\rangle$ だと右ビットを反転させるゲートとして作用することが分かる。

超並列実行

前節で定義した関数が超並列的に実行されることを理解するため、古典コンピュータ向けのプログラミング言語で同等の処理を実装してみる。

ここでは、 x_0, x_1, y の 3 ビットを入力とする U_f ゲートを考える。



例えば、 $|100\rangle$ を左ビットから x_0, x_1, y に割当てると、

$$x_0 = 1, x_1 = 0, y = 0$$

を入力として受け取り、

$$\begin{aligned} x_0 &= 1, \\ x_1 &= 0, \\ y &= 0 \oplus f(2) = 0 \oplus f(1) \oplus f(0) = 1 \end{aligned}$$

を出力として得る。つまり、 $|100\rangle$ の入力に対し、 $|101\rangle$ を得た。
以上の処理を JavaScript で実装すると以下のようなになる。

リスト 1: JavaScript による U_f ゲートの実装

```
1 <script>
2 function f(x) {
3   return x == 0 ? 1 : 0;
4 }
5
6 window.onload = function() {
7
8   var bits = [0, 0, 1];
9
10  bits[0] += f(bits[1]);
11  bits[0] += f(bits[2]);
12  bits[0] %= 2;
13 }
```



```

14     console.log("result = "+bits[0]);
15
16 };
17 </script>

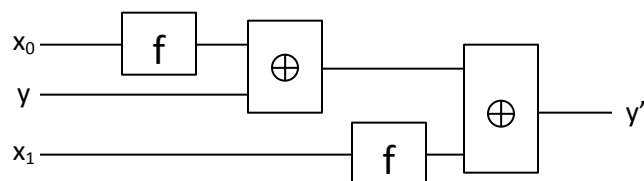
```

$|100\rangle$ は配列[0, 0, 1]として表され、bits[1]および bits[2]に対して関数 f を実行している。最後に $\%2$ としているのは桁あふれの部分を切り落とし、1 桁目のみのビット値を得るためである。

実行すると、期待通り結果が 1 と表示される。

```
result = 1
```

この JavaScript のコードを擬似的な回路図として描き表したのが以下の図である。



f ゲートが 2 回出現しているのが見て取れる。これは、関数 f が 2 回コールされることを意味している。それに比べ、 U_f ゲート図では U_f は 1 回しか出現しない。つまり、同じ演算を行うために関数 f は 1 回しか呼び出されないのである。これが更に重ね合わせの状態を入力として得た場合は、さらにその倍の 4 つの状態を一気に 1 回の関数コールで処理するのである。それに対し、JavaScript では 4 回の関数コールが必要となる。これこそが量子プログラミングにおける超並列処理の威力なのである。

4. JavaScript シミュレータによる量子プログラミング

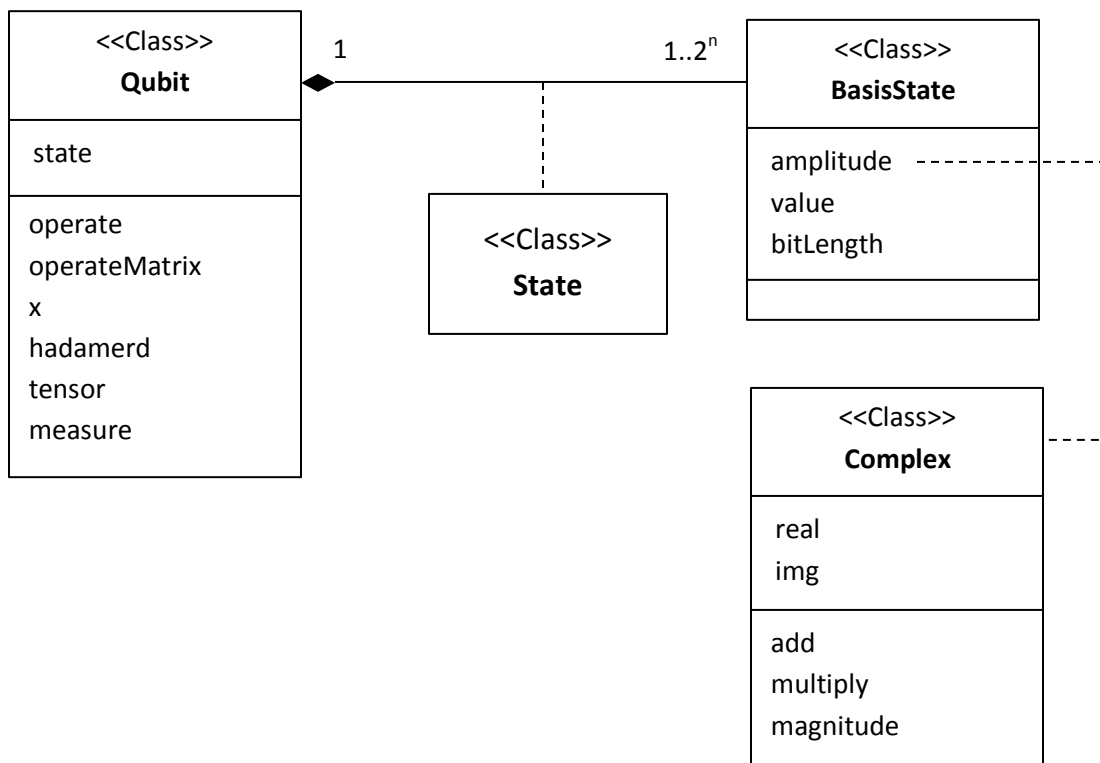
いよいよ量子プログラミングを実際に体験してみよう。本稿では JavaScript で実装したシミュレータである jsqb を用いる。jsqb は本稿で説明した内容を実体験する目的で作られたものである。そのため、機能は少ないがソースコードも 500 行以下と非常に小さく、コードを追って理解できるように配慮している。

