# A First Step to Quantum Computation with Qiskit

**Ryoichi Kawai**

Department of Physics

University of Alabama at Birmingham

mailto:kawai@uab.edu

https://kawaihome.link

**Warning**

I started writing this book in April, 2022. I hope this book will be ready for the course in Fall 2022 or at latest, Spring 2023. At the present, even the structure of book has not been fixed. You are free to read it but don't give me any comment nor ask me any question until the first full draft is completed.

Thank you for your patience!

Ryoichi Kawai

# 1. Preface

This book is written for undergraduate physics students in junior or senior year. Many students take quantum mechanics courses in senior years. On the other hand, many contempolary topics in physics involve quantum mechanics. If quantum mechanics is a prerequisite, many students have no chance to take the courses. Furthermore, the materials covered in traditional quantum mechanics courses are not sufficient to learn modern physics such as condensed matter physics and particle physics. It is a big challenge to expose the students to modern physics involving advanced topics in quantum mechanics. Quantum information theory and quantum computation suffers greatly from these issues. Most of physics students are not prepared to take courses on those topics.

Not only physics students are interested in quantum information theory and quantum computing. Students in mathematics, computer science, eletric engineering, data science, …. are eager to learn them. Obviously, most of them do not have experiences in quantum mechanics at all. Therefore, it is necessary to teach quantum mechanics along with quantum information and quantum computing. However, there is also time limit. Idealy, every thing must fit to a semester-long course. We cannot spend too much time on quantum mechanics.

Furthremore, it is desired to use a computer programming language to exercise quantum computation. In particular, python is apparently the preferred language in quantum computing. Not all students have epxerience in python coding.

At the end, I found myself teaching quantum mechanics, mathematics, and programming language along with quatum compuation. The contents of main topics, quantum computation, was necessarily squeezed into a very limited time.

Then, I found Jupyter Book. If I write lecture notes in Jupyter Book, students can self-teach *prerequisites* through interactive online lecture notes. So, my course is entirely written in Jupyter Book. A problem is solved.

Since I do not assume prior experience in quamtum mechanics nor python language, the course can be taken by students outside of physics. I hope I can expose students in many different fields to quantum computation.

Another problem is what topics in quantum computation should be covered. Despite that the quantum computation is rather young topics, it has made huge progresses and our knowledge still expanding rapidly. Most of introductory level textbooks explain mostly famous alfgorithms such as the Shor's algorithm, Grover's algorithm, and Fourier Transforms. However, I found very littel information about how to make your own algorithms. Perhaps we can talk about simpler problems and solve them using quantum computers for the sake of learning how quantum gates work. So, I put off discussion on *quantum supremacy* until students become familiar with quantum operations. Due to the lack of my experience, I could not find small but interesting problems. Nevertheless, you will find a few quantum computation with only a singe or two qubits. I hope they help students gain aptite to go further into quantum computation.

*Ryoichi Kawai, July 30, 2022*

# 2. Preparation

This book is entirely written either in the Jupyter Notebook or Markdown. The online version is generated by Jupyter Book. The first section of this chapter explains how the interactive book works. In the rest of this chapter, software used in this book and online tools are introduced.

We will use computer software, *Python*, *JupyterLab*, and *Qiskit*. In addition, we will use online tools and real quantum computers provided by *IBM Quantum Experience*. No previous experience is needed. However, you must train yourself as you read this book.

The use of Python, JupyterLab, Qiskit, and IBM Quantum Experience is absolutely required. If you do not have your own computer or the capacity of your computer is limited, free cloud-based online services, such as *IBM Quantum Lab* or *Google Colab* satisfies the requirement. Apart from some inconvenience, they work quite well. However, it will be convenient to run codes on your computer. Therefore, it is highly recommended to install required software on your computer. Installation instruction is given in the following sections.

## 2.1. About This Book

Each page of this book is a Jupyter Notebook file containing executable python codes. The pages are written with JupyterLab and converted to web pages by Jupyter Book. This section explains how to use interactive feature of the book.

### 2.1.1. Layout

You are presumably reading this book now. You see three columns. The left column contains the table of contents. You can jump to any chapter and section at any time. Above the table of contents, there is a search box. You can search a word through the entire book.

On the right column, you see the contents of the current page. You can jump within the page by clicking subsection title.

The center column is the actual content of the page. At the upper-right corner, there are four icons. The role of these icons is explained in next subsection.
At the bottom, you see navigation arrows one to the previous page and the other to the next page. Footnotes are also shown at the bottom.

The main document consists of two types of blocks. One is text block and the other is code block. A code block contains a python code and the output of the code appears below the code block You can actually run the code on your computer or at Google Colab as explained below.

There is a square icon at the top of the page. It toggles between full screen and window modes.

### 2.1.2. Linked references

There are many references in the documents, references to equation, figure, table, chapter, section, footnote, bibliography citation, … They are all linked to the actual target. If you click the reference, the page jump to the target. For example, an equation number cited in a page is linked to the corresponding equation even it is in a different page. Clicking the cited equation number, the page containing the equation is shown External URLs are also linked.

### 2.1.3. Downloading the page

There is a download button at the upper-right corner. You can download the original Jupyter notebook file (.ipynb) of the current page and the PDF version. Click one of them with the left mouse button, download is supposed to start. However, some web browser instead shows the contents in the browser. If your browser does that, click ".ipynb" with the *right* mouse button and use "save link as …". You can run the downloaded Jupyter notebook file on you computer using JupyterLab (See Section 2.3). You can also download any page as PDF in the same way. If you prefer to read it on paper, print the PDF file.

### 2.1.4. Github

The github button (octocat) is for the Github connection. It contains two items, *repository* and *open issue*. You can view the source files of this book in the github repository and write bug reports and comments which can be viewed at the githut repository.

### 2.1.5. Google Colab

The rocket icon at the top is known launch button. Currently, there is only one item "Colab" in it. This book is linked to Google Colab. By clicking "Colab", you open the current page in Google Colab where you can actually edit and run the page. For example, there is an python code in the current page, you can run it on the google computer. Furthermore, you can modify the code and run your own version. See Section Google Colab for more information about Google Colab.

## 2.2. Python

`Python` is a popular computer programming language. It is widely used in many different fields, scientific research, IT companies, financial institutions, $\cdots$ . The use of python is required in this short course. I hope that readers have a little bit of experience in coding with python. If you haven't use it yet, you need to learn it a long with this course. There are many online tutorials for self-studying. Some of popular ones are listed at sunscrapers.com. You can pick one that matches to your level. The following book will be useful in general.

> Allen B. Downey: *Think Python - How to think like a computer scientist* (O'Reilly Media, 2016) or (Green Tea Press, 2012)

which is freely available at greenteapress.com. If you prefer a hard copy, You can buy a paperback at amazon.com.

You must be able to execute python codes on a computer. A computer provided by your school may have python on it. You can also use online services such as Google Colab where you can run python codes for free. However, school computers or online services often do not have modules you need. It is highly recommended to have python on your own computer. It is completely free but takes up some disk space (at least 5Gb). In addition to basic installation, you need to install additional modules for quantum computing. In this chapter, I explain how to install standard python. Two additional modules, `jupyter` and `qiskit` will be introduced in the following sections.

### 2.2.1. Installation [1]

There are many ways to install python. One problem is that basic python (we shall call it *python platform*) is not enough to do almost anything. For example, the python platform does not provide any scientific function. We need to install many modules along with the python platform. Fortunately, most of modules we need are managed by a package manager called *conda*. Unless you are expert of python, I strongly recommend to use *conda*.

1. **Downloading anaconda package**
   Go to anaconda.com and download the installer for your OS . Linux, Microsoft Windows, and Mac OS are all supported.
2. **Run the installer**
   Execute the installer. If the installer asks if it is for single user or multiple users, chose single user. It also asks a location to install. The default location is fine.
3. **Launching anaconda**

The way to launch anaconda depends on the OS.

- *Linux*
  Anaconda configuration is written in `.bashrc`. When you open a new terminal, anaconda is ready.
- *Windows*
  Open the Start Menu and pull down Anaconda3 menu. Click “Anaconda Prompt” or “Anaconda Powershell Prompt”, whichever you like. Anaconda works only inside the Anaconda terminal window. You see another link in the menu: Anaconda Navigator. You can launch major applications from the navigator including JupyterLab. Since it must configure environment for each applications, the navigate starts slowly.
- *Mac OS*
  To be written.

1. **Updating the package**
   There may be newer version of packages. To update installed packages, run the following command inside the terminal window:

```
conda update -all
```

Now the basic installation is completed. If you are interested in what packages are installed, use the following command.

```
conda list
```

You will see hundreds of packages. We will be using many of them even without knowing it.

### 2.2.2. Using Python

You can use python interactively inside the text window. Just use command `python`.

```
python
Python 3.9.12 (main, Apr  4 2022, 05:22:27) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

“>>>” is python prompt. You enter a single line code here. For example

```
>>> 1+2
3
>>>
```

To exit from the python interactive mode, use the following exit function:

```
>>> exit()
```

We rarely use the interactive mode. Instead, we write a source code using a text editor and run it. There are also so-called integrated development environment (IDE). `spyder` and `jupyter` are the two most popular IDEs for python. We use`jupyter-lab` to develop python codes. Actually, this textbook is entirely written with `jupyter-lab`.

### 2.2.3. Numpy

Unlike other computer languages such as C++, Python platform is not capable of mathematical calculation. Mathematical modules are provided by various packages. Among them, *Numpy* is commonly used. Free documentation and tutorial are available at numpy.org.

To use it, we must *import* the module to your code. In this book, we import it as

```
import numpy as np
```

where `np` is a shorthand of `numpy`. You can access methods (functions) and attributes associated with the numpy class using the shorthand. For example, we calculate $\cos(\pi)$ as

```
import numpy as np

np.cos(np.pi)
```

```
-1.0
```

Notice `np.` attached to `cos()` and `pi`. We must tell python platform that the mathematical expressions are defined in numpy.

**Running python codes on your computer**

You can copy the python code to the clipboard on your computer. Move the cursor to the top-right corner of the code block. Copy icon shows up. Just click it. The content of the code block is copied. You can paste it to your jupyter-lab (or a python source file) and run it.

**Running python codes at Google Colab**

Launch Google Colab from the launch button at the top of the page. This page is automatically transferred to Colab. Sign in to Colab if you haven't yet. Now, you can run the python code by clicking a little triangle button at the upper-left corner of the code block. You can modify the code and run your own version.

**Exercise** 2.1   Calculate $\sin\left(\frac{\pi}{2}\right)$ using python.

### 2.2.4. Matplotlib

When we want to show results of computation as graphs, we use a plotting module `matplotlib`. It is a powerful visualization package for scientific research. Free documentation and tutorials are available at matplotlib.org.

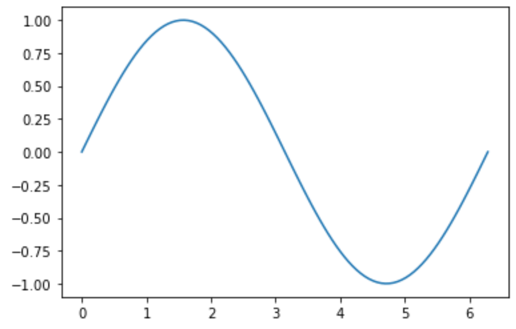
In this book we import it as

```
import matplotlib.pyplot as plt
```

where `plt` is a shorthand. The following example plots $\sin(x)$ for $x \in [0, 2\pi]$

```
import matplotlib.pyplot as plt   # visualization tools
import numpy as np  # numerical math package
x=np.linspace(0,2*np.pi,100)  # generates 100 points betwee 0 and 2pi
plt.plot(x,np.sin(x))
```

```
[<matplotlib.lines.Line2D at 0x7f4a78ee7b80>]
```



**Exercise** 2.2 Calculate $\cos(x)$ for $x \in [-\pi, +\pi]$ using python.

Lat modified on June 4, 2022.

[1] If your computer does not have enough storage or power to run Python and JupyterLab, skip Installation. Use an online IDE introduced in Section 2.5.

## 2.3. JupyterLab

Jupyter is a project to develop an interactive environment for data science and scientific computing. (See jupyter.org.) It provides a popular web-based interactive development environment (IDE), *Jupyter Notebook*. We use a next-generation of notebook called *JupyterLab* in this book. With JupyterLab, you can create an interactive document consisting of texts written in *Markdown* and Python codes. In fact, this book is entirely written in JupyterLab. Since it runs in a web browser, you can use JupyterLab running on remote computers. There are free online services that allow anyone to use their JupyterLab. However, it is convenient to have it on your own computer since we use some extensions which may not be available at the free services.

### 2.3.1. Installation [1]

Anaconda install JupyterLab by default. However, certain extensions used in this book are not automatically installed. To install them, open an Anaconda terminal window and install them as shown below.

- nbextensions

```
pip install jupyter-contrib-nbextensions
jupyter contrib nbextension install --user
```

- spellchecker

```
pip install jupyterlab-spellchecker
```

- mathjax

```
pip install jupyter-serever-mathjax
pip install jupyterlab-mathjax3
```

### 2.3.2. Launching

There are several different ways to launch JupyterLab depending on the OS. On any OS, you can launch it from the Anaconda terminal:

```
jupyter-lab
```

This opens an default web browser and JupyterLab appears in it. If the web browser is already opened, it appears in a new tab.

On MS Windows, you can start Anaconda Navigator from Start Menu. It takes a minute to launch the navigator. Just click JupyterLab shown inside the navigator. Anaconda does not provide an icon to launch JupyterLab but you find one for Jupyter Notebook in Start Menu. You can copy it to desktop and modify it. Simply replace notebook with lab in the target.

On Linux, you can create a desktop application file easily. The actual name of executable is jupyter-lab and it is located in ~/anaconda3/bin. Depending on the Linux distribution, notebook files with ".ipynb" extension is associated with jupyter-lab. If you click a notebook file in a file manager, jupyter-lab is automatically launched. (It seems difficult to do it on MS Windows.)

### 2.3.3. Usage

JupyterLab create a notebook consisting of three types of blocks, Markdown block, input code block, and output code block. In a Markdown block, you write texts, tables, images, and equations using Markdown. Markdown is used in many other applications such as wikipedia and github. Most of formatting methods used in github also work in JupyterLab. See writing-on-github for the formatting syntax.

In a code block, you write python codes. They are computed right a way. Right now, I am in Markdown block. The next block is an input code block. You see a python code in it and its output appears in output block right below the input code block.

```
x=2
y=3
print("x^2+y^2=",x**2+y**2)
```

```
x^2+y^2= 13
```

### 2.3.4. Mathematical expression

For us, ability to write mathematical expression is important. Here are examples. You can write mathematical equation using LaTeX. An in-line equation start with `$` and another `$` after the equation. For example,

```
$ \int \cos x dx = \sin x$
```

is rendered as $\int \cos x dx = \sin x$.

A display equation begins with `$$` which start a equation environment. In a new line, you write an equation. Then, another `$$` closes the environment. For example,

```
$$
\int_0^\infty e^{-x} dx = 1
$$
```

becomes

$$\int_0^\infty e^{-x} dx = 1$$

### 2.3.5. Closing JupyterLab

Just closing web browser does not stop JupyterLab. It is still running in the background. To close JupyterLab, use "shutdown" in "File".

---

[1] If your computer does not have enough storage or power to run Python and JupyterLab, skip Installation. Use an online IDE introduced in Section 2.5.

## 2.4. Qiskit

Qiskit is an open-source software development kit (SDK) for quantum computation. It runs inside Python platform.

QIskit provides a large set of tools for

1. developing new quantum algorithms and exploring new idea
2. constructing a quantum circuit and testing it by running simulation on a classical computer
3. executing the circuit on a real quantum computer through IBM Quantum Experience.

We will use QIskit for all these three important coding steps.

You can find useful information about Qiskit including tutorials and API documentation at [qiskit.org](qiskit.org).

### 2.4.1. Installation

It is a set of python libraries but not included in Anaconda. We need to install them manually.

```
pip install qiskit
pip install qiskit[visualization]
```

Since conda does not manage these packages, you must update the package when a new version becomes available. To check the current version, run the following command in the anaconda terminal window.

On MS Windows, use Anaconda Powershell Prompt.

