GSLetterNeo vol.146

2020 年 9 月

シミュレーションで始める
量子コンピューティング (3)

熊澤 努 kumazawa @ sra.co.jp

はじめに

Vol. 142、144 では量子アニーリング方式の量子コンピューティングを体験しました。量子アニーリング方式は「ハミルトニアンが最小になるような変数の値を計算する」量子コンピュータです。量子コンピュータを使う側からすると、変数とハミルトニアンを作って量子コンピュータ（本連載ではシミュレータ）に渡すだけで、答えが自動的に計算されるので、便利です。

今回は、量子アニーリング方式とは異なる方式である量子ゲート方式を体験します。普通のコンピュータは、ビット（2 進数）の計算である論理ゲート（AND や OR など）を組合せて複雑な計算を実現します。一方、量子ゲート方式は、量子ビットの変換操作を表す量子ゲートをプログラマ自身が組合せて計算を実現します。

Microsoft 社の Q#言語とその開発キット QDK を使います[1]。Q#は量子コンピュータで動かすアルゴリズムを記述するためのプログラミング言語です。Q#で書いたプログラムだけを動かすこともできますし、.NET 言語（C#や F#）あるいは Python のプログラムと連携することも可能です。普通のコンピュータで動かす部分は C#や Python でプログラムを書き、量子コンピュータを使うところは Q#でプログラムを書く、という使い分けをするようになっています。開発キット QDK には、Q#用のライブラリなどの他に、量子コンピュータの

[1] https://docs.microsoft.com/ja-jp/quantum/overview/what-is-qsharp-and-qdk

シミュレータが含まれています。今回は、Visual Studio Code 上で Python と連携し、シミュレータによりプログラムを動かす方法を説明します。

Q#の準備

最初に Q#を使うための準備をします[2]。まず、Python3.6 またはそれ以降のバージョンを準備してください。本記事は Windows 10 で Anaconda Navigator を使用した場合を想定しています。
次に.NET Core SDK をインストールします。以下のサイトにアクセスしてダウンロードしてください。本記事の執筆時点（2020 年 8 月）では.NET Core SDK 3.1 が必要です。

https://dotnet.microsoft.com/download

Q#のプログラムを Python から呼び出すために、qsharp パッケージを Python にインストールします。pip を使える場合、Python のコマンドライン環境で以下のコマンドを実行してください。

```
pip install qsharp
```

Q#を動かすために IQ#をインストールします。Windows のコマンドプロンプトを起動して、次のコマンドを実行します。

```
dotnet tool install -g Microsoft.Quantum.IQSharp
dotnet iqsharp install
```

ここからは開発・実行環境の整備です。開発環境として手軽に用意できるのは、エディタに Visual Studio Code を使う方法でしょう。Visual Studio Code をインストールします。

https://azure.microsoft.com/ja-jp/products/visual-studio-code/

[2] https://docs.microsoft.com/ja-jp/quantum/quickstarts/install-python?tabs=tabid-dotnetcli#install-the-qsharp-python-package にイントール手順の詳細が記載されています。

Visual Studio Code で Q# を動かすには、拡張機能である「Microsoft Quantum Development Kit for Visual Studio Code が」必要です。Visual Studio Code の拡張機能のリストから検索するか、下記のサイトからダウンロードしてインストールしてください。

https://marketplace.visualstudio.com/items?itemName=quantum.quantum-devkit-vscode

必要なら、Python 用拡張機能もインストールしておきましょう。

Q#プログラムを動かしてみる

次にサンプルプログラムを動かして動作を確認しましょう。
次の Q#のプログラムを作り、適当な名前（例えば Program.qs）のファイルで保存してください。拡張子は.qs とします。

```
namespace QSample {
    open Microsoft.Quantum.Intrinsic;

    operation OddEven() : (Result, String) {
        using (q = Qubit())  {
            H(q);
            let b = M(q);
            Reset(q);

            mutable result = "Even";
            if (b == One) {
                set result = "Odd";
            }
            return (b, result);
        }
    }
}
```

Q#のプログラムを動かす Python のサンプルプログラムを作ります。適当な名前（例えば、sample.py）で保存します。保存する場所は Program.qs と同じフォルダにしてください。

```
import qsharp
from QSample import OddEven

print(OddEven.simulate())
```

sample.py を実行します。sample.py を保存したフォルダに移動して、以下のコマンドを実行します。Visual Studio Code の Python 拡張機能を使っている場合には、エディタ上で実行できます。Program.qs のコンパイルは不要です。

```
python sample.py
```

以下のような結果が表示されることを確認してください。なお、プログラムの実行結果は、最終行の(0, 'Even')です。

```
Microsoft.Quantum.IQSharp...省略

Preparing Q# environment...
(0, 'Even')
```