なお、jsqb の JavaScript ファイルやサンプルコードは以下のサイトで公開している。また、本ドキュメントも同サイトで公開しているので、更新、訂正についてはそちらを参照願いたい。

<https://github.com/jsqb>

クラス図

qubit の構成要素を復習する意味でも jsqb のクラス図の一部を示す。



これらクラスが、qubit およびそれが保持する状態(state)を表している。まず、qubit は $|0\rangle$ や $|1\rangle$ などの basis state(basis vector)の線形的な集まりであり、例えば、1-qubit の場合は以下の式で表された。

$$|\phi\rangle = \alpha |0\rangle + \beta |1\rangle$$

ここで、 α 、 β は $|0\rangle$ 、 $|1\rangle$ が出現する確率振幅を表しており、値としては複素数を取り得る。2-qubit になると basis state としてとり得る値が $|00\rangle$ 、 $|01\rangle$ 、 $|10\rangle$ 、 $|11\rangle$ の4種類となり、以下の式で表された。

$$|\phi\rangle = c_0|00\rangle + c_1|01\rangle + c_2|10\rangle + c_3|11\rangle$$

つまり、n-qubit の中に 2^n 個の basis state を c_n の割合で保持することになる。jsqb のクラスもその構成をとっている。クラス図に戻って見てみると、Qubit クラスは一つの状態を state プロパティとして保持し、その state は複数の BasisState クラスと関連付けられている。そして、BasisState クラスには確率振幅を格納する amplitude プロパティ、qubit 値を格納する value プロパティ、そして qubit 値のビット長を格納する bitLength プロパティから成る。この3つのプロパティで一つの項 (basis state) を表現している。例えば、上記式の $c_3|11\rangle$ の項は以下のように BasisState クラスに格納される。

| <<Instance>> BasisState | |
|----------------------------|--|
| amplitude = c_3 | |
| value = 3 | |
| bitLength = 2 | |
| | |

なお、value プロパティには10進数で格納するため、11は3として格納する。しかし、それだと3を2進数に戻す時、11、011、0011等のどの値を表していたかが分からなくなるため、ビット長「2」を bitLength プロパティに保持しておく。また、確率振幅を表す amplitude プロパティは複素数の値をとるため、Complex クラスのインスタンスを格納することになる。

以上がクラス図の概要である。より細かい説明は後回しにして、まずはコードを実行して、どのような動きになるかを体感しよう。

新しい言語を学ぶ際、各入門書の最初のサンプルとして **Hello World** を出力することが慣例となっているが、残念ながら **jsqb** ではそんな高度なコードは書けない。マシン語で文字列を表示する時はシステムコールで行うように **jsqb** も **JavaScript** の機能をコールすることで可能とも言えるが、ここでは、これまでに紹介した量子計算や論理ゲートをコード化し、計算どおりに動作することを確認したい。

NOT 演算

パウリ行列 $X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ が 1-qubit を反転させることを示し、以下の演算を紹介した。

$$X |0\rangle = |1\rangle$$

$$X |1\rangle = |0\rangle$$

上記を jsqb で実装すると以下のコードとなる。

リスト 2 : Pauli X によるビット反転

```
1 <script src="jsqb.js"></script>
2 <script>
3   window.onload = function() {
4
5       var X = [[0, 1],
6                [1, 0]];
7
8       debug("X |0> = " + jsqb('|0>').operateMatrix(0, X));
9
10      debug("X |1> = " + jsqb('|1>').operateMatrix(0, X));
11
12  };
13 </script>
```

jsqb を利用するため、1 行目で JavaScript ファイル「jsqb.js」を読み込む。

そして、ページの読み込みが完了してからコードを実行したいので、window.onload に設定した function 内に処理コードを記述している（3 行目）。これは debug 関数でブラウザ画面上にメッセージを表示するためである。

次に、5, 6 行目でパウリ行列 X を二次元配列として定義する。

そして、いよいよ jsqb の登場である。まずは、jsqb にケット文字列を与え、qubit オブジェクトを作成する（8 行目の「jsqb('|0>）」の部分）。

```
Qubit = jsqb ( ketString );
```

引数にケット文字列、例えば、|0>, |1>, |101>等を指定すると、その値を表現する Qubit オブジェクトが返却される。

そして、次にパウリ X との積をとるため、operateMatrix 関数をコールする。

```
Qubit = qubit.operateMatrix ( targetBit, matrix );
```

targetBit は qubit のどのビットに対して処理をするかを指定する。右から左に向かって、0 ビット目、1 ビット目、2 ビット目、...と数える。

ここでは「0」が指定されているので、 $|0\rangle$ の 0 ビット目、つまり「0」に対して変換を行う。 $|0\rangle$ はベクトル $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ で与えられるので、ここでは行列 $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ とベクトル $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ の積を計算して「0」のあった場所にセットする、という操作を行っている。

10 行目も入力値が $|1\rangle$ となるだけで同様な処理となる。実行すると以下のように表示され、期待通り反転した値が得られた。

```
X |0> = |1>  
X |1> = |0>
```

テンソル積

複数の qubit を組み合わせて処理する場合、その組み合わせはテンソル積で表現できる。
例えばここに $|0\rangle$ と $|1\rangle$ があり、それらをつなぎ合わせ、2-qubit として処理する場合、それは、

$$|0\rangle \otimes |1\rangle$$

と書き表した。また、演算記号を省略し、単に $|01\rangle$ と書いてもよい。
重ね合わさった状態の場合も同様に、例えば

$$\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) \otimes \left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right)$$