```
pip list | select-string "qiskit"

qiskit                            0.36.2
qiskit-aer                        0.10.4
qiskit-ibmq-provider              0.19.1
qiskit-ignis                      0.7.1
qiskit-terra                      0.20.2
```

On Linux

```
pip list | grep qiskit

qiskit                            0.36.2
qiskit-aer                        0.10.4
qiskit-ibmq-provider              0.19.1
qiskit-ignis                      0.7.1
qiskit-terra                      0.20.2
```

To check if updates are available, the following command shows newer versions.

```
# On MS Windows
pip list --outdated | select-string "qiskit"

# On Linux
pip list --outdated | grep qiskit
```

### 2.4.2. IBM Quantum Experience

In order to take the full advantage of Qiskit, you must first create an IBM Quantum Experience account. With IBMid, you can run Qiskit codes on real IBM quantum computers as well as on realistic simulations on your computer. Go to [quantum-computing.ibm.com](quantum-computing.ibm.com) and set up an account. Log in to your account and take a look at IBM Quantum Dashboard where you find many useful stuffs which we discuss in later chapters.

### 2.4.3. API key

Next, you need to obtain an API key and save it in a local computer.

1. Log in to IBM Quantum Experience at [quantum-computing.ibm.com](quantum-computing.ibm.com)
2. Click the user icon at the upper-right corner.
3. Click “Account setting”.

4. Click “Generate new token”
5. Click copy icon at the right end of the token box. Your token is copied to the clipboard.
6. Open a text editor and paste the token. Save it it to a temporary file so that you can copy the token at a later time if needed. Delete the file after the key is properly installed.
7. Open an Anaconda terminal window.
8. Start python and execute the following command at the python prompt:

```
>>> from qiskit import IBMQ
>>> IBMQ.save_account('past your token here')
```

The token must be inside the single quotes. Now, we verify if the token works.

```
>>> IBMQ.load_account()
```

You should get the following response:

```
<AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>
```

If it worked, delete the temporary file created at step 6. Otherwise, something went wrong. Try step 8 again. Make it sure that the whole key is pasted.

If you work on multiple computers, you have to install the same API on each machine.

### 2.4.4. Using Qiskit

Since Qiskit is a collection of python modules, we must import it to your code before using it. The package is so large that importing the entire package is not a good idea. In this book, we use only a small portion of it. As you move on, this book introduces some basic modules absolutely necessary for quantum computing and explains how to use them step by step.

### 2.4.5. Suggested Reading

As mentioned above, there are various online resources at qiskit.org and quantum-computing.ibm.com. In particular, the following online textbook is recommended.

- Learn Quantum Computation using Qiskit

In addition, the following paperback book is recommended.

- H. Norlén: *Quantum Computing in Practice with Qiskit and IBM Quantum Experience* (Packt, 2020). Source codes can be obtained at github.

```
import qiskit.tools.jupyter
%qiskit_version_table
%qiskit_copyright
```

## Version Information

| **Qiskit Software** | **Version** |
|---|---|
| `qiskit-terra` | 0.21.1 |
| `qiskit-aer` | 0.10.4 |
| `qiskit-ignis` | 0.7.1 |
| `qiskit-ibmq-provider` | 0.19.2 |
| `qiskit` | 0.37.1 |
| **System information** | |
| Python version | 3.9.12 |
| Python compiler | GCC 7.5.0 |
| Python build | main, Jun 1 2022 11:38:51 |

| | |
|---|---|
| OS | Linux |
| CPUs | 6 |
| Memory (Gb) | 62.7442131042480 5 |
| Mon Aug 22 09:39:19 2022 CDT | |

## 2.5. Cloud IDE

If your computer does not have enough storage or power to run python and jupyter, you can develop python codes and run them on cloud-based online IDEs. You can also upload the jupyter notebook files used in this book and run them. The most popular one is *Google Colab*. However, it is a general purpose IDE and some packages we use are not available. You must install them manually each time you use their service.[1] *IBM Quantum Lab* is another cloud-based online IDE and it is designed to work with IBM Quantum Computing service via Qiskit and thus all packages we need are preloaded. For our purpose, IBM Quantum Lab works better than Google Colab.

### 2.5.1. IBM Quantum Lab

IBM Quantum Lab allows to run JupyterLab on their server. It is designed for quantum computing and thus it is ready to run the codes in this book. Then, upload it to IBM Quantum Lab. You can run it on their server for free.

If you created your IBM Quantum Experience account as instructed in Section 2.4, you are ready to use IBM Quantum Lab. Login to IBM Quantum Dashboard. Click “Launch Lab”. Then you enter IBM Quantum Lab, which is JupyterLab running on their computer. On the left panel, you can create new folders, new files, or upload existing jupyter notebook files.

### 2.5.2. Google Colab

Google Colab also gives you a similar free development environment. It is not identical but compatible to JupyterLab. However, it is for general purpose and you must install necessary python packages such as qiskit each time. Nevertheless, it is convenient if you do not have access to a local jupyter environment.

You need to have a google account to use this free service. If you don’t have one yet, you can open an account at accounts.google.com. Once you obtained your account, sign in at golan.research.google.com. You can start writing codes or upload jupyter notebook files.

Unfortunately, Colab does not have qiskit. However, you can install it. Add the following code block at the beginning.

```
!pip install qiskit ipywidgets pylatexenc
```

Now, your qiskit codes run in Colab. The installation is only temporary. When you close the session, qiskit is removed. You must install it each time you start new Colab session.

**Running Qiskit on Google Colab**

If you want to run qiskit codes on Google Colab, you need to add the following command in a code block.