Q#プログラムを読む

sample.py を何度も実行するとわかりますが、サンプルプログラムは(0, ‘Even’)あるいは(1, ‘Odd’)のどちらかをランダムに出力します。Q#について理解をするために、Program.qs を読み解いていきましょう。
1 行目は名前空間 QSample の定義です。名前空間は C#でも出てきますが、モジュールを QSample という名前でまとめて扱う機能です。

```
namespace QSample {...}
```

名前空間 QSample は、sample.py の 2 行目で Q#プログラムを呼び出すために使っています。

```
from QSample import OddEven
```

Program.qs の 2 行目は、プログラムが利用しているライブラリをオープンします。ここでは、量子ビットの測定や操作のためのライブラリである Microsoft.Quantum.Intrinsic ライブラリを使用します。

```
open Microsoft.Quantum.Intrinsic;
```

量子ビットを操作するプログラムはオペレーション（操作）に書きます。オペレーションの先頭は予約語 operation で始めます。次の OddEven はオペレーションの名前です。この名前は自由に決めることができます。コロン（:）の後には戻り値の型を書きます。ここでは、(Result, String)が戻り値の型で、量子ビットの観測結果の型（Result）と文字列型（String）の 2 項組（ペア）です。Q#には他にも整数型(Int)、浮動小数点型（Double）真理値型（Bool）などの型が用意されています[3]。オペレーションの本体は中かっこ内に書きます。

```
operation OddEven() : (Result, String) {...}
```

量子ビットを使うには、最初に変数に量子ビットを割り当てる必要があります。下のように using を使用すると、変数 q に量子ビットを割り当てます。有効範囲は using の次の中かっこ内に限られます。

```
using (q = Qubit())  {...}
```

量子ビットに対する操作は、演算→観測→解放の手順で行います。この手順は普通のコンピュータでの変数の扱い方と異なるので、サンプルプログラムを一つずつ見ていきます。
演算については、サンプルプログラムでは、量子ビットにアダマール変換を適用しています。

```
H(q);
```

H はアダマール変換で、Microsoft.Quantum.Intrinsic ライブラリに定義されています。アダマール変換によって、変数 q は Zero と One のどちらかを 50%の確率でとるようになります。このように、量子コンピューティングには確率的に値が決まるという特徴があります。また、このことは、変数 q が Zero と One の値を同時に持った状態であるとみなすことができ、「量子重ね合わせ」といわれています。
量子ビットが Zero と One のどちらの値になるかを知るには、観測を行わなければなりません。観測には、Microsoft.Quantum.Intrinsic ライブラリに定義されている M オペレーショ

```
let b = M(q);
```

[3] 詳しくは、 https://docs.microsoft.com/ja-jp/quantum/user-guide/language/types を参照してください。

ンを使います。サンプルプログラムでは、変数 q に割り当てた量子ビットを観測した結果を変数 b に格納しています。let は変数定義を表す Q#の予約語です。b は Result 型の変数で、観測した結果定まった量子ビットの値 Zero または One のいずれかの値をとります。整数型の数値 0 や 1 と間違えないように注意してください。
観測を行ったら、量子ビットの割り当てを解放します。Microsoft.Quantum.Intrinsic ライブラリ Reset により解放に必要な処理を実行します。

```
Reset(q);
```

続いて、呼び出し元に返す文字列を、観測した量子ビットの値に応じて決定します。

```
mutable result = "Even";
if (b == One) {
    set result = "Odd";
}
```

文字列は変数 result に格納します。result の定義には mutable という予約語がついています。mutable は値の上書きを許す変数として定義することを意味しています。実際、次の行の if 文で、量子ビットの値が One のときには“Odd”に書き換えています。一方、上で出てきた let は書き換えを許さない変数の定義に使います。
Python プログラムから OddEven オペレーションを呼び出すには、qsharp パッケージと QSample をインポートしたうえで、オペレーションのシミュレーションを実行します。「オペレーション名.simulate()」メソッドを呼ぶことで、シミュレーションを開始します。

```
import qsharp
from QSample import OddEven

print(OddEven.simulate())
```

おわりに

本稿では、Microsoft 社が公開している量子ゲート方式の量子コンピュータ向けプログラミング言語 Q#を紹介しました。Q#は、量子ビットに適用する操作を、プログラマ自身が組み合せて書くことができます。色々な操作がライブラリとして提供されているので、使ってみてください。Q#についての詳細は、以下の公式サイトのユーザーガイドを参照してください。

https://docs.microsoft.com/ja-jp/quantum/user-guide/

GSLetterNeo Vol.146
2020 年 9 月 20 日発行
発行者　株式会社 SRA 先端技術研究所

編集者　土屋正人
バックナンバー http://www.sra.co.jp/gsletter
お問い合わせ　gsneo@sra.co.jp

〒171-8513 東京都豊島区南池袋 2-32-8