と書き表された。この式の場合、直積計算を進め、より単純化すると以下のように表される。

$$= \frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle$$

上記に示した二通りの計算を行うコードを以下に示す。

リスト 3 : Tensor Product

```
1 <script src="jsqb.js"></script>
2 <script>
3   window.onload = function() {
4
5       debug("|0> (x) |1> = " + jsqb('|0>').tensor(jsqb('|1>')));
6
7       var qubit = jsqb('|0>+|1>');
8       debug(qubit+" (x) "+qubit+" = " + qubit.tensor(qubit));
9
10  };
11 </script>
```

そして、以下がその実行結果である。

```
|0> (x) |1> = |01>

0.7071 |0> +0.7071 |1> (x) 0.7071 |0> +0.7071 |1> = 0.5 |00> +0.5 |01> +0.5 |10> +0.5 |11>
```

まず、5 行目では $|0\rangle$ (x) $|1\rangle$ の計算を行っている。テンソル積は Qubit オブジェクトの `tensor` メソッドで求められる。

```
Qubit = qubit.tensor ( qubit );
```

メソッド `tensor` に `qubit` オブジェクトを与えると、自身の `qubit` とのテンソル積を行った結果の `qubit` を返却する。

ここでは、 $|0\rangle$ を表す Qubit オブジェクトで `tensor` メソッドを実行し、 $|1\rangle$ を表す Qubit オブジェクトを引数に渡している。したがって、その答えは $|01\rangle$ となる。

7, 8 行目では、 $|0\rangle$ と $|1\rangle$ が重ね合わさった状態同士のテンソル積を求めている。まず、重ね合わせ状態の `qubit` を作成するため、`jsqb` に " $|0\rangle + |1\rangle$ " という文字列を与えている（7 行目）。なお、本来この表現は正しくない。規格化条件 ($|\alpha|^2 + |\beta|^2 = 1$) より、確率振幅の二乗の和は 1 でなければならない。そのため、正しくは " $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ " と書くべきだが、ここでは確率振幅を省略している。Qubit オブジェクトを生成する際、規格化条件に合致するよう確率振幅を自動で再計算してくれるからだ。実際に、その次の行で `qubit` 変数を表示しているが、その実行結果をみると「0.7071 $|0\rangle$ + 0.7071 $|1\rangle$ 」という正しい値になっている。なお、0.7071 は $\frac{1}{\sqrt{2}}$ を小数で表した値である。

これで変数 `qubit` に重ね合わせ状態の `qubit` オブジェクトが格納できたので、あとはそれを `tensor` メソッドで積をとればよい。

実行結果には先に手計算した結果と同じ値（00, 01, 10, 11 の四状態がそれぞれ振幅 1/2 で重ね合わさった状態）が表示されている。

qubit の観測

独立状態の qubit の観測は $|0\rangle$ なら 0、 $|1\rangle$ なら 1 と一意に求められる。しかし、重ね合わせ状態にある qubit の場合、その観測結果は確率的に決定される。

その様子を jsqb のコードを用いて確認しよう。qubit を観測するには `measure` メソッドを用いる。

```
Measurement = qubit.measure ( targetBit );
```

この `measure` メソッドは観測したいビット番号（右から数え、0 から始まる）を受け取ると、その位置のビット値と観測後の qubit オブジェクトをプロパティとして保持した `Measurement` オブジェクトを返却する。

| <<Class>> Measurement |
|--------------------------|
| qubit value |
| |

「観測する」の節でも述べたが、qubit は観測すると観測したビットは 0 または 1 に収束し、その値以外のビット値を含む状態は消滅する。そのため、`measure` を実行する前と後とでは qubit の状態が異なるため注意が必要である。以下がそのことを確認するための具体的なコードである。

リスト 4 :Measurement

```
1 <script src="jsqb.js"></script>
2 <script>
3   window.onload = function() {
4
5       debug("|0> measure(0) = " + jsqb('|0>').measure(0));
6       debug("|100> measure(2) = " + jsqb('|100>').measure(2));
7
8       var qubit = jsqb('|0> + |1>');
9       var qubit2 = qubit.clone();
10
11       debug(qubit+" measure(0) = " + qubit.measure(0));
```

```

12      debug(qubit+" measure(0) = " + qubit.measure(0));
13      debug(qubit+" measure(0) = " + qubit.measure(0));
14
15      debug(qubit2+" measure(0) = " + qubit2.measure(0));
16
17  };
18  </script>

```

このサンプルを実行すると以下の出力を得る。なお、これは一例であり、実行のたびに異なる値を出力するので注意されたい。

```

|0> measure(0) = 0
|100> measure(2) = 1

0.7071 |0> +0.7071 |1> measure(0) = 0
|0> measure(0) = 0
|0> measure(0) = 0

0.7071 |0> +0.7071 |1> measure(0) = 1

```

まず、5 行目は $|0\rangle$ を観測し、0 を得ている。 $|0\rangle$ は独立状態なので、F5 ボタンを押してブラウザ画面を何度リロードしても常に 0 が表示されるはずである。

6 行目は独立状態にある 3-qubit の 2 ビット目（0 ビット目から始まるので、実際には右から 3 ビット目）を観測している。 $|100\rangle$ の 2 ビット目は 1 なので常に 1 が返却される。

8 行目は $|0\rangle$ と $|1\rangle$ が同確率で重ね合わさった状態の Qubit オブジェクトを生成し、変数 `qubit` に格納している。そして、9 行目でその `qubit` と同じ状態のコピーをもつ `qubit2` も作成しておく。今、`qubit` および `qubit2` 変数は下記の式で表される状態にある。

$$\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

確率振幅が $\frac{1}{\sqrt{2}}$ なので、確率 $1/2$ で $|0\rangle$ 、残り $1/2$ で $|1\rangle$ になることを意味している。つまり、同確率で 0 または 1 が観測されるはずである。