```
!pip install qiskit ipywidgets pylatexenc
```

[1] There are methods to store packages permanently but the procedure is rather complicated. Therefore, we do not discuss it here.

# 3. Quantum Mechanics

Quantum computation obviously relies on the principles of quantum mechanics. You need to understand them to learn quantum computation. However, traditional quantum mechanics courses usually do not cover the principles essential to quantum computation. Therefore, quantum mechanics is not prerequisite. No prior knowledge in quantum mechanics is assumed and I introduce basic mathematical expressions of quantum mechanics in Section 3.1 and Section 3.2. If you wish to learn them in depth, the following book is recommended.

- J. Audretsch: *Entangled Systems - New Directions in Quantum Physics* (Wiley-VCH, 2007)

The following book is the most popular book on quantum computation and contains a comprehensive introduction to quantum mechanics which covers everything you need to learn quantum computation. Advanced students are encourage to read it.

- M. A. Nielsen and I. L. Chuang: *Quantum Computation and Quantum Information*

## 3.1. State vectors

The theory of quantum mechanics is built on linear algebra. I hope that you have studied elementary linear algebra prior to this book. While I briefly introduce necessary mathematics, I expect you to review the theory of complex vector spaces (Hilbert spaces). You can find a good summary of mathematics is also given in this book

- J. Audretsch: *Entangled Systems - New Directions in Quantum Physics* (Wiley-VCH, 2007)

which I already recommended in the previous section.

If you want to practice mathematical calculation in depth, there is an exercise book on quantum computation.

- W.-H. Steeb and Y. Hardy: *Problems and Solutions in Quantum Computing and Quantum Information* (World Scientific, 2012).

The following is a summary of Hilbert space using abstract mathematical expressions. The ket and bra notation of vectors may be new to you. Otherwise, it is just a standard linear algebra. I will discuss each item in later chapters with practical examples for quantum computation. For now, refresh your understanding of linear algebra.

### 3.1.1. Ket and bra

The state of a quantum object is mathematically described by a complex vector in a *Hilbert space*, which is called *state vector*. The state vector is denoted as $|\bullet\rangle$, which is called *ket vector* or simply *ket*. "●" in the ket can be any symbol representing the state. For example, an electron in a spin-up state is commonly expressed by $|\uparrow\rangle$. We could use $|\heartsuit\rangle$, $|\spadesuit\rangle$, $|\diamondsuit\rangle$, $|\clubsuit\rangle$ to express the states of quantum cards (if they exist.) We often write an arbitrary state as $|\psi\rangle$.

For every ket $|\psi\rangle$, there is a corresponding complex vector $\langle\psi|$, which is known as *bra vector* or just *bra*. Mathematically, $\langle\psi|$ is the *adjoint* or *dual* of $|\psi\rangle$. The corresponding bra with the same symbol as a ket represents the same state of the qubit. For example, both $|\uparrow\rangle$ and $\langle\uparrow|$ represent the same spin-up state. We need bra vectors to compute inner products (See subsection Inner product).

You are familiar with Euclidean vectors such as forces and velocities and the rules of calculation with them. Since kets and bras are vectors, they share the same mathematical operations with the Euclidean vectors, namely two basic mathematical operations, *addition* $|a\rangle + |b\rangle$ and *scalar mulitplication* $\lambda|a\rangle$ ($\lambda$ is a complex number). The regular rules of calculation are listed below. The dual of the operations are $\langle a| + \langle b|$ and $\langle a|\lambda^*$ where $\lambda^*$ is a complex conjugate of $\lambda$.

### 3.1.2. Addition

The following statements are valid for any $|a\rangle$, $|b\rangle$, $|c\rangle$ in the same vector space. The same rules are applied to bra vectors.

| rule | mathematical expression |
|---|---|
| closure | $|a\rangle + |b\rangle$ is a ket in the same vector space. |
| commutativity | $|a\rangle + |b\rangle = |b\rangle + |a\rangle$ |
| associativity | $(|a\rangle + |b\rangle) + |c\rangle = |a\rangle + (|b\rangle + |c\rangle)$ |
| null vector | There exists null vector $0$ such that $|a\rangle + 0 = |a\rangle$ |
| inverse | There exists inverse vector $|-a\rangle$ such that $|a\rangle + |-a\rangle = 0$. In a common expression, $|-a\rangle = -|a\rangle$ |

### 3.1.3. Scalar multiplication

The following statements are valid for any $|a\rangle$ and $|b\rangle$ in the same vector space and any complex numbers $\alpha$ and $\beta$. The same rules are applied to bra vectors.

| rule | mathematical expression |
|---|---|
| closure | $\alpha|a\rangle$ is another ket in the same vector space |
| vector distributivity | $\alpha(|a\rangle + |b\rangle) = \alpha|a\rangle + \alpha|b\rangle$ |
| scalar distributivity | $(\alpha + \beta)|a\rangle = \alpha|a\rangle + \beta|a\rangle$ |
| associativity | $\alpha(\beta|a\rangle) = (\alpha\beta)|a\rangle$ |
| identity | There exists identity $I$ such that $I|a\rangle = |a\rangle$ (We write $I$ as $1$.) |

### 3.1.4. Superposition

Combining addition and scalar multiplication, we can construct a superposition of vectors,

$$|\psi\rangle = \alpha|a\rangle + \beta|b\rangle$$

where $\alpha$ and $\beta$ are complex numbers. The corresponding bra is

$$\langle\psi| = \langle a|\alpha^* + \langle b|\beta^*.$$

Notice that the coefficients are complex conjugate of the original ones in the ket.

**Exercise** 3.1.1
Consider a superposition state $\alpha|a\rangle + e^{i\theta}\alpha|b\rangle$ where $\theta$ is real. Find the corresponding bra vector.

### 3.1.5. Inner product

In the Euclidean vector space, an inner product (dot product) $\vec{a}\cdot\vec{b}$ is real and $\vec{a}\cdot\vec{b} = \vec{b}\cdot\vec{a}$. The inner product in a Hilbert space differs in two points. Firstly, the inner product in a Hilbert space $\langle a|b\rangle$ is complex. Secondly, the inner product is not symmetric between $a$ and $b$, that is $\langle a|b\rangle \neq \langle b|a\rangle$. Instead, the two inner products are complex conjugate to each other as $\langle a|b\rangle^* = \langle b|a\rangle$.

The inner product between $|c\rangle$ and a superposition state $|\psi\rangle = \alpha|a\rangle + \beta|b\rangle$ can be computed with the distribution rule

$$\begin{aligned}\langle c|\psi\rangle &= \langle c|\left(\alpha|a\rangle + \beta|b\rangle\right) = \alpha\langle c|a\rangle + \beta\langle c|b\rangle\\ \langle\psi|c\rangle &= \left(\alpha^*\langle a| + \beta^*\langle b|\right)|c\rangle = \alpha^*\langle a|c\rangle + \beta^*\langle b|c\rangle.\end{aligned}$$

**Exercise** 3.1.2

1. In the above example of inner product, show that $\langle c|\psi\rangle^* = \langle\psi|c\rangle$. (otherwise, the distribution rule does not work.)
2. Show that $\langle\psi|\psi\rangle$ is real for any $|\psi\rangle$.

## 3.1.6. Orthogonality

The inner product of two Euclidean vectors $\vec{a}$ and $\vec{b}$ may be expressed as $\vec{a}\cdot\vec{b} = ab\cos\theta$ where $\theta$ is the angle between the two vectors. When $\theta = \pm\frac{\pi}{2}$, the inner product vanishes. Then, the two vectors are said to be orthogonal. Although no such angle can be defined between two kets, we say that $|a\rangle$ and $|a\rangle$ are *orthogonal* if $\langle a|b\rangle = 0$.

## 3.1.7. Norm and normarization

Recall that the norm (magnitude) of an Euclidean vector is defined by $\|\vec{a}\| = \sqrt{\vec{a}\cdot\vec{a}}$. The norm of $|a\rangle$ is defined in the same way. Replacing the dot product with inner product, we have the norm of $|a\rangle$ as

$$\||a\rangle\| = \sqrt{\langle a|a\rangle} \geq 0.$$

When $\langle a|a\rangle = 1$, the ket is said to be *normalized*. Although we cannot express the ket as an arrow, physicists often use an imaginary arrow to represent a ket and the norm can be viewed as the length of the imaginary arrow. This analogy works quite well. So, the normalized ket is much like a unit vector.

**Exercise** 3.1.3 Show that $|b\rangle = \frac{|a\rangle}{\sqrt{\langle a|a\rangle}}$ is normalized.

**Exercise** 3.1.4 $|a\rangle$ and $|b\rangle$ are normalized and orthogonal to each other. Show that $|\psi\rangle = |a\rangle + i|b\rangle$ and $|\varphi\rangle = |a\rangle - i|b\rangle$ are orthogonal.

## 3.1.8. Global phase

It is easy to show that if $|\psi\rangle$ is normalized, $e^{i\theta}|\psi\rangle,\ \theta \in \mathbb{R}$ is also normalized. Although they are mathematically different, the two kets corresponds to the same physical state. The factor $e^{i\theta}$ is known as global phase and does not play any significant role in physics. Hence, we can ignore it. Quantum computation can be simplified by utilizing the freedom to chose any global phase.

## 3.1.9. Basis sets

In a freshman physics course, perhaps you expressed a force vector in the three-dimensional Euclidean space as $\vec{F} = F_x\vec{i} + F_y\vec{j} + F_z\vec{k}$. In fact, any vector can be expressed in the same way using the basis vectors $\vec{i}$, $\vec{j}$ and $\vec{k}$. These three vectors are orthogonal to each other and their magnitude is 1. Hence they are *orthonormal basis*.

Similarly, a $n$-dimensional Hilbert space is *spanned* by a basis set $\{|e_0\rangle, |e_1\rangle, \cdots, |e_{n-1}\rangle\}$. We assume that the basis vectors are normized and orthogonal to each other. Such basis set is called *orthonormal basis* and the basis vectors satisfy

$$\langle e_i|e_j\rangle = \delta_{ij} \tag{3.1}$$

where $\delta_{ij}$ is the usual Kronecker's delta.

The basis vectors are *linearly independent* and *complete*. That means that any ket vector can be expressed as a linear combination of the base vectors as

$$|\psi\rangle = c_0|e_0\rangle + c_1|e_1\rangle \cdots c_{n-1}|e_{n-1}\rangle = \sum_{i=0}^{n-1} c_i|e_i\rangle \tag{3.2}$$

where $c_i$ is complex. Using the orthonormality condition (3.1) , we can find the expansion coefficient as $c_i = \langle e_i|\psi\rangle$ which is a crucial quantity in quantum mechanics.

The corresponding bra vector are expressed as

$$\langle\psi| = c_0^*\langle e_0| + c_1^*\langle e_1| \cdots c_{n-1}^*\langle e_{n-1}| = \sum_{i=0}^{n-1} c_i^*\langle e_i|.$$

Notice that the expansion coefficients of the bra are complex conjugate of the coefficients of the ket.

If $|\psi\rangle$ is normalized, the coefficients is normalized as $\sum_j |c_j|^2 = 1$.

### 3.1.10. Completeness

In Equation (3.2) we expanded an arbitrary vector in a complete basis set. The following *closure relation* guarantees the completeness.

$$|e_0\rangle\langle e_0| + |e_1\rangle\langle e_1| + \cdots + |e_{n-1}\rangle\langle e_{n-1}| = I \tag{3.3}$$

where $\{|e_i\rangle\}$ is an orthonormal basis. $|e_i\rangle\langle e_i|$ is a kind of operator called *projection operator*, which we study in XXX. It acts on a ket as follows:

$$(|e_i\rangle\langle e_i|)\,|\psi\rangle = |e_i\rangle(\langle e_i|\psi\rangle) = c_i e_i\rangle$$

where $c_i = \langle e_i|\psi\rangle$.

### 3.1.11. Two-dimensional Hilbert space

In the theory of quantum computation, a two-dimensional Hilbert space denoted as $\mathbb{C}^2$, appears quite often. It is a convention to use a complete orthonormal basis set $\{|0\rangle, |1\rangle\}$ known as *computational basis*. We use this basis set extensively through out the book. The orthonormality of this basis set is given by

$$\langle 0|0\rangle = \langle 1|1\rangle = 1, \quad \langle 0|1\rangle = \langle 1|0\rangle = 0 \tag{3.4}$$

Another basis set

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{3.5}$$

is also regularly used.

**Exercise** 3.1.5 $|\pm\rangle$ are orthonormal.

## 3.2. Operators

An operator is defined on a Hilbert space and acts on a vector in the same Hilbert space. More precisely, we use *linear* operators. In this section, we learn the following mathematical operations associated with operators. Physical interpretations will be followed after this section.

| operation | mathematical expression |
|---|---|
| operator times ket | $A\|\psi\rangle$ |
| adjoint operator | $\langle\psi\|A^\dagger$ |
| scalar multiplication | $\lambda A, \quad \lambda \in \mathbb{C}$ |
| operator sum | $A+B$ |
| identity operator | $I$ |
| operator multiplication | $AB$ |
| commutation | $[A\, B] = AB - BA$ |
| matrix element | $\langle\psi\|A\|\varphi\rangle$ |
| trace | $\mathrm{tr} A$ |
| dyad | $\|\psi\rangle\langle\varphi\|$ |
| projection operator | $P^2 = P$ |
| Self-adjoint operators | $A^\dagger = A$ |
| Unitary operator | $U^\dagger U = I$ |
| Eigenvalue and eigenvector | $A\|a_i\rangle = \lambda_i\|a_i\rangle$ |
| Pauli operator | $X, Y, Z$ |
| function of operator | $f(A)$ |

### 3.2.1. Operator times ket

When a operator $A$ acts on any ket in the Hilbert space, it transforms the ket to another ket in the same Hilbert space,

$$|b\rangle = A|a\rangle.$$

Similarly every bra is transformed as

$$\langle b| = \langle a|A^\dagger$$

where $A^\dagger$ is *Hermite conjugate* or *adjoint* of $A$. The adjoint of the adjoint operator is the original operator, that is $(A^\dagger)^\dagger = A$.

It is wrong to think that $A^\dagger$ always act on bra. It can be applied to ket and $A^\dagger|a\rangle$ is perfectly OK. However, in general $A|a\rangle \neq A^\dagger|A\rangle$. Therefore $A$ and $A^\dagger$ are two different operators.. By definition, the adjoint expression of $A^\dagger|A\rangle$ is $\langle a|(A^\dagger)^\dagger = \langle a|A$.

**Example** 3.2.1 Operator $X$ transforms the basis vectors as $X|0\rangle \equiv |1\rangle$ and $X|1\rangle \equiv |0\rangle$.

**Example** 3.2.2 Consider $H$ transforms the basis vectors as $H|0\rangle \equiv |+\rangle$ and $H|1\rangle \equiv |-\rangle$. (See (3.5) for the definition of $|\pm\rangle$.) $H$ is known as *Hadamard* gate and one of the very important quantum gates.

### 3.2.2. Scalar multiplication

$\lambda A$ is another operator where $\lambda$ is any complex number. You can place the scalar in the either side of the operator, i.e., $\lambda A = A\lambda$. The adjoint of $\lambda A$ is $\lambda^* A^\dagger$. Notice that the complex conjugate of the scalar in the adjoint.

The following associativity holds:

$$(\lambda A)|a\rangle = \lambda(A|a\rangle) = A(\lambda|a\rangle)$$
$$(\lambda\eta)A = \lambda(\eta A)$$

**Exercise** 3.2.1   Find the adjoint of $\lambda A|a\rangle$.

### 3.2.3. Linearity

When an operator acts on a superposition state, the following linearity holds:

$$A\left(\alpha|\psi\rangle + \beta|\phi\rangle\right) = \alpha A|\psi\rangle + \beta A|\phi\rangle$$

where $\alpha, \beta \in \mathbb{C}$.

---

**Exercise** 3.2.2 Show that $H|+\rangle = |0\rangle$ and $H|-\rangle = |1\rangle$ where $H$ is the Hadamard gate.

---

### 3.2.4. Operator sum

The sum of operators satisfies the following distributivity:

$$(A+B)|\psi\rangle = A|\psi\rangle + B|\psi\rangle.$$

### 3.2.5. Operator products

For any operators $P$ and $Q$ in a Hilbert space, the product $PQ$ is also an operator in the same Hilbert space. When $Q$ is applied on $|a\rangle$ and $P$ is applied on the result of the first operation, the final result is also a vector in the same Hilbert space:

$$P(Q|a\rangle) = P|b\rangle = |c\rangle$$

where $|b\rangle = Q|a\rangle$. The associativity $P(Q|a\rangle) = (PQ)|a\rangle$ holds.

Similarly, when we apply $P$ on $|a\rangle$ first and $Q$ afterward, the associativity $Q(P|a\rangle) = (QP)|a\rangle$ holds as well. In general the outcomes of these two operations are different and thus $PQ \neq QP$. In other words, $P$ and $Q$ do not *commute*. The difference between $PQ$ and $QP$ is expressed with *commutator*

$$[P,Q] \equiv PQ - QP.$$

Only when $[P,Q] = 0$, we can change the order of operations. The commutation relation is of fundamental importance in quantum mechanics.

For some operators, anti-commutation

$$\{P,Q\} \equiv PQ + QP$$

also plays an important role.

The dual of the above transformation is

$$\langle c| = \langle b|P^\dagger = (\langle a|Q^\dagger)P^\dagger$$

Using the associativity $PQ$, $\langle c| = \langle a|(PQ)^\dagger$. By direct comparison, we find

$$(PQ)^\dagger = Q^\dagger P^\dagger. \tag{3.6}$$

We learned a similar relation in a linear algebra course: $(PQ)^T = Q^T P^T$ where $P$ and $Q$ are matrices and $T$ is transpose. We will see that this equality is indeed equivalent to (3.6).

### 3.2.6. Identity operator $I$

The simplest operator is the *identity* operator $I$ which transforms a vector to itself:

$$|a\rangle = I|a\rangle.$$

The adjoit of the transformation is given by

$$\langle a| = \langle a|I^\dagger$$

It is easy to show that for any operator $A$,

$$A\,I = I\,A = A.$$

The identity operator looks trivial. However, it is ubiquitous in quantum computation. First of all, it is needed to define inverse operators. The inverse operator of $A$ is defined by

$$A^{-1}A = AA^{-1} = I.$$

Not all operators have their inverse but most of operators we use in quantum computation do.

**Exercise** 3.2.3 The identity operator $I$ satisfies $I = I^{-1} = I^\dagger = I^2$. Prove it.

## 3.2.7. Matrix elements

Consider an expression $\langle a|A|b\rangle$. It is not clear if $A$ is acting on the ket or the bra. Let us assume that $A$ acts on the ket as$\langle b|(A|a\rangle)$. Then, it is a regular inner product. If $A|a\rangle = |c\rangle$, it is indeed $\langle b|c\rangle$. It turns out that when $A$ can act on the bra, the same inner product is obtained. Therefore, we don't have to specify which way the operator acts. The quantity $\langle a|A|b\rangle$ is called *matrix element* of $A$.

Most of mathematical objects we encounter in quantum mechanics are defined in some abstract space and they are not directly connected to what we see in the real world. The inner products and also matrix elements are numbers we are familiar with. Hence, abstract expressions of quantum mechanics and physical values we experience in the real world are connected through the inner product. In this sense, the inner product is very essential. Without it, quantum mechanics is completely separated from the real world and would be useless.

**Example** 3.2.3 $\langle 0|X|0\rangle = 0$.

Calculation: $\langle 0|X|0\rangle = \langle 0|1\rangle = 0$ where $X|0\rangle = |1\rangle$ is used.

**Example** 3.2.4 $\langle +|H|+\rangle = \frac{1}{\sqrt{2}}$.

Calculation: $\langle +|H|+\rangle = \langle +|0\rangle = \left(\frac{1}{\sqrt{2}}(\langle 0| + \langle 1|)\right)|0\rangle = \frac{1}{\sqrt{2}}(\langle 0|0\rangle + \langle 1|0\rangle) = \frac{1}{\sqrt{2}}.$.

**Exercise** 3.2.4 Evaluate matrix element $\langle +|H|-\rangle$.

## 3.2.8. Dyads

A kind of product between a bra $\langle\psi|$ and a ket $|\varphi\rangle$

$$|\varphi\rangle\langle\psi|$$

is an operator known as *dyad*. In fact, it transforms a ket to another ket:

$$(|\varphi\rangle\langle\psi|)|a\rangle = |\varphi\rangle\langle\psi|a\rangle = \lambda|\varphi\rangle$$

where $\lambda = \langle\psi|a\rangle$.

A special case of dyad, $|\psi\rangle\langle\psi|$ is known as *orthogonal projection operator*. Assuming $|\psi\rangle$ is normalized, it satisfies the idempotency $P^2 = P$. Mathematically, the idempotency is the definition of projection operators and the orthogonal projection operator is a special kind of projection operators. However, as in physics literature, we shall call it projection operator without "orthogonal". We have already encounter it in the closure relation of the complete basis set (3.3).

## 3.2.9. Self-adjoint operators

When $A = A^\dagger$, operator $A$ is self-adjoint (or Hermitian). Identity $I$ and projection operator $P$ are self-adjoint.

## 3.2.10. Unitary operators

When an operator transform a *state vector* to another *state vector*, the transformation should preserve the norm of the *stat vectors*. Consider a transformation $|\psi'\rangle = U|\psi\rangle$ and its adjoint $\langle\psi'| = \langle\psi|U^\dagger$. Since the norm must preserve, we have $\langle\psi'|\psi'\rangle = \langle\psi|U^\dagger U|\psi\rangle = \langle\psi|\psi\rangle$, which indicates that

$$U^\dagger U = I.$$

Any operator that satisfies this condition is *unitary*. In quantum computation , we apply a set of gates on an input state vector and the outcome is also a state vector, quantum circuit must be unitary operators (unless quantum measurement is involved in it).

Unitary operators are conveniently written in the exponential form

$$U = e^{-i\theta A}$$

where $A$ is a self-adjoint operator and $\theta$ is a real parameter.

Its adjoint is

$$U^\dagger = e^{i\theta A^\dagger} = e^{i\theta A}$$

which leads to $U^\dagger U = I$ and thus $U$ is unitary.

The operator exponential functions are used for time-evolution, space rotation, space translation, $\cdots$ and they are a key mathematical component for symmetry groups and conservation laws.

---

**Exercise** 3.2.5   Show that $U^\dagger U = e^{i\theta A^\dagger} e^{-i\theta A} = I$ if $A$ is self-adjoint.

---

### 3.2.11. Eigenvalues and eigenvectors

When an operator $A$ and its adjoint $A^\dagger$ commute, i.e. $[A, A^\dagger] = 0$, $A$ is a *normal* operator and there are special kets which are transformed to "itself" by $A$. Mathematically, it is expressed by

$$A|a\rangle = \lambda |a\rangle \tag{3.7}$$

where $\lambda \in \mathbb{C}$. The right hand side is not exactly the "itself" but multiplied by a constant $\lambda$ which is called *eigenvalue*. (In next section, we will see that $|a\rangle$ and $\lambda|a\rangle$ represent the same physical state.) The ket $|a\rangle$ is called *eigenvector* or *eigenket* of $A$.

There are many such vectors. More precisely, there are $N$ eigenkets in $N$-dimensional Hilbert space. Therefore, we write (3.7) as

$$A|a_i\rangle = \lambda_i |a_i\rangle, \quad i = 1, \cdots, N.$$

When $A$ is self-adjoint, the eigenvalues are all real, which is important when we discuss measurement of physical quantities in {numref}`sec_measurement}.

### 3.2.12. Trace of operators

Another operation on operators we often use is tracing. For an operator $A \in \mathcal{H}$, the trace of the operator is defined by

$$\text{tr} A = \sum_j \langle e_j | A | e_j \rangle$$

where $\{|e_j\rangle, \, j = 1, n\}$ is a basis set in the Hilbert space. $\langle e_j | A | e_j \rangle$ is matrix element.

The following properties of the trace are useful. ($A$and $B$ are operators and $\alpha, \beta \in \mathbb{C}$.)

- linearity

$$\text{tr}\,(\alpha A) = \alpha\, \text{tr} A$$
$$\text{tr}\,(\alpha A + \beta B) = \alpha\, \text{tr} A + \beta\, \text{tr} B$$

- cyclic permutations

$$\text{tr}\,(AB) = \text{tr}\,(BA)$$

- adjoint operator

$$\mathrm{tr} A^\dagger = (\mathrm{tr} A)^*$$

The following trace formula for dyads are also usefull

$$\mathrm{tr}\left(|\varphi\rangle\langle\psi|\right) = \langle\psi|\varphi\rangle.$$
$$\mathrm{tr}\left(A|\varphi\rangle\langle\psi|\right) = \langle\psi|A|\varphi\rangle.$$

---

**Exercise** 3.2.6 Show that $\mathrm{tr}\left([A,B]\right) = 0$, even when $[A,B] \neq 0$.

---

### 3.2.13. Pauli Operators

Pauli operators $\sigma_x$. $\sigma_y$, and $\sigma_z$ are originally used to express electron spin. They plays very important roles in the qubit-based quantum computation.

Traditionally in literature on quantum computation, symbols $X \equiv \sigma_x$, $Y \equiv \sigma_y$, and $Z \equiv \sigma_z$ are used. By definition, the Pauli operators satisfy

- commutation relation

$$[X,Y] = 2iZ, \qquad [Y,Z] = 2iX, \qquad [Z,X] = 2iY$$

- anti-commutatioan relation

$$\{X,Y\} = \{Y,Z\} = \{Z,X\} = 0$$

- self-inverse

$$X^2 = Y^2 = Z^2 = I$$

- self-adjoint

$$X^\dagger = X, \qquad Y^\dagger = Y, \qquad Z^\dagger = Z$$

- unitary

$$X^\dagger X = Y^\dagger Y = Z^\dagger Z = I$$

In $\mathbb{C}^2$, any operator $A$ can be written as

$$A = c_0 I + c_1 X + c_2 Y + c_3 Z$$

where $c_0 = \mathrm{tr}(IA)$, $c_1 = \mathrm{tr}(XA)$, $c_2 = \mathrm{tr}(YA)$, and $c_3 = \mathrm{tr}(ZA)$. When $A$ is self-adjoint, all the coefficients are real.

---

**Exercise** 3.2.7 Using the properties given above

1. Show that $XY = iZ$,
2. Show that $XYZ = iI$,
3. Show that $\mathrm{tr}X = \mathrm{tr}Y = \mathrm{tr}Z = 0$.

---

Among the three Pauli operators, $X$ is the most important for quantum computation. Its main roles are:

- $X$ flips the computational basis.

$$X|0\rangle = |1\rangle, \qquad X|1\rangle = |0\rangle.$$

- $X$ swaps the coefficients in a superposition state.

$$X\left(c_0|0\rangle + c_1|1\rangle\right) = c_1|0\rangle + c_0|1\rangle,$$

$Z$ does not flip the basis but change the relative phase.

- $Z$ does nothing on $|0\rangle$

$$Z|0\rangle = |0\rangle.$$

- $Z$ inverts the phase of $|1\rangle$.

$$Z|1\rangle = -|1\rangle.$$

- When acting on a superposition state $Z$ changes the relative phase as

$$Z\left(c_0|0\rangle + c_1|1\rangle\right) = c_0|0\rangle - c_1|1\rangle$$

When $Z$ acts on $|1\rangle$, the ket acquires a new phase “$-$”. Since the global phase is irrelevant, the operator essential does nothing thing to $|0\rangle$ nor $|1\rangle$. However, when it acts on a superposition state, the relative phase between $|0\rangle$ and $|1\rangle$ changes.

Any operator $A$ In $\mathbb{C}^2$ can be decomposed as

$$A = c_0 I + c_1 X + c_2 Y + c_3 Z$$

where $c_0 = \mathrm{tr}(IA)$, $c_1 = \mathrm{tr}(XA)$, $c_2 = \mathrm{tr}(YA)$, and $c_3 = \mathrm{tr}(ZA)$. This decomposition suggests that we need only $I$ and the Pauli operators to transform a qubit.

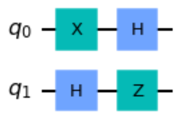
---

**Exercise** 3.2.8 Initially a qubit is in $|0\rangle$. We want to transform it to $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. In previous sections, we learn that the Hadamard gate transforms $|1\rangle$ to $|-\rangle$. Pick two operators from $X$, $Z$, $H$, and construct a quantum circuit that transforms $|0\rangle$ to $|-\rangle$. There are two possible combinations. Find both. (The following Qiskit example veryify the solutions.

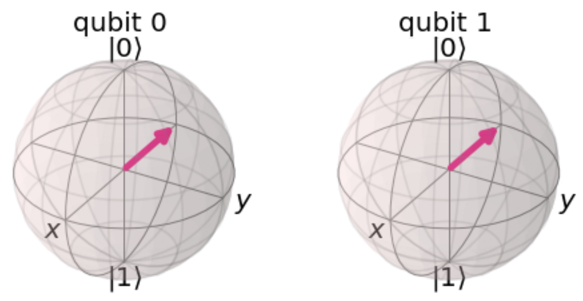
**Qiskit Example**

We shall demonstrate the two solution to the above exercise using Qiskit. We use two qubits. Initially they are both in $|0\rangle$. Then, we apply different gates to them. At the end, both should be in $|-\rangle$. In this example, we use `statevector_simulator` which calculate the transformation exactly.