まず、11 行目で `qubit` 変数に対して観測を行う。すると、0 が表示されたり 1 が表示されたりと、定まった値にはならない。しかし、例えば 11 行目で 0 が観測されたとすれば、12, 13 行目でも必ず 0 が観測される。それは 11 行目の観測によって `qubit` の状態は重ね合わせ状態から $|0\rangle$ という独立状態へと収束したためである。

しかし、観測前にコピーした `qubit2` では収束は起こっていないため、15 行目で観測を行うと、11 行目の観測結果に関係なく、0 であったり、1 であったりする。

なお、余談であるが `clone` メソッドは便宜上用意した機能で、現時点では実際の量子コンピュータ上では実装できない機能であることに留意して欲しい。なぜなら、`qubit` の状態を一度観測するとその状態が変化するため、元の状態を知ることができないからである。コピーするためには `qubit` の中を覗かなければならないが、そうすると変化した状態をコピーすることになる。このことは数学的にも「量子複製不可能定理」として証明されている。完全なコピーでなくてもコーディング上、簡単にクローンオブジェクト作成できる方法が発見されるまで量子コンピュータ上では `clone` メソッドの実装は不可能なのである。

エンタングルメント状態の観測

次に、重ね合わせ状態の更に特殊な状態であるエンタングルメント状態の **qubit** を観測してみる。エンタングルメント状態の **qubit** とはビット同士が絡み合い、分離できない場合を指す（「量子もつれ」とも言う）。例えば **2-qubit** の場合、左ビットを観測したら、右ビットを観測しなくてもその値が分かる場合である。

そのことを以下のコードを実行して確認しよう。

リスト 5 : Measure entangled qubits

```
1 <script src="jsqb.js"></script>
2 <script>
3   window.onload = function() {
4
5       var qubit = jsqb('|00> + |11>');
6       var value1 = qubit.measure(1).value;
7       var value0 = qubit.measure(0).value;
8       debug("|00> + |11> measure => (" + value1 + ", " + value0 + ")");
9
10      var qubit = jsqb('|01> + |10>');
11      var value1 = qubit.measure(1).value;
12      var value0 = qubit.measure(0).value;
13      debug("|01> + |10> measure => (" + value1 + ", " + value0 + ")");
14
15  };
16 </script>
```

実行結果はランダムに変化する。起こりうる実行結果の内、代表的な二通りを以下に示す。

実行結果 1

```
|00> + |11> measure => (0, 0)
|01> + |10> measure => (0, 1)
```

実行結果 2

```
|00> + |11> measure => (1, 1)
|01> + |10> measure => (1, 0)
```

まず、5行目で $|00\rangle$ と $|11\rangle$ の重ね合わせ状態にある qubit を作成する。そして、6行目で1ビット目（左ビット）を観測し、観測結果の値を `value1` に格納する。7行目では同様に0ビット目（右ビット）を観測し、値を `value0` に格納する。そして、8行目でそれらの値を表示している。

ここで、このコードを何度か実行してみて欲しい。実行する毎に得られる値が変化するかかもしれないが、その値は(0, 0)または(1, 1)の二通りしかないことに気づくだろう。決して(0, 1)のように値が混ざることはない。これは、左ビットを観測して、もしそれが0ならば、状態 $|00\rangle$ に収束するからである。その後で右ビットを観測するのだから、右ビットも必ず0となる。反対に、左ビットが1と観測されれば $|11\rangle$ に収束し、右ビットも1となる。

エンタングルメント状態のもう一つの例として $|01\rangle + |10\rangle$ の場合も試してみよう。11行目で $|01\rangle + |10\rangle$ を指定している以外は、先の例と同じ操作を行っている。こちらは、(0, 1)か(1, 0)のどちらかかの結果しか現れない。それは左ビットが0と観測されれば $|01\rangle$ に収束し、右ビットは1となる。同様に、左ビットが1と観測されれば $|10\rangle$ に収束し、右ビットは0となるためである。

パウリ X ゲート

パウリ行列 X はビットを反転させる作用があった。この操作は頻繁に使われるため、jsqb ではその簡易版として `x` メソッドも提供している。

```
Qubit = qubit.x ( targetBit );
```

引数の `targetBit` に反転させたいビット番号（右から数え、0 から始まる）を指定し、`x` メソッドをコールすると、該当のビットだけが反転した新しい Qubit オブジェクトが得られる。リスト 2 を `x` メソッドで書き換えると以下のようなになる。

リスト 6 :Pauli X

```
1 <script src="jsqb.js"></script>
2 <script>
3   window.onload = function() {
4
5       debug("X |0> = " + jsqb('|0>').x(0));
6
7       debug("X |1> = " + jsqb('|1>').x(0));
8
9   };
10 </script>
```

実行結果もリスト 2 と同じで、それぞれ反転した値が返される。

```
X |0> = |1>
X |1> = |0>
```

アダマールゲート

アダマール(Hadamard)ゲートは重ね合わせ状態の qubit を作成する場合に、よく用いられる。

```
Qubit = qubit.hadamard ( [ targetBits ] );
```

引数の targetBits に反転させたいビット番号を指定し、hadamard メソッドをコールすると、アダマール変換を行った後の Qubit オブジェクトが得られる。targetBits には複数の番号を配列を用いて指定することができる。例えば、0,1,2 ビット目を変換したい場合は[0, 1, 2] を指定する。また、targetBits の指定を省略すると全ビットに対して変換される。

```
var qubit = qubit.hadamard( [0, 1, 2] ); // apply on bit #0, 1 and 2  
var qubit = qubit.hadamard(); // apply on all bits
```