```python
from qiskit import *
from qiskit.quantum_info import Statevector
# declare to use two qubit
qr = QuantumRegister(2,'q')

# reset a quantum circuit with the two qubit
qc = QuantumCircuit(qr)

# We apply X and H on the first qubit
qc.x(0)
qc.h(0)

# We apply H and Z on the second qubit
qc.h(1)
qc.z(1)

# Show the circuit
qc.draw('mpl')
```



```
# We use state vector simulator
backend = Aer.get_backend('statevector_simulator')
job = backend.run(qc)
result = job.result()
psi=result.get_statevector(qc)

# plot_bloch_multivector plots Bloch vector for each qubit.
from qiskit.visualization import plot_bloch_multivector
plot_bloch_multivector(psi)

# Find that both qubits are |->
```



**Qiskit note: Pauli Operators**

In Section 3.1, we calculated inner products of computational basis using `qiskit.opflow` module. The same module provides predefined Pauli operators `I`, `X`, `Y`, and `Z`. Applying operator on ket is done with “@”. For example $X|0\rangle$ is `(X@One).eval()` in Qiskit.

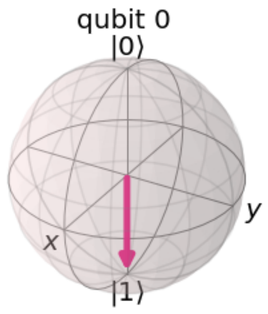
QIskit API Reference: qiskit.opflow

```
from qiskit.opflow import Zero, One, I, X, Y, Z

# Show that I|0>=|0> and (|1>=|1>
print("I|0> == |0> ",I@Zero == Zero)
print("I|1> == |1> ",I@One == One)

print()

# Show that Z|0>=|1> and X|1>=|0>
# X*ket must be evaluated by .eval()
print("X|0> == |1> ",(X@One).eval() == Zero)
print("X|1> == |0> ",(X@Zero).eval() == One)

print()

# Check if X is self-sdjoint
print("X is self-adjoint ",~X == X)

# Check if X is unitary
# I and X are defined in different way, we need to eval() in both sieds.
print("X is unitary ",(~X @ X).eval() == I.eval())
```

```
I|0> == |0>  True
I|1> == |1>  True

X|0> == |1>  True
X|1> == |0>  True

X is self-adjoint  True
X is unitary  True
```

**Exercise** 3.2.9 Demonstrate $Z|0\rangle = |0\rangle$ and $Z|1\rangle = -|1\rangle$ using the `qiskit.opflow` module.

```
from qiskit.visualization import plot_bloch_multivector
plot_bloch_multivector(One)
```



## 3.2.14. Functions of operators

A function $f(x)$ maps an input value $x$ to an output value $f$. Then what $f(A)$ means if $A$ is an operator? It is interpreted as a map from an operator $A$ to another operator $f$. For example exponential function $e^x$ is a number and $e^A$ is an operator. But it is not clear how the function of of operator is applied to a vector? We define $e^A$ using a power series (Taylor expansion). For example,

$$e^x = 1 + x + \frac{1}{2}x^2 + \cdots + \frac{1}{n!}x^n + \cdots$$

Now we replace 1 with $I$ and $x$ with $A$.

$$e^A = I + A + \frac{1}{2}A^2 + \cdots \frac{1}{n!}A^n + \cdots$$

Since we know how to calcuate the product of operators, there is no problem with $A^n$. Hence, we evaluate

$$e^A|\psi\rangle = I|\psi\rangle + A|\psi\rangle + \cdots + \frac{1}{n!}A^n|\psi\rangle + \cdots.$$

For a general function $f(A)$, we assume that $f(x)$ is a nice smooth function which can be expanded in a power series. Then, the function of opperator is understood as

$$f(A) = I + c_1 A + c_2 A^2 + \cdot + c_n A^n + \cdots, \qquad \text{where } c_n = \frac{1}{n!}\frac{df}{dx}\Big|_{x=0}$$

We don't consider a function singular at $x = 0$ like $\log A$.

A special care is required for a function of multiple operators. For example we are familiar with $e^{x+y} = e^x e^y,$ if $x$ and $y$ are regular numbers. In contrast, $e^{A+B} \neq e^A e^B,$ if $[A, B] \neq 0$.

---

**Exercise** 3.2.10 For Pauli operator $X$, show that $e^{i\theta X} = I\cos\theta - iX\sin\theta$ where $\theta$ is a real number. Hint: Use $X^2 = I$ in the power series.

When $\theta = \frac{\pi}{4}$, the equality indicates that $e^{i\frac{\pi}{4}X} = \frac{1}{\sqrt{2}}(I - iX)$. Applying the operator on $|0\rangle$ we obtain $e^{i\frac{\pi}{4}X}|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)$. In the following Qiskit example, we will check it using `qiskit.opflow` module.

---