それでは、独立状態の qubit である $|0\rangle$ や $|1\rangle$ が重ね合わせ状態へ変換される様子をコードでみてみよう。

リスト 7 :Hadamard

```
1 <script src="jsqb.js"></script>  
2 <script>  
3 window.onload = function() {  
4  
5     debug("H |0> = " + jsqb('|0>').hadamard(0));  
6  
7     debug("H |1> = " + jsqb('|1>').hadamard(0));  
8  
9 };  
10 </script>
```

実行結果は以下の通りである。

```
H |0> = 0.7071 |0> +0.7071 |1>  
H |1> = 0.7071 |0> -0.7071 |1>
```

5 行目では、 $|0\rangle$ に hadamard メソッドを実行し、「 $0.7071 |0\rangle + 0.7071 |1\rangle$ 」という結果を得ている。 $|0\rangle$ が $|0\rangle$ と $|1\rangle$ が半々に出現する状態へと変換された。

同様に 7 行目では $|1\rangle$ に対して hadamard 変換を実行している。こちらも「 $0.7071 |0\rangle - 0.7071 |1\rangle$ 」という結果が示す通り、 $|0\rangle$ と $|1\rangle$ の重ね合わせ状態となっている。

Function の実行

jsqb では JavaScript の function を実行する operate メソッドを提供している。これは Function の章で説明した Uf ゲートを実装したものである。

```
Qubit = qubit.operate ( srcBitRange, destBitRange, func );
```

このメソッドの引数には、関数へ渡すビット範囲、関数から出力された値を反映するビット範囲、そしてあなたが作成した JavaScript 関数の 3 つを指定する。ビット範囲とは qubit 内の部分ビットの開始ビット番号と終了ビット番号の組のことであり、jsqb では要素が 2 つの配列で指定する。例えば、右から 3 番目のビットから 5 番目までのビットを計算対象とする場合は、[2,4]と指定する（0 始まりなので、3 番目の index 番号は「2」、5 番目は「4」となる）。1 ビットのみを対象とする場合は、[2,2]と指定してもよいし、単に数値の 2 を指定してもよい。

なお、srcBitRange に指定したビット範囲が関数への入力となり、destBitRange に指定したビット範囲が関数からの出力を受け入れるビットとなる。ただし、srcBitRange の範囲と destBitRange の範囲はオーバーラップしてはならない。

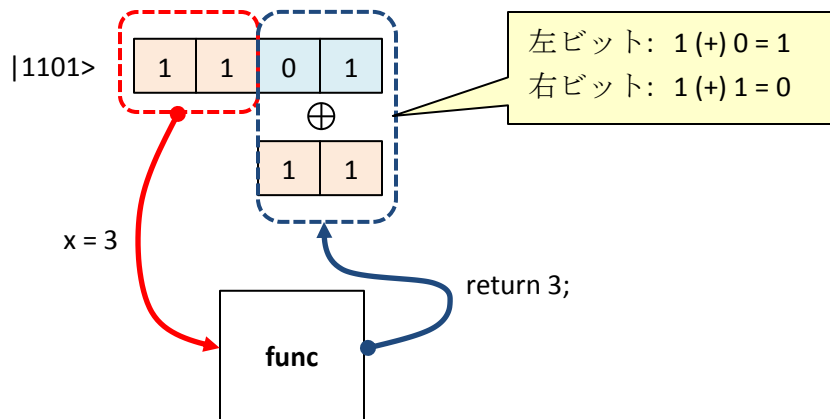
第 3 引数の func に、あなたが作成したカスタム function を指定するが、それは一つの引数を取り、1 つの値を返却する関数でなければならない。例えば、あなたは以下のように引数 x に渡された値をそのまま返す関数 func を作成したとしよう。

```
function func(x) {  
    return x;  
}
```

すると、operate メソッドは srcBitRange 範囲内のビット列を 10 進数に変換し、func 関数に渡す。そして、func 関数からの戻り値を 2 進数に変換し、destBitRange の値とビット単位で加算した結果を設定する。その際、加算は各ビット単位で行い、桁あふれは捨てられる。

例えば|1101>という qubit に対して、operate([2,3], [0,1], func)を実行したとしよう。すると func の引数 x には「11」を 10 進数に変換した「3」が渡される。func 関数は x の値をそのまま返却するので、「3」つまり 2 進数で「11」が 0 ビット目と 1 ビット目に加算される。ここで注意して欲しいのは、ビット毎に独立して加算が行われる点である。つまり、0 ビット目は「1+1=10」となるが、桁あふれした 1 は捨てられ、0 が 0 ビット目に設定される。1 ビット目は「1+0=1」となり、1 が 1 ビット目に設定される。

結果、qubit の値は|1110>になる。「11+01=100」という計算にはならない。



上記の例を含めた関数実行のコードを以下に示す。

リスト 8 :Execute a function

```

1  <script src="../../jsqb.js"></script>
2  <script>
3  function f(x) { return x; }
4
5  window.onload = function() {
6
7      debug("Uf |10> = " + jsqb('|10>').operate(1, 0, f));
8      debug("Uf |11> = " + jsqb('|11>').operate(1, 0, f));
9
10     debug("Uf |1000> = " + jsqb('|1000>').operate([2,3], [0,1], f));
11     debug("Uf |1101> = " + jsqb('|1101>').operate([2,3], [0,1], f));
12
13 };
14 </script>

```

実行結果は以下の通りである。

```

Uf |10> = |11>
Uf |11> = |10>

Uf |1000> = |1010>
Uf |1101> = |1110>

```

いずれの例も 3 行目で定義した関数 **f** をコールしている。関数 **f** は先の例と同じく、引数 **x** をそのまま返却する関数である。

まず、7, 8 行目は 1 ビットの入出力データに対する操作を行っている。

7 行目は 1 ビット目の「1」を受け取り、「1(+)0」を 0 ビット目に設定する。なお、(+)は加算演算後、1 ビット目のみ残すことを意味している。したがって、実行結果は|11>となる。

8 行目も 1 ビット目の「1」を入力として受け取るが、0 ビットとの加算で「1 + 1 = 10」となり、桁あふれが生じる。そのため、あふれた「1」は捨てられ「0」だけが残し、実行結果は|10>となる。

10, 11 行目は 2 ビットの入出力データに対する操作の例で、10 行目は桁あふれが発生しないので、入力ビットがそのまま 2 桁の値となり、|1010>が出力される。

11 行目は本節の例として説明したとおり、0 ビット目で桁あふれが生じ、|1110>が出力される。

5. Deutsch's algorithm

これまでの説明で量子コンピュータ上でのプログラミングの基礎知識とそれを実装したシミュレータの理解は一通りできたものと思う。そこで本章では「問題を解く」というスタイルのコーディングを見てもらいたい。その例として、量子コンピュータ向けのアルゴリズムの一つである Deutsch's algorithm を紹介する。Deutsch は真の量子アルゴリズムを示した最初の人である。Deutsch's algorithm は実用的な仕事で役立つアルゴリズムとは言えないが、非常にシンプルなロジックで量子コンピュータの有効性を示した点で意義がある。また、量子コンピュータ向けのアルゴリズムを学ぶ最初の一歩として最適な教材でもある。

アルゴリズムの概要

Deutsch's algorithm が解決する問題は、以下の通りである。

今、引数 x に 0 または 1 のみを取り、戻り値も 0 または 1 のみを返す関数 $f(x)$ がある。すると、この関数は以下に示す 4 つのいずれかになるはずである。

- | | | |
|-------------------------|---|------------|
| 1. 入力値に関わらず、常に 0 を返す。 | } | Constant 型 |
| 2. 入力値に関わらず、常に 1 を返す。 | | |
| 3. 入力値と常に同じ値を返す。 | } | Balanced 型 |
| 4. 入力値と常に反対の値を返す。 | | |

ここで、最初の二つは常に一定の値を返す点で同種であり、Constant 型として分類できる。それに対し、下から二つは 0 または 1 が返ってくる点で同種であり、Balanced 型と分類しよう。

ここで、あなたへの問題は、与えられた関数 $f(x)$ が Constant 型なのか Balanced 型なのかを判定することである。

JavaScript での実装

まず、古典コンピュータを用いた場合のアルゴリズムを考えてみよう。入力はいずれも 0 または 1 しかないのだから、 $f(0)$ と $f(1)$ を実行してそれらからの戻り値を比較すればよい。同じ値になれば Constant 型であり、異なる値であれば Balanced 型だと判定できる。

この考え方を JavaScript で実装すると以下のようなになる。

リスト 9 : Deutsch's Algorithm implemented with JavaScript

```
1 <script src="jsqb.js"></script>
2 <script>
3   function f0(x) { return 0; }; // Constant 0
4   function f1(x) { return 1; }; // Constant 1
5   function f2(x) { return x; }; // ID (balanced)
6   function f3(x) { return (x + 1) % 2; }; // NOT (balanced)
7
8   function deutsch(f) {
9       return (f(0) + f(1)) % 2;
10  }
11
12  window.onload = function() {
13
14      debug("0:constant, 1:balanced");
15      debug("deutsch(f0) = " + deutsch(f0));
16      debug("deutsch(f1) = " + deutsch(f1));
17      debug("deutsch(f2) = " + deutsch(f2));
18      debug("deutsch(f3) = " + deutsch(f3));
19
20  };
21 </script>
```

そして、実行結果は以下の通りである。

```
0:constant, 1:balanced
deutsch(f0) = 0
deutsch(f1) = 0
deutsch(f2) = 1
deutsch(f3) = 1
```

まず、3 から 6 行目は Constant 型および Balanced 型あわせて四通りの関数を定義している。ここでは f_0 , f_1 が Constant 型であり、 f_2 , f_3 が Balanced 型である。

そして、8 から 10 行目で問題を解決する関数 `deutsch` が定義されている。`f(0)` と `f(1)` を計算し、その結果を足して 2 で割った余りを求めている。もし関数 `f` が **Constant** 型だとすると、

`0 + 0 = 0` または、

`(1 + 1) % 2 = 0`

となり、いずれの場合も 0 が求まる。一方もし **Balanced** 型であれば

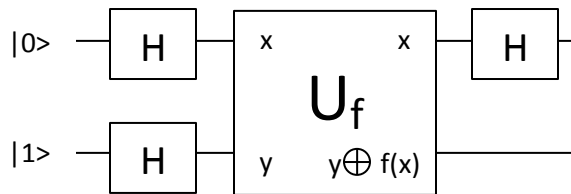
`0 + 1 = 1` または、

`1 + 0 = 1`

となり、いずれの場合も 1 が求まる。実行結果もその通りの値を出力している。
このように古典コンピュータでは関数 `f` を **2 回** 実行することで、どちらの型かを判別することができた。

量子コンピュータでの実装

Deutsch はこの問題を解決するため、以下のゲートを考え出した。



Uf ゲートの入力と出力にアダマールゲートを通しただけの単純な回路であることが分かるだろう。そして、入力値として $|01\rangle$ を与えれば、出力の 1 ビット目が **Constant** 型か **Balanced** 方かを示すというのである。