```
# import numpy
import numpy as np

# import qiskit.opflow
from qiskit.opflow import Zero, One, I, X, Y, Z

# we use Statevector class for vidualization
from qiskit import *

# construct the exponential operator using a method exp_i()
# X.exp_i() creates exp(i X)
expX=(np.pi/4*X).exp_i()

# evaluate expX * Zero
psi=(expX@Zero).eval()

# print out the results
# attribute 'primitive' extracts the information of the vector
# method 'draw('latex')' print out the vector using LaTeX.
# The detailed usage of attributes and methods are given in later chapters.
psi.primitive.draw('latex')
```

$$\frac{\sqrt{2}}{2}|0\rangle - \frac{\sqrt{2}i}{2}|1\rangle$$

Version 0.8.0: Last modified on May 3, 2022

## 3.3. Time-evolution

The state of the classical particles change according to the Newton equation, which is a set of ordinary differential equations. Equivalently, the trajectory can be computed from the Hamilton equations of motion, by measurement

$$\dot{q} = \frac{\partial H}{\partial p}, \qquad \dot{p} = -\frac{\partial H}{\partial q}$$

where $H$ is a Hamiltonian.

The state vector of a quantum system evolves in time according to the Schrödinger equation

$$i\frac{\partial}{\partial t}|\psi\rangle = H|\psi\rangle$$

where $H$ is Hamiltonian operator. Equivalently, the solution can be written as

$$|\psi(t)\rangle = U(t)|\psi(0)\rangle$$

where $U(t)$ is a unitary operator and $|\psi(0)\rangle$ an initial state. We call this type of evolution *unitary* evolution. Quantum computation solves this equation. We will solve it using `Qiskis`.

(version 0.0.1: Lat modified on April 7, 2022)

## 3.4. Quantum measurement

Wikipedia: Measurement in quantum mechanics

An important role of theoretical physics is to predict physical quantities such as energy and experimental physics tries to measure them. Without measurement as accurate as possible. Without measurement, there is no physics.

In classical mechanics, we can obtain a precise value from any physical quantity upon measurement and we can predict it by solving the Newton equation. If the same measurement is done on identical copies of the original system, the same value is always obtained. If the outcomes differ, then that must be due to measurement error (e.g., a bad measurement device).

In quantum mechanics, as we already discussed in previous chapters, the outcomes of measurement is *stochastic* even when the measurement apparatus is perfect. The principle of quantum mechanics cannot predict the outcome of single measurement. Only we can predict is the probabilities. In this chapter, we present a formal theory of quantum measurement.

### 3.4.1. Observables

Wikipedia: Observable

We haven't discussed how to describe physical quantities in quantum mechanics. They are not number. They are a special kind of operators in the Hilbert space of a quantum system. The operators corresponding to physical quantities are called *observable*. In order for operator $A$ to be observable, it must be *Hermitian* (self-adjoint). That is $A^\dagger = A$. Since it satisfies $[A^\dagger, A] = 0$, the observable is also a *normal* operator, which guarantees that the observable has eigenvalues $a_i$ and eigenvectors $|e_i\rangle$. Thus we have

$$A|a_i\rangle = a_i|a_i\rangle, \quad i = 1, \cdots, d$$

where $d$ is the dimension of Hilbert space. There are two important properties of the observable:

1. all eigenvalues are real
2. eigenvectors form a complete orthonormal basis

which are essential to the quantum measurement.

In quantum computation, computational basis vectors $|0\rangle$ and $|1\rangle$ are the eigenvectors of a Pauli operator $Z$. Hence, we measure $Z$.

### 3.4.2. Outcomes of measurement

When we measure a physical quantity, we expect that the outcome is a number, not an operator. The role of operator (observable) is to determine possible outcome. The outcome of the measurement must be an eigenvalue of the observable. We obtain one of $a_i$ when $A$ is measured. But there are many eigenvalues. Which one are we going to obtain? Quantum mechanics can't tell us what precisely we get. No one can predict it. Only we know the probability to obtain a particular eigenvalue. If the state of the system is $\rho$, the probability of finding $a_i$ is given by

$$p_i = \langle a_i|\rho|a_i\rangle. \tag{3.8}$$

which is vaklid for both pure and mixed states.

For pure state, we already use the probability. Consider the system on a pure state $|\psi\rangle$, we expand it in basis $\{|a_i\rangle\}$ (We can do so since the eigenvectors form a complete orthonormal basis):

$$|\psi\rangle = c_1|a_1\rangle + c_2|a_2\rangle + \cdots + c_d|a_d\rangle$$

The Born rule indicates that we obtain $a_i$ with probability $p_i = |c_i|^2 = |\langle a_i|\psi\rangle|^2$. Recalling that the density operator of the pure state is $\rho = |\psi\rangle\langle\psi|$, Eq. (3.8) gives the same probability.

**Exercise** 3.4.1 Show that $\langle a_i|\rho|a_i\rangle = |\langle a_i|\psi\rangle|^2$ for the pure state $|\psi\rangle$.

### 3.4.3. Projective measurement

In classical mechanics, we can measure a physical quantity without disturbing the system state. That is not the case of quantum systems. When the measurement results in $a_i$, the state also is transformed to $|a_i\rangle$ regardless of the original state before the measurement. This transformation is known as *wave function collapse*. The mechanism of the collapse is still not known but we are able to explain experiment results based on this postulate. No exception has been reported since the inception of quantum mechanics 100 years ago.

Recall that the quantum mechanics is is applied to an ensemble and measurement is needed to be repeated. If we throw away other states and keep only $|a_i\rangle$, then the resulting state is a pure state $|a_i\rangle$. This kind of measurement is called *selective measurement*. On the other hand, if we keep all outcomes (*non-selective measurement*), we have a classical mixture of $|a_i\rangle$ with probability (3.8):

$$\rho_{\text{out}} = \sum_i p_i |a_i\rangle\langle a_i| = \sum_i |a_i\rangle\langle a_i|\rho_{\text{in}}|a_i\rangle\langle a_i|. \tag{3.9}$$

where $\rho_{\text{in}}$ and $\rho_{\text{out}}$ are the density operators before and after the measurement, respectively. Noting that $|a_i\rangle\langle a_i|$ is a projection operator, we can interpret Eq. (3.9) as a projection of the state to $|a_i\rangle$ and this type of measurement is called *projective measurement*.

### 3.4.4. Expectation values

### 3.4.5. Uncertainty principle

In classical mechanics, we can obtain a precise value from any physical quantity upon measurement and we can predict it by solving the Newton equation. If the same measurement is done on indentical copies of the original system, the same value is always obtained. If the outcomes differ, then tha must be due to measurement error (e.g., a bad measurement device).

In quantum mechanics, as we already discussed in previous chapters, the outcomes of measurement is *stochastic* even when the measurement apparatus is perfect. The principle of quantum mechanics cannot predict the outcome of single measurement. Only we can predict is the probabilities. In this chapter, we present a formal theory of quantum measurement.

When we have many copies of the identical quantum states, the outcomes of measurement are usually different. What quantum mechanics can tell us is a *probability* of finding a particular value. That means that the outcome of measurement is randomly chosen according to the probability. Only in special cases, we can get the same value from the all copies, which we can predict from the state vector.

When the outcome is stochastic, we need to resort to the methods of statistics. Computing expectation value and standard deviation is essential since we can predict them based on the current theory of quantum mechanics.

Furthermore, the measurement changes the state and the new state depends on the value of the outcome. When the measurement picked an eigenvalue of the observable, the state jumps to the corresponding eigenvector. This jump cannot be described by the current theory of quantum mechanics. *Decoherence* may explain a part of this process but the detail remains unclear.

In addition to the stochastic nature of quantum measurement, there is another notable difference between classical and quantum measurements. In classical mechanics, we can determine the precise values of two or more physical quantities simultaneously by measurement. That may not be possible in quantum mechanics depending on what you measures. For example, the position and momentum of a classical particle can be precisely determined. In quatum mechanics, on the other hand, if both quantities are measured from many identical copies, it is impossible to have the same outcome from the all copies. If all copies have the same position, then every copy has very different value of momentum. In typical cases, each copy results in a pair of different values from other copies, hence there are uncertainties in both quantities.

The interpretation of quantum measurement is very counter intuitive. We still don’t know how quantum measurement takes place. Many physicists are not fully convinced by the current theory of quantum mechanics and believe that the theory of quantum mechanics is still incomplete. There are other interpretation of quantum mechanics such as the pilot wave theory. However, all experimental results agree with the probabilities computed from the state vector. Therefore, we adopt the standard theory of quantum mechanics in this book.

Finding the state of a system is not necessarily the final goal of physics. We want to know the values of other quantities such as energy In classical mechanics, physical quantities are naturally defined as number and the measurement of a physical quantity produces a unique value. On contrary to our intuition, physical quantities in quantum mechanics are not number. They are *self-adjoint operator* in the Hilbert space. The outcome of a measurement is one of the *eigenvalues* of the operator and we don't know which one will be the result of the measurement. This strange behaviors are the next topics.

**Related topics in Wikipedia**

- Measurement in quantum mechanics
- Observable
- Born rule
- Wave function collapse

Version 0.1.0: Last modified on May 3, 2022

# 4. What is a Quantum Computer?

In modern life style, computers are used everywhere, inside cars, TV sets, kitchen appliances and cell phones, to name a few. They are all operated in the same way with classical *bits*. The principles of computation are developed by major mathematicians such as Alan Turing (universal Turing machine) and von Neumann (von Neumann architecture). Furthermore, classical information theory developed by another genius mathematician, Claude Shannon advanced the classical computation to a much higher level.

While the classical computation technology is matured, the computability theory shows that there are many hard problems that cannot be solved by the classical computer. See for example travelling salesman problem. The demands of new computers that can solve such hard problems keep rising. On the other hand, the current encryption based on RSA keys works precisely because classical computers have the limitation. However, in 1994 Peter Shore found a *quantum algorithm* to breaking the RSA encryption. If quantum computers should become available, data security based on the RSA keys would be obsolete. This is a national security problem (See this national security memorandom on May 4, 2022) and it is quickly becoming a real issue. See "The race to save the Internet from quantum hackers".

So, what is a quantum computer? It is our current understanding that all physical phenomena are governed by quantum mechanics. The computer you are using is indeed built upon the laws of quantum mechanics. Electrons in the semiconductor chips must be treated as quantum particles. Nevertheless, it is called classical computer. The difference between classical and quantum computing is how *information* is stored and processed. Information must be stored in certain *stable* physical states. If they are unstable, information is lost quickly. Information in classical devices is safely stored in macroscopic states against disturbances from the environment but an important signature of quantum mechanical behavior, *coherence*, is completely abandoned. On the other hand, quantum computers store in information in a microscopic state which is fragile and easily disturbed by the environmental noises but quantum mechanical coherence is used as computational resources. The recent advances in quantum technology have made it possible to keep quantum information alive long enough to carry out computation.

In this chapter, I briefly introduce basic idea of classical and quantum computers and discuss what can be advantage of quantum computers over classical ones.

## 4.1. Classical computation

We carry out classical computation every day on many different devices, laptop computers, mobile phones, automobiles, smart TVs, refrigerators, … to name a few. These devices are made from semiconductors chips and optionally magnetic materials and their functionalities are governed by quantum mechanics. Nevertheless, we call them *classical computer*. Why? While the underlying physical processes are quantum mechanical, *information* stored and processed in these materials is classical. For example, macroscopic electric current and voltage, which are classical quantities, are used to store and process information. In other words, a large number of electrons are used to store even the smallest amount of information. In contrast, quantum computation relies on microscopic states , e. g., energy eigenstates in an atomic ion. In th following, we discuss what is the classical information and how it is processed in the classical devices.

### 4.1.1. What is classical information?

We ask a question because we do not know something and try to find it. In other words, wey to get new information. But what is information and can we quantify the amount of information? To answer this question, information theory was developed in the first half of twenty century by Claude Shannon and others.

In essence, the amount of information we obtain is the amount of uncertainty decreased. If there is no uncertainty, there is nothing to ask. The undertainties arise in several different ways. The outcome of an event that is going to happen in future can be uncertain if we cannot predict it precisely. The uncertainty is due to the stochastic nature of the event. IIn another case, the event has already happened and some people know the outcome but we dont' know it. For us, this is uncertainty but it is due to our ignorance. In either case, once we know the outcome, the uncertainty disappears and we obtain the information.

The uncertainty can be mathematically expressed as probability. Consider coin tossing. For tossing the coin, no one knows the outcome. However, we know that the outcome must be one of head and tail. We cannot predict the outcome because the motion of th coin is chaotic and thus the outcome is stochastic. Assuming the coin is ideal, the probability of finding head and tail ares $p_{\text{head}} = \frac{1}{2}$ and $p_{\text{tail}} = \frac{1}{2}$, which is the uncertainty associated with the coin tossing. The amount of the information (undertainty) is given by

$$H = -p_{\text{head}} \log p_{\text{head}} - p_{\text{tai}} \log p_{\text{tail}} = \log 2$$

which is known as *Shannon entropy* or *information entropy*. If base 2 is used, $\log_2 2 = 1$ and we say one *bit* of information is gained. Here *bit* is the unit of the amount of information.

After the coin is tossed and landed on someone's hand, the outcome is not known to us until the person shows it. The uncertainty is due to ignorance but the amount of information we will find remain the same.

Here is the general definition of the Shannon entropy For a set of all possible outcomes, $\{\omega_1, \omega_2, \cdots, \omega_N\}$ where $N$ is the total number of possible outcomes and the probability $p_i$ of finding the outcome $\omega_i$ for all $i$ is defined, the amount of information is measured by \$$H = -\sum_{i=1}^{N} p_i \log p_i$.
\$

### 4.1.2. Classical bits

Selecting one out of two possibilities is the simplest question. Coin tossing is one of them. In classical computer the smallest information unit has two possibilities `0` and `1`, which are assigned to two well-defined physical states in the devices, e.g., `0` to $0$ volt and `1` to $1$ volt if the voltage is used.

Mathematicall, we use a variable $b \in \{\mathtt{0}, \mathtt{1}\}$ for a single bit. The computer consists of many bits, bit 0, bit 1, bit 2, ……. For convenience, we denote the bits as $b_0, b_1, b_2, \cdots$ and altogether the information is stored in a string of n bits, $'\, b_{n-1}\, b_{n-2}\, \cdots\, b_1\, b_0\, '$, for example '0010101' (in this book, the bit string is in single quotes.) Notice the order of bits in the string. It is the standard to write string from the right to the left.

### 4.1.3. Copying a state of bit

In classical computers, the information is stored as bit strings in random access memory (RAM). To process information, the bit strings are copied to the central proccessing unit (CPU). After the information is processed, the outcome is copied to some location in RAM. Copying the bits is a fundamental aspect of classical information processsing. As you see below, the input information is usually lost in during processing in CPU. In contrast, quantum computation does not have such luxury since cloning of quantum state is not possible (*no cloning theorem*).

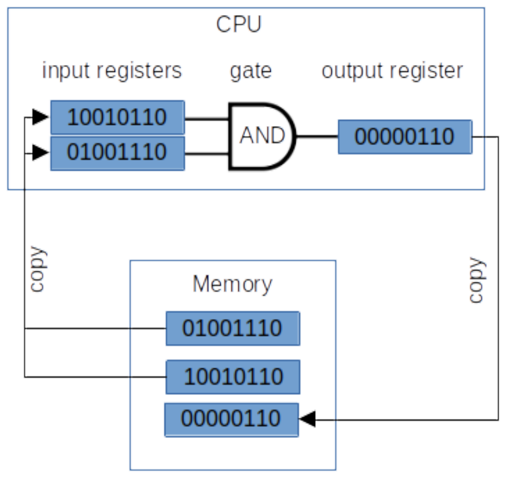


*Fig. 4.1* A schematic diagram of information flows in a typical classical computer. The state of classical bit can be copied to another classical bit with no restriction.

### 4.1.4. Gates

The computer can change the state of any bit as directed by a program. A simple operation is to flip a bit, that is $0 \rightarrow 1$ and $1 \rightarrow 0$. We express it using a concept of *gate*. A bit enters a gate and comes out with a different value. In the classical computer, there are only two possibility, no flip or flip. “No flip” means nothing happens to the bit and thus we are not interesting it. The gate that flips a bit is called `NOT` gate. (See Fig. 4.2.) This is the only gate acts on a single bit outputs a single bit. The relation between the input and the output bit is given in the truth table 4.1.

| $i$ | $o$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

*Table 4.1*
NOT Gate

Classical computation uses various kind of gates which takes two input bits ($i_0$ and $i_1$) and outputs one bit. For example, `AND` gate outputs $i_0 \times i_1$. (See Fig. 4.2.) The corresponding truth table is given in Table 4.2. Notice that this process is *irreversible*, meaning that we cannot recover the input information $i_0$ and $i_1$ from the output $o$. Hence, we says that a bit of information is lost. Two-bit classical gates are in general irreversible and 1 bit of information is lost. However, this is not a major issue since classical bits can be cloned and saved in a separate bit string.

In principle, classical computation needs only $\texttt{NOT}$ and $\texttt{AND}$ gates. It is using other gates makes classical computing much more efficient.

| $i_0$ | $i_1$ | $o$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

*Table 4.2* AND Gate

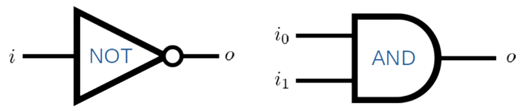


*Fig. 4.2* An example of classical gates. The $\texttt{NOT}$ gate takes one bit and output one bit.

### 4.1.5. Landauer principle

The irreversibility of classical gates has important physical consequence. *Landauer's principle* states that the loss of one bit of information necessarily dissipates $k_B T \log 2$ of heat. For example, every time an $\texttt{AND}$ gate is used, at least $k_B T \log 2$ of heat must be generated. This heat is actually minute and much smaller than Joule heat that makes CPU really hot. Therefore, it odes not causes a significant problem. However, it imposes a theoretical limit of classical computation. In contrast, quantum computing is reversible and thus no dissipation is mandatory.

**Exercise** Is $\texttt{NOT}$ gate reversible?

### 4.1.6. Encoding

The computer can understand only bit strings and manipulates them by applying gates. However, problems we want to solve are usually not expressed in bit strings. Somehow, we need to map the problems to bit strings the computer can understand. In other words, we need to *encode* the information in our expression to the bit strings. As a simple example, we consider a boolean operation between two logical variables $x$ and $y$. Their value is either $\texttt{True}$ or $\texttt{False}$. Since the valiables takes only one of two possibilities, the mapping between the boolean variables and bits are naturally

$$\texttt{False} \Rightarrow \texttt{0}, \quad \texttt{True} \Rightarrow \texttt{1}$$

Now, variable $x$ is expressed by a bit $b_0$ and $y$ by $b_1$.

In many problems, encoding is not obvious at all. Integer numbers can be mapped to bit strings using their binary expressions. For example, $5 \Rightarrow \texttt{101}$, which requires three bits. Exact encoding of continuous numbers is not possible since bits are digital. Approximated representation such as *floating point arithmetic* numbers are used.

### 4.1.7. Instructions

Computation means that by manipu, quantum or classicallating the state of bits toward the solution. The device can flip the specified bits . It can be thought of bits entering a *gate* and coming out with different values. We need to instruct the device how to change the state of the bits. In other words we to find find what gates should be applied to the bits. Let us consider a simple operation $x+y \mod 2$ where $x, y \in \{0,1\}$. WE shall write this operation simply as $x \oplus y$ . We map $x$ and $y$ to bits $b_0$ and $b_1$. Applying an `XOR` gate on $q_0$ and $b_1$ produces desired outputs. The following diagram classically computes $x \oplus y$ and the output is stored in $x$. The value of $y$ remains the same.

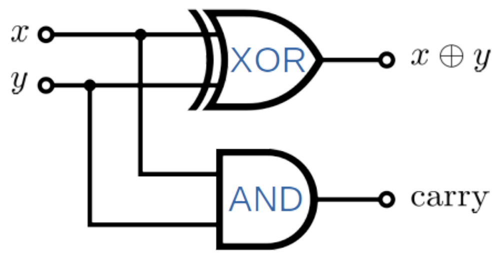


*Fig. 4.3* An example of classical circuit. This circuit computer $x \oplus y$ using $\texttt{XOR}$ and $\texttt{AND}$. In addition to the answer, carry over bit

In real world applications, we don't apply the gates by ourselves. Instead, we use a programming language such as `python`. The language translates the mathematical expression such as $x+y$ into gate instructions. So, we really don't need to know about the classical gate. Quantum computation has not reached that level yet. Users must picks appropriate gates by themselves.

### 4.1.8. Readout and decoding

Once the computation is completed at the device level, we need to read out the values of the bits and decode the outcome to an integer. The readout of the classical bits can be done without damaging their state. The value of the classical bits remains the same before and after the readout. (That is not the case in quantum computers.)

## 4.2. Quantum computation

Here is a brief description of quantum computer and the details will be discussed in this book. It is necessary for you to understand the following at this stage. Just read them and move on.

### 4.2.1. Qubits

Quantum computers are also made of many small building blocks but unlike classical computers we need to use quantum mechanics to describe their state. Current quantum computers use the smallest quantum system, that is a two-dimensional Hilbert space $\mathbb{C}_2$ spanned by complex vectors $|0\rangle$ and $|1\rangle$, known as *qubit*. These two basis states looks similar to `0` and `1` for a classical bit . However, unlike the classical bit, the state of a qubit can be in a superposition state

$$|\psi\rangle = c_0|0\rangle + c_1|1\rangle \tag{4.1}$$

where $c_0$ and $c_1$ are complex number satisfying $|c_0|^2 + |c_1|^2 = 1$ (normalization). Similarly to the classical bit, we obtain either $|0\rangle$ or $|1\rangle$ when the superposition state is measured. We shall call them *computational basis*. However, the state of the qubit is neither $|0\rangle$ nor $|1\rangle$ but $|0\rangle$ AND $|1\rangle$ simultaneously. Hence, a qubit can compute two different cases at once (if a programmer is smart enough). Quantum computers exploit this superposition state. Unlike the classical bit which has only two possible states, a qubit can take infinitely many different states since $c_0$ and $c_1$ can be any complex numbers satisfying the normalization condition.

Obviously, quantum computers are made of many qubits as a composite system. Let us consider two qubits, $q_0$ and $q_1$. A composite system of two classical bits can have four different states, ‘00’, ‘01’, 10’, and ‘11’. Similarly the pair of qubits are spanned by four basis vectors $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$. Unlike the classical bits, the qubits can be in a superposition state

$$|\psi\rangle = c_{00}|00\rangle + c_{01}|01\rangle + c_{10}|10\rangle + c_{00}|11\rangle$$

The four different possibilities are simultaneously considered in the superposition state. The complexity of the superposition state grows very rapidly as the number of qubits increases. If there are $n$ qubits, the number of terms in the superposition state can be as many as $2^n$. Then, $2^n$ different cases are simultaneously computed. We will take the advantage in various applications.

In classical computers, a bit string $b_{n-1}\, b_{n-2}\, \cdots\, b_1\, b_0$ expresses the state of the system. In quantum computer, a tensor product $|q_{n-1}\rangle \otimes |q_{n-2}\rangle \otimes \cdots \otimes |q_1\rangle \otimes |q_0\rangle$ represents the state of $n$ qubits. For two qubits, $|01\rangle = |0\rangle \otimes |1\rangle$.

### 4.2.2. Restriction on quantum information processes

Qubits have many advantages over classical bits, which is the main topics of this book. However, they have also many disadvantages. Here are some restrictions imposed on the quantum information.

1. **No-cloning theorem**
   The theorem states that it is not possible to duplicate an arbitrary qubit. During quantum computation we cannot make a copy of a qubit in an unknown state. Recall that classical bit can be cloned.
2. **No-teleportation theorem**
   If the information stored in a qubit can be converted to a classical bit string and vice versa, we can *teleport* the quantum information to a distant place via a classical bit string. Such teleportation is not possible. This theorem imposes more important restriction. We cannot read out the full information in an arbitrary qubit since the outcome of the measurement is classical.
3. **No-deleting theorem**
   If two qubits happened to be in the same but arbitrary state, it is not possible to delete the information in one of the qubits. This is a no-go theorem and time-reversal of no cloning theorem.

4. **No-broadcasting theorem**
   It is possible to transfer the full information in an arbitrary qubit to another qubit but the information in the original qubit must be destroyed. This is known as *quantum teleportation*. However, broadcasting the full information in an arbitrary qubit to multiple qubits is not possible. This is a corollary of no cloning theorem.
5. **No-hiding theorem**
   Briefly stating, the information stored in quantum system must be conserved. It is not possible to create or destroy quantum information. In contrast, classical information can be created or destroyed.

### 4.2.3. Quantum Information

Recall that information is about uncertainty or probability. a classical bit is dichotomous and there is only one uncertainty, 0 or 1. In contrast, the state of a qubit is continuous and there are infinitely many different states. The no-teleportation theorem tells us that even infinitely many classical bits cannot describe the state of a qubit. Does this mean a qubit contains infinite amount of information? It turns out that we can ask only a dichotomous question.

Consider a qubit in a superposition state

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

If we ask if the qubit is in $|0\rangle$, the answer is either “yes” or “no” but there is no definite answer. There is 50% chance to get “yes” and 50% chance to get “no”. The situation is quite similar to the coin tossing. However, there are differences. Even when we know precisely that the qubit is in the state $|\psi\rangle$, there is still the uncertainty, hence it is no due to our ignorance. The uncertainty is not due to future event since $|\psi\rangle$ already exists. Nevertheless, the amount of information seems $\log 2$.

You can ask a different question. Is the state $|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ or $|-\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$? $|\pm\rangle$ forms an orthonormal basis, this question makes a sense. Now, the answer is “yes” without ambiguity. For this question, $|\psi\rangle$ contains no information! Although the question we can ask is always dichotomous, we can ask infinitely many different questions. This is the main difference between qubit and classical bit.

We define quantum information entropy using density operator $\rho = |\psi\rangle\langle\psi|$ as

$$S = -\mathrm{Tr}\,(\rho \log \rho)$$

which is known as von-Neumann entropy. Plugin $\rho = |\psi\rangle\langle\psi|$ , we find $S = 0$. Hence, the state has no information (uncertainty). When quantum measurement is done, the state collapses to a different state, which can have a non-vanishing information entropy. In this sense, information is created, when measurement is done. For the current example, the state after measurement of $|0\rangle$ and $|1\rangle$, the state after the measurement is

$$\rho = \frac{1}{2}|0\rangle\langle 0| + \frac{1}{2}|1\rangle\langle 1|$$

and its von Neumann entropy is $S = \log 2$ as expected.

### 4.2.4. Gates

As a general purpose computer, a quantum computer must be able to change the state of qubits based on given instructions. Recall that a classical bit can only be flipped. In contrast, quantum computers can change the state of a qubit in infinitely many different ways. Here we use also the concept of gates. A qubit enters a gate and comes out in a different state. In other words, a state vector of the qubit is transformed to another state vector by the gate. In quantum mechanics, that is achieved by a unitary transformation $U$. The transformation $|\psi'\rangle = U|\psi\rangle$ in quantum mechanics is equivalent to applying a 1-qubit gate $U$ to a qubit and expressed as

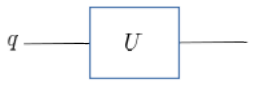


*Fig. 4.4* Action of one-qubit gate.

As an example, consider a Pauli operator $X \equiv \sigma_x$. When it acts on the computational basis, $X|0\rangle = |1\rangle$ and $X|1\rangle = |0\rangle$. Hence, it acts like `NOT` in the classical computation. See the Table 4.3. The qubit can be in a superposition state. The classical truth table is not enough. When it acts on a general state, we get

$$X|\psi\rangle = c_0 X|0\rangle + c_1 X|1\rangle = c_0|1\rangle + c_1|0\rangle = c_1|0\rangle + c_0|1\rangle \tag{4.2}$$

which actually swaps the coefficients. The first two rows in Table 4.3 is much like the classical truth table 4.1. The third row makes quantum computers more powerful than the classical computers.

| $i$ | $o$ |
|---|---|
| $\|0\rangle$ | $\|1\rangle$ |
| $\|1\rangle$ | $\|0\rangle$ |
| $c_0\|0\rangle + c_1\|1\rangle$ | $c_1\|0\rangle + c_0\|1\rangle$ |

*Table 4.3* X Gate

Like classical computers, quantum computers use gates that take two qubits as input. Two-qubit gates are also a unitary operator and express as

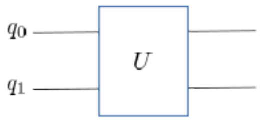


*Fig. 4.5* Action of two-qubit gate.

Logically, infinitely many 1-qubit and 2-qubits gates are possible but actual devices can understand only some of them. Depending on the underlying quantum systems, the available gates are different. However, almost any gate can be expressed equivalently with a set of other gates in principle (gate *decomposition*). So, if a desired gate is not available on the device you are using, you must find a decomposition of the gate. For IBM devices, `Qiskit` finds a decomposition suitable for them.

## 4.2.5. Encoding

Encoding the information of the problem to be solved to qubits in a quantum computer is not a trivial task. Let us try to use the same encoding scheme as the above classical case:

$$\texttt{False} \Rightarrow |0\rangle, \quad \texttt{True} \Rightarrow |1\rangle$$

This works fine. However, there are many other encoding schemes since a qubit can take a superposition state. For example,

$$\texttt{False} \Rightarrow |+\rangle \equiv \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad \texttt{True} \Rightarrow |-\rangle \equiv \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

is another choice.

To make a good use of quantum computer, finding a good encoding scheme is crucial.

## 4.2.6. Instructions

Once an encoding scheme is chosen, we need to give a set of instructions to a quantum computer. That is we find a set of unitary operators that are applied on qubits one after the other. Unlike classical computation, no advanced programming language is available for quantum computing. Therefore, programmers must right instructions the device can understand directly (similar to coding with machine or assembly language for classical computers.) A code for quantum computation looks like

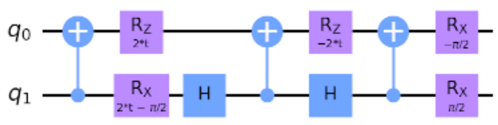


*Fig. 4.6* An example of quantum circuit.

which is known as quantum circuit. Each object in the circuit is a unitary gate.

The following circuit computes $x \oplus y$ using a quantum computer.

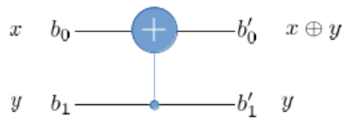


*Fig. 4.7* Quantum computation of $x \oplus y$.

The gate in the circuit is known as *controlled X* or *controlled NOT*. It applies $X$ on $q_0$ only when $q_1 = |1\rangle$. This is equivalent to the classical computation with `XOR` gate if input qubits are either $|0\rangle$ or $|1\rangle$. We can see the difference between classical and quantum computation when a superposition state is used as input. Consider a case where $q_0 = |0\rangle$ and $q_1 = c_0|0\rangle + c_1|1\rangle$. The state of the two qubits is

$$|\Psi_{\text{in}}\rangle = |0\rangle \otimes (c_0|0\rangle + c_1|1\rangle) = c_0|0\rangle \otimes |0\rangle + c_1|0\rangle \otimes |1\rangle.$$

The output is

$$|\Psi_{\text{out}}\rangle = c_0|0\rangle \otimes |0\rangle + c_1|1\rangle \otimes |1\rangle,$$

which is known as *entangled* state. There is no way to express this state in a classical computer. Quantum computation utilizes entangled states.

## 4.2.7. Readouts

When a qubit is in $|0\rangle$ or $|1\rangle$, readout is in principle accurate (not on current NISQ computers due to device errors). However, the outcome is completely uncertain if the qubit is in a superposition state (4.1). We will obtain `0` or `1` at random. If we prepare many qubits in the same superposition state, we obtain `0` from some and `1` from others. Hence, the readout of qubits is stochastic, from which we know that the qubits are in a superposition state but which one? The principle of quantum mechanics tells us that the probability of getting `0` is given by $|c_0|^2$ and `1` by $|c_1|^2$. By measuring many qubits in the same state, we can calculate the probabilities from which $|c_0|^2$ and $|c_1|^2$ can be determined. The phase of $c_0$ and $c_1$ are still missing. There is a complicate procedure, known as *quantum tomography*, which can determine the superposition state if and only if sufficiently large number of the same superposition states are measured. Therefore, it is necessary to repeat the same quantum computation many times if the final result is a superposition state. Even when the result of the computation is designed to be $|0\rangle$ or $|1\rangle$ by the algorithm, the outcome can be a superposition state due to device errors. Even worse, the final state can be so-called *mixed state*, which contains a mixture of classical and quantum uncertainties simultaneously. A good quantum circuit avoids both classical and quantum uncertainties as much as possible.

# 5. Single qubit

In quantum computer, information is processed in a set of qubits. Correlation among the qubits distinguishes the quantum computation from the classical computation. However, understanding the properties of a single qubit is the first step toward the quantum computation.

Mathematically, a qubit lives in the smallest Hilbert space, that is a two-dimensional Hilbert space. Physically, a qubit is realized in many different form. Any quantum system that has only two states or two states well separated from other states is a good candidate of a qubit. For example, a photon has two directions of linear polarization, the horizontal and vertical polarization. Hence, a single photon can be used as a qubit. Various different types of qubits are developed, including Superconducting quantum computing and Trapped ion quantum computer. Regardless its physical implementation, the same mathematical theory can be used for any type of a qubit.

In this chapter, first mathematical expressions and properties of a single qubit are explained. It is very important for you to familiarize yourself with the notations and the rules of calculation since they are used through the book. Secondly, the concept of quantum measurement is introduced. As we discussed in the previous chapter, the quantum measurement is quite different from the classical measurement. You must get rid of the classical intuition and believe in the principles of quantum mechanics.

After the theory of a qubit, I explain how to create a qubit in Qiskit and how to visualize the state of the qubit. At the end, we simulate quantum coin tossing and quantum state tomography using a quantum computer.

## 5.1. Pure states

Quantum computers store information in a two-dimensional Hilbert space called a qubit. In this section, I will introduce mathematical description of a single qubit.

### 5.1.1. Pure states

In ideal situations, the state of a qubit is specified by a single ket vector in a two-dimensional Hilbert space. We call the ket *state vector* and the ideal state *pure state*. Unless otherwise is specified, the state vectors are assumed to be normalized. That is $\langle\psi|\psi\rangle = 1$ (See Section 3.1). In less ideal situations, a qubit is in a *mix state* involving multiple pure states, which will be discussed in a later chapter. In this chapter, we focus on pure states.

Any qubit state can be written as

$$|\psi\rangle = c_0|0\rangle + c_1|1\rangle \tag{5.1}$$

where $|0\rangle$ and $|1\rangle$ are orthonormal basis vectors satisfying $\langle 0|0\rangle = \langle 1|1\rangle = 1$ and $\langle 0|1\rangle = \langle 1|0\rangle = 0$. The coefficients $c_0$ and $c_1$ are complex numbers. The normalization requirement is satisfied if $|c_0|^2 + |c_1|^2 = 1$.

This orthonormal complete basis set $\{|0\rangle, |1\rangle\}$ is known as *computational basis* which is the default basis in quantum computing. However, this basis set do not represent any particular physical basis set. In real quantum computers, a qubit can be realized, for example, by an electron spin, $|0\rangle \equiv |\uparrow\rangle$ and $|1\rangle \equiv |\downarrow\rangle$ or the polarization of a photon, $|0\rangle \equiv |H\rangle$ (horizontal polarization) and $|1\rangle \equiv |V\rangle$ (vertical polarization). By using the computational basis set, we don't have to worry about what type of hardware is used. The mathematical expression based on the computational basis can be applied to all of qubit-based quantum computers.

### 5.1.2. Standard Basis sets

In addition to the computational basis (also known as *z-basis*) $\{|0\rangle, |1\rangle\}$, we often use other orthonormal basis set, namely *x-basis* $\{|+\rangle, |-\rangle\}$ and *y-basis* $\{|L\rangle, |R\rangle\}$. They are related to the computational basis as follows:

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{5.2}$$

$$|L\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle), \quad |R\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle) \tag{5.3}$$

**Exercise** 5.1.1   Assuming that the computational basis is orthonormal, show that the x-basis and the y-basis are orthonormal.

### 5.1.3. The basis sets in Qiskit

In Qiskit, the computatioal basis kets $\{|0\rangle, |1\rangle\}$ and x-basis kets $\{|+\rangle, |-\rangle\}$ are predefined. See the following Qiskit note.

**Qiskit note: Computational Basis**

The `qiskit.opflow` library provides us with basic tools to describe quantum mechanics using expressions very similar to the original methematical expressions. Here are the correspondence between mathematical expressions and `opflow` expressions.

| | | |
|---|---|---|
| $\|0\rangle$ | $\Rightarrow$ | `Zero` |
| $\|1\rangle$ | $\Rightarrow$ | `One` |
| $\|+\rangle$ | $\Rightarrow$ | `Plus` |
| $\|-\rangle$ | $\Rightarrow$ | `Minus` |

The vector with “~” indicates bra. For example,

$\langle 0|$   $\Rightarrow$   `~One`

The regular addition works.

$|0\rangle + |1\rangle$   $\Rightarrow$   `Zero + One`

The operator for inner product is “@”. You need to evalute it with `.eval()`.

$\langle 0|+\rangle$   $\Rightarrow$   `(~Zero @ Plus).eval()`

`eval` is a method associated with `OperatorBase` class in `qiskit.opflow`.

For more detail, see Qiskit API Document: qiskit.opflow

**Qiskit Example** 5.1.1   Let us check the orthonormality of the computational basis set in Qiskit.