上記回路を実装したのが以下のコードである。

リスト 10 : Deutsch's Algorithm

```
1 <script src="jsqb.js"></script>
2 <script>
3   function f0(x) { return 0; }; // Constant 0
4   function f1(x) { return 1; }; // Constant 1
5   function f2(x) { return x; }; // ID (balanced)
6   function f3(x) { return (x + 1) % 2; }; // NOT (balanced)
7
8   function deutsch(f) {
9     return jsqb('|01>')
10      .hadamard()
11      .operate(1, 0, f)
12      .hadamard()
13      .measure(1);
14 }
15
16 window.onload = function() {
17
18   debug("0:constant, 1:balanced");
19   debug("deutsch(f0) = " + deutsch(f0));
20   debug("deutsch(f1) = " + deutsch(f1));
21   debug("deutsch(f2) = " + deutsch(f2));
22   debug("deutsch(f3) = " + deutsch(f3));
23
24 };
```

25 </script>

このコードはリスト 9 (JavaScript で実装) とほとんど同じである。異なるのは関数 `deutsch` の中味だけである。そこでは、以下の操作を順に行っているだけである。

- Step 1: $|01\rangle$ の qubit オブジェクトを生成する。
- Step 2: hadamard ゲートに通し、重ね合わせ状態を作り出す。
- Step 3: 関数 `f` を実行する。
- Step 4: 再度、hadamard ゲートに通す。
- Step 5: 1 ビット目を観測し、答えを得る。

机上で上記操作による状態変化を追うのは大変なので、次のようなプログラムを作成し、ステップ毎の状態を画面表示させてみよう。ここでは関数 `f3` (引数 `x` を反転した値を返却) を実行している。

リスト 11 : Deutsch's Algorithm – Process of calculation

```
1 <script src="jsqb.js"></script>
2 <script>
3   function f0(x) { return 0; }; // Constant 0
4   function f1(x) { return 1; }; // Constant 1
5   function f2(x) { return x; }; // ID (balanced)
6   function f3(x) { return (x + 1) % 2; }; // NOT (balanced)
7
8   window.onload = function() {
9
10      var qubit = jsqb('|01>');
11      debug("Step 1: |01> = " + qubit);
12
13      qubit.hadamard();
14      debug("Step 2: H|01> = " + qubit);
15
16      qubit.operate(1, 0, f3);
17      debug("Step 3: Uf(H|01>) = " + qubit);
18
19      qubit.hadamard();
20      debug("Step 4: H(Uf(H|01>)) = " + qubit);
21
22      debug("Step 5: Measure qubit-#1 = " + qubit.measure(1));
23
24   };
25 </script>
```


このコードの実行結果は以下のようになる。

```

Step 1: |01> = |01>
Step 2: H|01> = 0.5 |00> -0.5 |01> +0.5 |10> -0.5 |11>
Step 3: Uf(H|01>) = -0.5 |00> +0.5 |01> +0.5 |10> -0.5 |11>
Step 4: H(Uf(H|01>)) = -1 |11>
Step 5: Measure qubit-#1 = 1

```

まず、Step 1 では|01>を初期状態として持つ qubit を作成した。今、この qubit オブジェクトには|01>という独立状態の basis state が一つ保持されている。

そして、Step 2 ではアダマール(Hadamard)ゲートにより、|00>, |01>, |10>, |11>の 4 つが重ね合わさった状態にしている。具体的な操作を数式で表すと以下のようになる。まず、左ビット|0>に対して行列 H を掛ける。すると、|0>と|1>の重ね合わせ状態に変化する。

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1*1 + 1*0 \\ 1*1 - 1*0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \left(\begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right) = \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle)$$

同様に右ビットに対しても行列 H を掛けると、|0>と|1>の重ね合わせ状態に変化する。上記とは符号が異なる点に注意されたい。

$$H|1\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1*0 + 1*1 \\ 1*0 - 1*1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \frac{1}{\sqrt{2}} \left(\begin{pmatrix} 1 \\ 0 \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right) = \frac{1}{\sqrt{2}} (|0\rangle - |1\rangle)$$

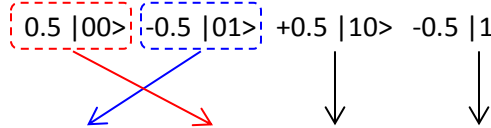
これらは 2 つのビットはテンソル積でつながり合っているため、テンソル積をとると、

$$\begin{aligned}
&= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\
&= \left(\frac{1}{\sqrt{2}} * \frac{1}{\sqrt{2}}\right) ((|0\rangle + |1\rangle) \otimes (|0\rangle - |1\rangle)) \\
&= \frac{1}{2} (|0\rangle \otimes |0\rangle - |0\rangle \otimes |1\rangle + |1\rangle \otimes |0\rangle - |1\rangle \otimes |1\rangle) \\
&= \frac{1}{2} (|00\rangle - |01\rangle + |10\rangle - |11\rangle) \\
&= \frac{1}{2}|00\rangle - \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle - \frac{1}{2}|11\rangle
\end{aligned}$$

となり、Step 2 の実行結果と一致する。これで 2-qubit が取り得るすべての状態 (00, 01, 10, 11 の 4 種類の値) を作り出せ、関数を実行するための前準備が完了した。

次の Step 3 では U_f ゲートを使い、先の 4 つの値に対して関数を一気に実行している。以下のように計算前と後で項の位置が入れ替わっているため、混乱しやすいので注意して欲しい。

実行前: $0.5 |00\rangle - 0.5 |01\rangle + 0.5 |10\rangle - 0.5 |11\rangle$



実行後: $-0.5 |00\rangle + 0.5 |01\rangle + 0.5 |10\rangle - 0.5 |11\rangle$

左の basis state (項) からひとつひとつ計算過程を示しておこう。まず、最初の項「 $0.5|00\rangle$ 」の 1 ビット目「0」を関数 f_3 の引数として渡して実行すると反転値「1」返ってくる。それを 0 ビット目の「0」と加算して結果を得るので、計算後は「 $0.5|01\rangle$ 」となる。