```python
# Import numpy (for sqrt)
import numpy as np

# Import the base vectors from qiskit.opflow library
from qiskit.opflow import Zero, One, Plus, Minus

# Example of inner product calculation with qiskit

# First example calculate the norm of |0> and |1>
# Zero and One are ket
# ~Zero and ~One are corresponding bra
# @ product between bra and ket, that is inner product
# To complete the operation @, we must evaluate it by .eval()

Norm_of_Zero = np.sqrt((~Zero @ Zero).eval())
Norm_of_One  = np.sqrt((~One @ One).eval())

# Show the reults
print("Norm of |0> =", Norm_of_Zero)
print("Norm of |1> =", Norm_of_One)

# Next we check the orthogonality between |0> and |1>
Inner_Product = (~Zero @ One).eval()

# Show the result.
print("<0|1> =", Inner_Product)
```

```
Norm of |0> = 1.0
Norm of |1> = 1.0
<0|1> = 0.0
```

**Qiskit Example** 5.1.2 Let us construct another orthonormal basis set $|L\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$ and $|R\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)$. Notice the complex unit $i$ on $|1\rangle$.

```python
# Let us try more complicated case.
# we will solve the above exercise problem using Qiskit

# import numpy
import numpy as np

# imaginary unit "i" is "1j" in python.
L = (Zero + 1j* One)/np.sqrt(2)
R = (Zero - 1j* One)/np.sqrt(2)

print( "<L|L> =",(~L@L).eval() )
print( "<R|R> =",(~R@R).eval() )
print( "<L|R> =",(~L @ R).eval() )

# Anser should be 0 since they are orthogonal.
```

```
<L|L> = (0.9999999999999998+0j)
<R|R> = (0.9999999999999998+0j)
<L|R> = 0j
```

**Python note: Complex numbers**

In `python`, a complex number is expressed as $2+3j$. Note that symbol $j$ is used instead of $i$. The imaginary unit $i$ is $1j$. Notice $1$ in from of $j$. $j$ must be used with a regular number. $j$ alone causes an error.

**Exercise** 5.1.2

1. Confirm that base vectors `Plus` and `Minus` are orthonormal using Qiskit.
2. Calculate the inner product between `One` and `Minus` by hand and confirm your answer using Qiskit.

**Exercise** 5.1.2 In the previous example, change the order of rotations to `rx`, `rz`, `ry`. Does it come back to the initial state?

Last modified: 07/09/2022

## 5.2. Bloch sphere

While Eq. (5.1) can describe any qubit state, there is a better expression suitable for visualizing the qubit state. Writing the complex coefficient in the polar expression, $c_0 = r_0 e^{i\phi_0}$ and $c_1 = r_1 e^{i\phi_1}$ where $r_i$ and $\phi_i$ are modulus and argument. Then, we remove a global phase as

$$|\psi\rangle = r_0 e^{i\phi_0}|0\rangle + r_1 e^{i\phi_1}|1\rangle = e^{i\phi_0}\left(r_0|0\rangle + r_1 e^{i(\phi_1-\phi_0)}|1\rangle\right) \simeq r_0|0\rangle + r_1 e^{i\theta}|1\rangle$$

where $\phi = \phi_1 - \phi_0,\ \phi \in [0, 2\pi)$ and "$\simeq$" means "equivalent up to global phase". Now, the normalization condition becomes $r_0^2 + r_1^2 = 1$. Since $r_0$ and $r_1$ are positive, we can write them as $r_0 = \cos\left(\frac{\theta}{2}\right)$ and $r_1 = \sin\left(\frac{\theta}{2}\right)$ with $0 < \theta < \pi$. Now the general qubit state is written as

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + \sin\left(\frac{\theta}{2}\right)e^{i\phi}|1\rangle, \quad 0 \le \theta \le \pi,\ 0 \le \phi < 2\pi. \tag{5.4}$$

This expression suggests that any pure state can be map to a point on the surface of a unit sphere using spherical coordinates $\theta$ and $\phi$. The north pole of the sphere ($\theta = 0$) corresponds to $|0\rangle$ and the south pole ($\theta = \pi$) to $|1\rangle$ (recall that the global phase $e^{i\phi}$ can be omitted. When $\theta = \frac{\pi}{2}$ and $\phi = 0$, we obtain $|+\rangle$ and when $\theta = \frac{\pi}{2}$ and $\phi = \pi$, we have $|-\rangle$.

The sphere is known as *Bloch sphere* and the arrow from the center of the sphere to the point on the surface is called *Bloch vector*. Each Bloch vector corresponds to a qubit state.

**Qiskit note: Bloch sphere**

Qiskit provides four tools to visualize the Bloch sphere and vector.

1. Use `plot_bloch_vector` in `qiskit.visuakization` if $\theta$ and $\phi$ are known.

```
plot_bloch_vector([r,theta,phi], coord_type='spherical', figsize=(w,h) )
```
API reference:plot_bloch_vector

1. Use `plot_bloch_multivector` if a state vector is known.

```
plot_bloch_multivector(state)
```
`state` is a quantum state. Various format is allowed, including Statevector class objects. For other optional arguments.
API reference:plot_bloch_multivector

1. Use `draw` method associated with `statevector` class

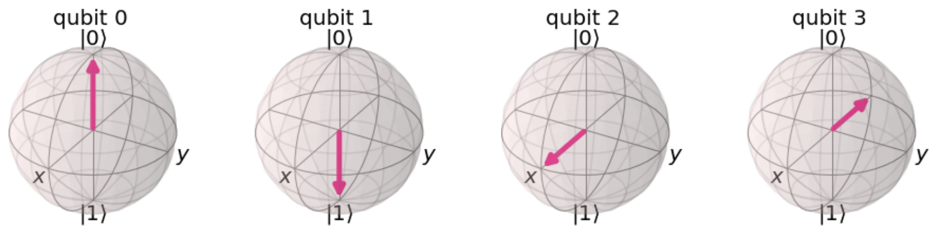
psi`.draw('mpl')`
where psi is a `staetvector` class object. `mpl` specifies the use of `matplotlib` module. This method is just a front end to plot-bloch-multivector.
API reference:Statevector.draw

1. We can even draw evolution of the Bloch vector as a movie using `visualize_transition`.

visualize_transition(qc,fpg=50, spg=1) where `qc` is a quantum circuit (I will explain it later.) See the following documentation for other parameters.
API reference:visualize_transition

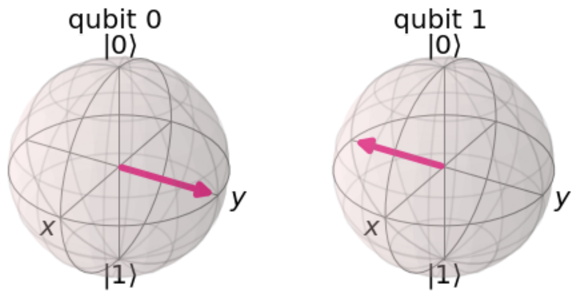
**Qiskit Example** 5.2.1 The following example plots the Bloch vectors of $|0\rangle$, $|1\rangle$, and $|\pm\rangle$. To plot all of them at once, we use a tensor product of all vectors. In Qiskit, tensor product is done by "^".

```
# import
# importhe visualization tool
from qiskit.visualization import plot_bloch_multivector

# impor
# import basis vectors
from qiskit.opflow import Zero, One, Plus, Minus

# show Bloch vector for |0>, |1>, |+>, and |->
plot_bloch_multivector(Minus^Plus^One^Zero)
```



**Qiskit Example** 5.2.4 Plot the Bloch vectors of the states $|L\rangle$ and $|R\rangle$ discussed in **Qiskit Example** 5.1.2.

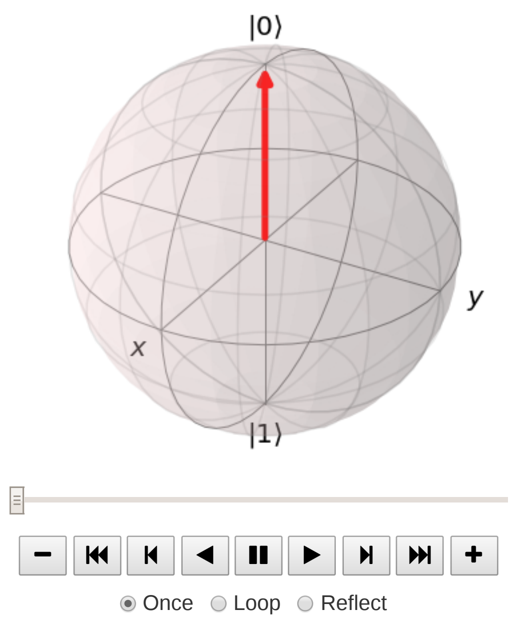
```
import numpy as np
# generate basis |L> and |R>
L = (Zero + 1j* One)/np.sqrt(2)
R = (Zero - 1j* One)/np.sqrt(2)
# show their Bloch vectors
plot_bloch_multivector(R^L)
```



**Qiskit Example** 5.2.3 A statevector is initially $|0\rangle$. It is rotated around $y$, $z$, and $x$ by angle $\frac{\pi}{2}$ for each rotation. It is done by rotation gates `ry`, `rz`, and `rx`. (They are introduced in next chapter.) Construction of quantum circuits will be discussed later in great detail. The Bloch vector should come back to the starting point. See the animation.

```
%%capture
# The output block is disabled.

# load numpy and qiskit
import numpy as np
from qiskit import *

# create an empty quantum circuit
qr = QuantumRegister(1)
qc = QuantumCircuit(qr)

# add rotation gates to the circuit
qc.ry(np.pi/2,0)
qc.rz(np.pi/2,0)
qc.rx(np.pi/2,0)

# load the visdualization tool
from qiskit.visualization import visualize_transition

# generate a movie (it will be shown in next cell,
movie=visualize_transition(qc,fpg=20, spg=1)
```

```
# Show the movie
movie
```



## 5.3. Qubit Measurement

Measuring a physical quantity in quantum mechanics is a bit weird. I am not going to present a full theory of measurement. If you are interested in it, read Ref. [1]. See also wikipedia.

### 5.3.1. Projective measurement

In classical computer, readout processes determine the outcome of computation stored in each bit. That is to determine if the state of each bit is $0$ or $1$. The readout can be done at any time without disturbing the state of the bit. If you repeat the same measurement, the same outcome is obtained.

Similarly, we want to know whether each qubit is in $|0\rangle$ or $|1\rangle$. However, if a qubit is in a superposition state, we have a big problem. Equation (5.1) indicates that the state of the qubit is neither $|0\rangle$ nor $|1\rangle$. Despite of it, quantum mechanics allows us to ask if the qubit is in $|0\rangle$ or $|1\rangle$ and surprisingly the answer is one of $|0\rangle$ and $|1\rangle$ even it is in the superposition state. Suppose that the outcome of the measurement is $|0\rangle$, it does not mean that the qubit was in $|0\rangle$. The measurement process has transformed $|psi\rangle$ to $|0\rangle$. This transition is referred as the *collapse of wavefunction*. Mathematically, we say that the state $|\psi\rangle$ is projected to $|0\rangle$ and thus it is called *projective measurement*. See Fig. Fig. 5.1. You may obtain $|1\rangle$ from the same superposition state. Then, $\psi\rangle$ is projected to $|1\rangle$. The measurement of a single qubit seems suggesting that the outcome is not related to the state of the qubit. We never be able to determine the superposition from the outcome. Then, what is the purpose of the measurement? Even worse, the measurement destroys the superposition state.

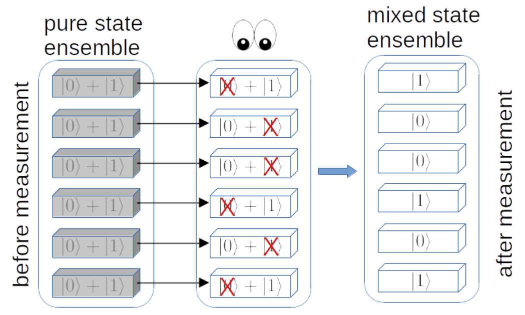


*Fig. 5.1* Measurement of a superposition state. Before measurement, every qubit in the ensemble is in the same superposition state (5.1). It is known as a purestate ensemble The measurement selects one of $|0\rangle$ and $|1\rangle$. As the result, some qubits in the ensemble are in $|0\rangle$ and others in $\backslash 1\rangle$. After the measurement, we have a mixed state ensemble.

### 5.3.2. Born rule

It seems that the outcome is picked at random. The theory of quantum mechanics is powerless if we measure a single qubit. So, what is the physical meaning of the state vector $|\psi\rangle$? A resolution was offered by Born. Consider an ensemble of qubits. All qubits in the ensemble are prepared exactly in the same state $|\psi\rangle$ and exactly the same measurement process is applied to them. Some of them are transformed to $|0\rangle$ and others to $|1\rangle$. It turns out that the probability to obtain $0\rangle$ is $|c_0|^2$ and that of $|1\rangle$ is $|c_1|^2$. This interpretation of the superposition state is known as *Born* rule. Since we will obtain one of them from each measurement, $|c_0|^2 + |c_1|^2 = 1$, which is satisfied by the normalization condition.

### 5.3.3. Quantum coin flipping

As an example of the Born rule, we simulate quantum coin flipping.

It is a common practice to pick one of two choices by tossing up a coin. The classical motion of coin is chaotic and the probability to get head and tail is approximately equal. We can simulate the process on a classical computer using a random number generator. Such simulation is known as Monte Carlo simulation after the name of the famous casino city. One can simulate the same process on a quantum computer without random number generator by exploiting the stochastic nature of quantum measurement. For comparison, both a classical and quantum simulation are presented below.

#### Classical simulation

Using `random.choice` function in numpy, we generate random choice of 0=head and 1=tail. Then, count the number of head and tail.

```
# Classical Monte Carlo simulation of coin tossing
import numpy as np

# coin is tossed 10000 times for statistical analysis
shots=10000

# generate random choice 0 or 1
# head=0 and tail=1
face = np.random.choice(2,size=shots)

# count number of heads and tails
# if the random number is less than 0.5, it is head.
# otherwise, it is tail.
head = np.sum(face==0)
tail = shots-head

# print out the probabilities
print("head=",head/shots)
print("tail=",tail/shots)
```

```
head= 0.4961
tail= 0.5039
```

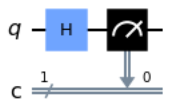
#### Quantum simulation

To demonstrate the stochastic nature of quantum measurement, we consider a quantum coin. The quantum coin has two states, head and tail. We assign the head to $|0\rangle$ and the tail to $|1\rangle$ The superposition state $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ indicates that the state of the quantum coin is neither head nor tail. Based on the Born rule, the chance to get head is 50% and tail 50%. Since the outcome is completely random, measuring the state of the quantum coin is equivalent to tossing a classical coin.

In the following qiskit code, the superposition state using a Hadamard gate (we will discuss this in next chapter) and then the state of the qubit is measured. The result is stored in a classical bit. The quantum computation is simulated with `qasm_simulator` on your computer instead of sending the code to a real quantum computer. (We will send this to real quantum computer in a later chapter.) We repeat the simulation 10000 times and find the probabilities to find head and tail. Since the number of samplings is finite, the result is not exactly 50%-50% but close to it.

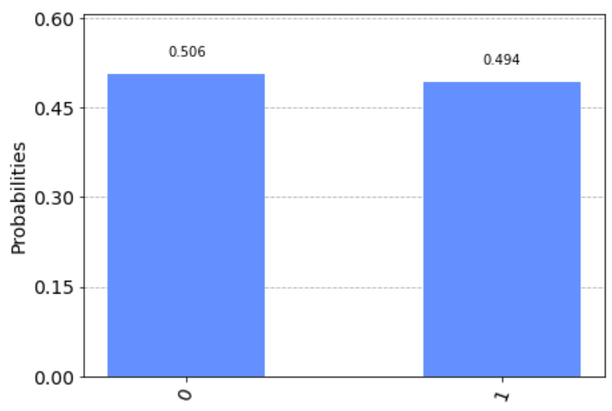
The quantum circuit shows two lines, one for quantum register and the other for classical register. The classical register contains a bit. Usually the outcome of measurement on a qubit is stored. The quantum register contains a qubit. It is initially reset to $|0\rangle$. The diagram shows that a single-qubit gate $H$ (Hadamard gate) is applied on the qubit. The outcome is $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. The next gate shows that the state of the qubit is measured and the result, either $0$ or $1$, is sent to the classical register. In the current example, $0$ or $1$ is obtained with the equal probability. First, we construct a quantum circuit.

```
# Tossing Up Quantum Coin

# import numpy
import numpy as np

# import entire qiskit
from qiskit import *

# set classical register (bit)
cr = ClassicalRegister(1,'c')

# set quantum register (qubit)
qr = QuantumRegister(1,'q')

# reset the quantum circuit
qc = QuantumCircuit(qr,cr)

# construct quantum circuit
# step 1: create the superposition state with Hardamard gate
qc.h(0)

# measurement on qubit
# output is stored in cllasical register
qc.measure(qr,cr)

# show the quantum circuit
qc.draw('mpl')
```



Now we execute the circuit using a simulator of quantum computer.

```
# Chose a general quantum simulator without noise.
# The simulator behaves as an ideal quantum computer.

backend = Aer.get_backend('qasm_simulator')

# Execute the quantum circuit 10000 times
job = backend.run(qc,shots=10000)

# get the result
result = job.result()

# Count the outcome
# Numbers of |0> and |1>
counts = result.get_counts()

# Visualize the outcome
# import histgram plotting function
from qiskit.visualization import plot_histogram
plot_histogram(counts)

# The result should be close to 0.5 for both |0> and |1>.
```



The outcome is not exactly 50%-50%. This is because the simulation is executed with a finite shots and thus there is statistical error due to finite sampling. Theoretically infinite sampling is required but practically it is not possible. However, this is not an issue of quantum computation.

## 5.4. Mixed states

TBW

# 6. One-qubit gates

In quantum computing, we manipulate the state of qubits by applying a series of gates on qubits. In this chapter, I introduce various one-qubit gates commonly used in quantum computing. When a one-qubit gate acts on a single qubit, the Bloch vector of the qubit rotates around a certain axis by a certain angle. The most general rotation is *Euler rotation*, which is done by $U$ gate. In theory, we need only this gate. However, it is not convenient since we have to figure out three parameters (angles) for each operation. Furthermore, in actual quantum computing devices, the general gate may not be available. In practice, we use parameter-free gates with predefined rotation axes and angles, and a few others that require a single parameter.

Mathematically, the gates are unitary operators in $\mathbb{C}^2$. Some are Hermitian and others are not. Table 6.1 lists commonly used single-qubit gates. I explain some the most important ones in the following section. See also Qiskit documentation:

Qiskit Circuit Library.

| Generaic Name | Qiskit Ciruit Name | # of Parameters | Symbols |
|---|---|---|---|
| Unitary | u | 3 | $U$ |
| Rotation X | rx | 1 | $R_x$ |
| Rotation Y | ry | 1 | $R_y$ |
| Rotation Z | rz | 1 | $R_z$ |
| Pauli X | x | 0 | $X$ |
| Pauli Y | y | 0 | $Y$ |
| Pauli Z | z | 0 | $Z$ |
| Hadamard | h | 0 | $H$ |
| Sqrt X | sx | 0 | $SX$ |
| Inverse sqrt X | sxdg | 0 | $SX^\dagger$ |
| Sqrt Z | s | 0 | $S$ |
| Inverse sqrt Z | sdg | 0 | $S^\dagger$ |
| 4th root Z | t | 0 | $T$ |
| Inverse 4th root Z | tdg | 0 | $T^\dagger$ |
| Phase | p | 1 | $P$ |

*Table 6.1* Commonly used single-qubit gates

Their definition and typical usage of those gates are explained in the following sections.

## 6.1. Mathematical expressions

There are several ways to define the gates. In the following sections, the gates are defined in four different ways but they are mathematically all equivalent. In one definition, a gate is defined by how the computational basis kets are transformed by the gate. This definition is practical. For computational purpose, the matrix expression of the gate is more convenient. The matrix expression assumes the computational basis. Since every gate is a special case of the general $U$ gate, the gate can be expressed with $U$, which is the third definition.

## 6.2. Transformation of general superposition states

The definitions do not give you a clear idea on the operational functionality of a gate. It is important to understand how a superposition state is transformed by the gate. Pay attention to how the coefficients are transformed.

## 6.3. Combination of gates

When gate $Y$ is applied after $X$, we write it $Y \cdot X$. We can think of a gate $(Y \cdot X)$ acts on a state vector as

$$Y(X|\psi\rangle) = (Y \cdot X)|\psi\rangle$$

We put "·" between the gates to avoid confusion. For example $SX$ is a single gate and $S \cdot X$ is a product of $S$ and $X$.

The order is important. $Y \cdot X$ is not necessarily equal to $X \cdot Y$.

In quantum circuit, the gates act from left to right. For example, $Y \cdot X|0\rangle$ becomes