同様に二番目の項「 $-0.5|01\rangle$ 」の 1 ビット目「0」を関数 f_3 の引数として渡して実行すると反転値「1」返ってくる。それを 0 ビット目の「1」と加算すると「10」となるが、ここは桁あふれした 1 は捨てるので、結果は「 $-0.5|00\rangle$ 」となる。

三番目と四番目は 1 ビット目が「1」なので反転すると「0」となり、0 ビット目には何も影響を与えない。つまり、計算後も計算前と同じ値となる。

Step 4 からは、計算結果を取り出すための操作となる。超並列計算を行うために重ね合わせの状態を作ったが、このままではどの値が観測されるかは確率的に決定されるため、常に決まった値が得られない。何らかの方法で値を独立状態に戻す必要がある。その方法の一つとして量子フーリエ変換があるが、2-qubit の場合はアダマール変換と等価であることが知られている。したがって、ここでは Step 2 同様に、再度 hadamard ゲートを通せばよい。

Step 2 で行列 H と $|01\rangle$ との積の計算過程を示したように、

$$H|01\rangle = \frac{1}{2}|00\rangle - \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle - \frac{1}{2}|11\rangle$$

という計算結果を得た。同様に他の $|00\rangle$, $|10\rangle$, $|11\rangle$ に対しても行列 H との積をとると、

$$H|00\rangle = \frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle$$

$$H|10\rangle = \frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle - \frac{1}{2}|10\rangle - \frac{1}{2}|11\rangle$$

$$H|11\rangle = \frac{1}{2}|00\rangle - \frac{1}{2}|01\rangle - \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle$$

よなる。以上を Step 3 で得た計算結果に H との積をとった以下の式に代入してみよう。

$$\begin{aligned} & H\left(-\frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle - \frac{1}{2}|11\rangle\right) \\ &= -\frac{1}{2}H|00\rangle + \frac{1}{2}H|01\rangle + \frac{1}{2}H|10\rangle - \frac{1}{2}H|11\rangle \\ &= -\frac{1}{4}|00\rangle - \frac{1}{4}|01\rangle - \frac{1}{4}|10\rangle - \frac{1}{4}|11\rangle \\ &\quad + \frac{1}{4}|00\rangle - \frac{1}{4}|01\rangle + \frac{1}{4}|10\rangle - \frac{1}{4}|11\rangle \\ &\quad + \frac{1}{4}|00\rangle + \frac{1}{4}|01\rangle - \frac{1}{4}|10\rangle - \frac{1}{4}|11\rangle \\ &\quad - \frac{1}{4}|00\rangle + \frac{1}{4}|01\rangle + \frac{1}{4}|10\rangle - \frac{1}{4}|11\rangle \\ &= -|11\rangle \end{aligned}$$

計算途中、16 項のまで増えたがその殆どが打ち消され、 $|11\rangle$ だけが残った。

この状態で Step 5 の 1 ビット目の観測を行うと、常に「1」という値を得る。この 1 は実行した関数が **Balanced** 型であることを示している。確かに今回は関数 **f3** を実行したので期待通りの結果を得た。

ここで重要なのは、答えを得るのに U_f ゲートによる関数実行は 1 回しか行われていない点である。Step 3 の説明では 00, 01, 10, 11 の 4 つの値にそれぞれ関数を実行したので 4 回の関数実行が行われるかのように思われたかもしれない。確かに JavaScript シミュレータ内では 4 回の関数実行を行っている。しかし、本物の量子コンピュータ上で実行すれば 1 回のゲート操作で 4 つの **basis state** に対する計算を行ってしまうのである。つまり、関数 **f** は 1 回しか実行されない。

JavaScript 版では 2 回の実行が必要であったが、それが 1 回で済むのだ。今回の例のように非常に単純なロジックであればその差は微々たるものであるが、関数 **f** が入力データ数に対し、指数関数的に組み合わせが膨らむ問題の場合、非常に大きな効果を示すのである。

6. おわりに

本稿では量子プログラミングを行うために習得しておくべき前提知識と JavaScript シミュレータ上での基本的なコーディング方法について説明した。また、最後に最も単純な量子アルゴリズムの一つである **Deutsch's algorithm** を紹介し、より実践的なプログラミングコードにもふれた。しかしながら、それだけではまだ不十分な点が多い。例えば、本稿では量子ゲートとして、パウリ X ゲートとアダマールゲートの 2 だけを紹介したが、その他にもプログラミングする上で必要となるゲートはいくつもあるし、コードの動きを真に理解するには量子力学の知識も不可欠となる。

本稿では、量子力学を学んだことの無い方を対象としているため、そうした難解な点はあえて避けてきた。しかし、今後、あなたがより実用的なプログラムを作成しようとコーディングを始めたなら、たちどころに未開の問題にぶつかってしまうだろう。まだ誰も解いていない、もしくはネットを検索しただけでは容易に解を得られない問題である。量子コンピュータ用にもすでに高級言語が用意されている段階であればそのような問題に遭遇することは稀だが、本稿で紹介したコードを見ても分かるとおり、今はまだアセンブリレベルの低レベルな操作を行っている段階である。また、十分な解説書も無い。そうした状況下ではコーディングのルールを作り出している基礎数理モデルを正しく理解し、計算過程を追える力がなければならない。

本稿を読破し、サンプルコードを理解できたなら、もう十分に次へのステップへ進む準備ができたはずである。是非、量子力学の基礎も学び、量子プログラミングの真の理解を目指して欲しい。