```
from qiskit import QuantumCircuit

qc=QuantumCircuit(1)
qc.x(0)
qc.y(0)
qc.draw()
```

```
   ┌───┐┌───┐
q: ┤ X ├┤ Y ├
   └───┘└───┘
```

Last modified: 07/09/2022

## 6.4. U Gate

When a gate acts on a qubit, it rotates the Bloch vector. Any rotation can be specified with three angles, $\theta$, $\phi$, and $\lambda$. They are known as *Euler angles*. In standard quantum computation, a standard Euler rotation $z-y-z$ is used. That is rotating 1) around $z$ axis by $\lambda$, 2) around $y$ axis by $\theta$, and 3) around $z$ axis by $\phi$. The rotations must be done in this order. This gate transforms $|0\rangle$ to a general qubit state.

Qiskit API References: UGate

### 6.4.1. Definition

**Transformation**

$$\begin{aligned} U(\theta,\phi,\lambda)|0\rangle &= \cos(\theta/2)|0\rangle + \sin(\theta/2)e^{i\phi}|1\rangle \\ U(\theta,\phi,\lambda)|1\rangle &= -e^{i\phi}\left[\sin(\theta/2)|0\rangle - \cos(\theta/2)e^{i\lambda}|1\rangle\right] \end{aligned}$$

Notice that when it acts on $|0\rangle$, we obtain a general state in the Bloch sphere expression (5.4) (no $\lambda$ appears in it).

**Matrix expression**

$$U(\theta,\phi,\lambda) \doteq \begin{bmatrix} \cos(\theta/2) & -e^{i\phi}\sin(\theta/2) \\ e^{i\lambda}\sin(\theta/2) & e^{i(\lambda+\phi)}\cos(\theta/2) \end{bmatrix}$$

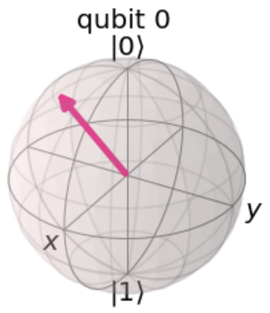
**Using Rotation gates**

The $U$ gate used to be defined as $U(\theta,\phi,\lambda) = R_z(\phi)R_y(\theta)R_z(\lambda)$. (See OpenQASM 2.) However, it adds a global phase on the Bloch sphere expression. To remove the inconvenience, now it is officially defined in OpenQASM 3 as

$$U(\theta,\phi,\lambda) = e^{i(\lambda+\phi)/2}R_z(\phi)R_y(\theta)R_z(\lambda).$$

**Qiskit Example** 6.4.1  Generate a state $\cos(\pi/3)|0\rangle + \sin(\pi/3)e^{-i\pi/7}|1\rangle$ in a quantum circuit.

```
# import numpy
import numpy as np

# import QuatumCircuit and QuantumRegister classes.
from qiskit import QuantumCircuit, QuantumRegister

# import STatevector class
from qiskit.quantum_info import Statevector

# Preparation
qr=QuantumRegister(1,'q') # create a single qubit with name 'q'.
qc=QuantumCircuit(qr)  # create a quantum circuit

#  apply the Ugate to the qubit
qc.u(np.pi/5,-np.pi/3,0,0)

# Plot the final state
Statevector(qc).draw('bloch')
```



## 6.4.2. Acting on a superposition state

$$ U(\theta,\phi,\lambda)\left(c_0|0\rangle+c_1|1\rangle\right) =\left[c_0 \cos(\theta/2) - c_1 e^{i\phi} \sin(\theta/2)\right]|0\rangle

- e^{i \phi} \left[ c_0 \sin(\theta/2) + c_1 e^{i \lambda}\cos(\theta/2) \right]|1\rangle $$

## 6.5. Rotation Gates, RX, RY, RZ

The rotation gates $R_x(\theta)$, $R_y(\theta)$, and $R_z(\theta)$ rotate the vector by $\theta$ around $x$, $y$, and $z$ axis, respectively. All rotation gates require one parameter $\theta$.

Qiskit circuit names are `rx`, `ry`, and `rx`, respectively.

API References: RXGate

API References: RYGate

API References: RZGate

## 6.5.1. Definition

**Transformation**

$$\begin{aligned} R_x(\theta)|0\rangle &= \cos(\theta/2)|0\rangle - i\sin(\theta/2)|1\rangle \\ R_x(\theta)|1\rangle &= -i\sin(\theta/2)|0\rangle + \cos(\theta/2)|1\rangle \end{aligned}$$

$$\begin{aligned} R_y(\theta)|0\rangle &= \cos(\theta/2)|0\rangle - \sin(\theta/2)|1\rangle \\ R_y(\theta)|1\rangle &= \sin(\theta/2)|0\rangle + \cos(\theta/2)|1\rangle \end{aligned}$$

$$\begin{aligned} R_z(\theta)|0\rangle &= e^{-i\theta/2}|0\rangle \\ R_z(\theta)|1\rangle &= e^{i\theta/2}|1\rangle \end{aligned}$$

**Pauli expressions**

$$\begin{aligned} R_x(\theta) &= e^{-iX\theta/2} = \cos(\theta/2)I - i\sin(\theta/2)X \\ R_y(\theta) &= e^{-iY\theta/2} = \cos(\theta/2)I - i\sin(\theta/2)Y \\ R_z(\theta) &= e^{-iZ\theta/2} = \cos(\theta/2)I - i\sin(\theta/2)Z \end{aligned}$$

**Matrix expressions**

$$\begin{aligned} R_x(\theta) &\doteq \begin{bmatrix} \cos(\theta/2) & -i\sin(\theta/2) \\ -i\sin(\theta/2) & \cos(\theta/2) \end{bmatrix} \\ R_y(\theta) &\doteq \begin{bmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{bmatrix} \\ R_z(\theta) &\doteq \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix} \end{aligned}$$

These gates rotates the Bloch vector arounf $x$, $y$, and $z$ axis by $\theta$, respectively. They appear in quantum circuit as The qiskit circuit symbols are `rx`, `ry`, and `rz`, respectively and they appear in quantum circuit as

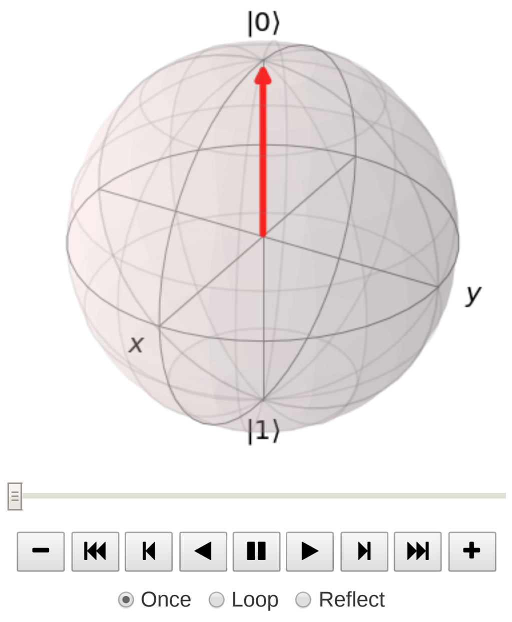
```
from qiskit.circuit import QuantumCircuit, Parameter

# set parameter symbol to theta
t=Parameter('\u03B8')

# costruct the circuit
qc=QuantumCircuit(1)
qc.rx(t,0)
qc.ry(t,0)
qc.rz(t,0)

# show the circuit
qc.draw()
```

```
   ┌───────┐┌───────┐┌───────┐
q: ┤ Rx(θ) ├┤ Ry(θ) ├┤ Rz(θ) ├
   └───────┘└───────┘└───────┘
```

**Example** 6.5.1 Starting with $|0\rangle$, rotate about the $y$ axis by $\pi/3$, about $z$ axis by $\pi/2$, and about $x$ axis by $-2\pi/3$. This example shows that the final state is $|1\rangle$.

```
%%capture
import numpy as np
from qiskit.circuit import QuantumCircuit
from qiskit.quantum_info import Statevector

qc=QuantumCircuit(1)
qc.ry(np.pi/3,0)
qc.rz(np.pi/2,0)
qc.rx(-2*np.pi/3,0)

# load the visdualization tool
from qiskit.visualization import visualize_transition

# generate a movie (it will be shown in next cell,
movie=visualize_transition(qc,fpg=20, spg=1)
```

```
movie
```



## 6.5.2. Additional useful Properties

In the following $R$ represent any of $RX$, $RY$, and $RZ$.

1. $R^{\dagger}(\theta) = R^{-1}(\theta) = R(-\theta)$
2. $R^{\alpha}(\theta) = R(\alpha\theta)$
3. $R(\theta_2) \cdot R(\theta_1) = R(\theta_1 + \theta_2)$ (note: all rotations must be around the same axis.)

## 6.5.3. Relation with other gates

By definition a one-qubit gate transforms a Bloch vector to another, which is a rotation of the Bloch vector. Hence, Any one-qubit gate can be expressed as rotation. In turn, any rotation can be expressed by a combination of rotations. Hence, any one-qubit gate can be expressed with a comination of RXGate, RYGate, and RZGate. Mathematically, these three gates are enough to describe quantum computation. However, the combination of rotation gates are not necessarily the most efficient implementation of gates. Parameter-free gates are still preferred.

The previous gates are related to the rotation as

- $X = iRX(\pi) \simeq RX(\pi)$
- $Y = iRY(\pi) \simeq RY(\pi)$
- $Z = iRZ(\pi) \simeq RZ(\pi)$
- $H = X \cdot Y^{1/2} \simeq RX(\pi)RY(\pi/2)$
- $S = Z^{1/2} \simeq RZ(\pi/2)$
- $T = Z^{1/4} \simeq RZ(\pi/4)$
- $SX = X^{1/2} \simeq RX(\pi/2)$

Last modified

## 6.6. XGate

This is a Pauli operator and one of the most important one-qubit gates. Hence, this section covers its properties at depth.

Qiskit API References: XGate

### 6.6.1. Definition

**Transformation**

$$\begin{aligned} X|0\rangle &= |1\rangle \\ X|1\rangle &= |0\rangle \end{aligned} \tag{6.1}$$

$X$ gate flips the computational basis, which resembles to the $NOT$ gate for classical computation. However, when it acts on superposition states, the state does not flip (the Bloch vector does not inverted).

**Matrix expression**

$$X \doteq \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{6.2}$$

**U gate expression**

$$X = U(\pi, 0, \pi) \tag{6.3}$$

**R gate expression**

$$X = iR_x(\pi)$$

The qiskit circuit symbol is `x` and it appears in quantum circuit as

```python
from qiskit import QuantumCircuit
qc=QuantumCircuit(1)
qc.x(0)
qc.draw()
```

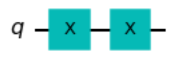
```
   ┌───┐
q: ┤ X ├
   └───┘
```

**Qiskit Example** 6.6.1

We construct a short quantum circuit using two Xgates and check how the state vector is transformed. The initial state is always $|0\rangle$. The first Xgate flips it to $|1\rangle$ and the second Xgate flips it back to $|0\rangle$.

```python
# import QuatumCircuit and QuantumRegister classes.
from qiskit import QuantumCircuit, QuantumRegister

# import STatevector class
from qiskit.quantum_info import Statevector

# Preparation
qr=QuantumRegister(1,'q') # create a single qubit with name 'q'.
qc=QuantumCircuit(qr)  # create a quantum circuit

# Intial state
psi0 = Statevector(qc)

#  apply the first Xgate to 0-th qubit
qc.x(0)

# Intermediate state
psi1 = Statevector(qc)

#  apply the second Xgate to 0-th qubit
qc.x(0)

# Final state
psi2 = Statevector(qc)

# Format ket vector with LaTeX.
ket0 = psi0.draw('latex')
ket1 = psi1.draw('latex')
ket2 = psi2.draw('latex')

# Show the result using display function
from IPython.display import display, Math
display("Quantum circuit",qc.draw('mpl'),"State vector before the gate",
        ket0,"State vector after the first Xgate",ket1,
        "State vector after the second Xgate",ket2)
```

```
'Quantum circuit'
```



```
'State vector before the gate'
```

$$|0\rangle$$

```
'State vector after the first Xgate'
```

$$|1\rangle$$

```
'State vector after the second Xgate'
```

$$|0\rangle$$

## 6.6.2. Acting on a superposition state

When XGate is applied to a super position state the coefficient is swapped. That is

$$X\left(c_0|0\rangle + c_1|1\rangle\right) = c_1|0\rangle + c_0|1\rangle \tag{6.4}$$

**Exercise** 6.6.1  Prove Eq. (6.4).

**Qiskit Example** 6.6.1  How do the following superposition states transformed by the Xgate?

$$|L\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle), \quad |R\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)$$

Using the general result (6.4), we find

$$X|L\rangle = \frac{1}{\sqrt{2}}(i|0\rangle + |1\rangle) = i|R\rangle, \quad X|R\rangle = \frac{1}{\sqrt{2}}(-i|0\rangle + |1\rangle) = -i|L\rangle$$

Recalling that the global phase factor can be ignored, we conclude that

$$X|L\rangle \simeq |R\rangle, \quad X|R\rangle \simeq |L\rangle.$$

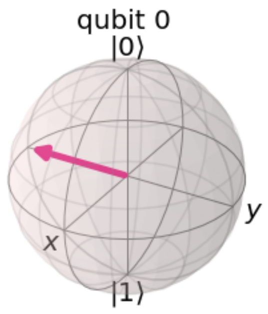
The following Qiskit code demonstrates that mathematically $X|L\rangle = i|R\rangle$ but $X|L\rangle$ \simeq |R\rangle$ when plotted in the Bloch sphere.

```
# import QuatumCircuit and QuantumRegister classes.
from qiskit import QuantumCircuit, QuantumRegister
from qiskit.opflow import Zero, One

# import STatevector class
from qiskit.quantum_info import Statevector

# import numpy
import numpy as np


L=(Zero+1j*One)/np.sqrt(2)
R=(Zero-1j*One)/np.sqrt(2)

# Preparation
qr=QuantumRegister(1,'q') # create a single qubit with name 'q'.
qc=QuantumCircuit(qr)  # create a quantum circuit

# set the qubit to |L>
qc.initialize(L.to_matrix())

# apply Xgate
qc.x(0)

# Final state
final=Statevector(qc)

# Show that the final state is not exactly the same as |R>
final.draw('latex')
```

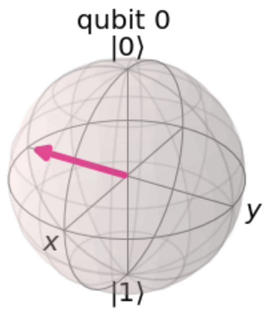
$$\frac{\sqrt{2}i}{2}|0\rangle + \frac{\sqrt{2}}{2}|1\rangle$$

```
# Compare the final state with |R> in Bloch sphere.
from qiskit.visualization import plot_bloch_multivector

# Generate Bloch vectors
R_bloch = plot_bloch_multivector(R)
final_bloch = plot_bloch_multivector(final)

from IPython.display import display

# Compare X|L> and |R>.  They are equivalent in the Bloch sphere.
display("Original |R>",R_bloch,"X|L>",final_bloch)
```

```
'Original |R>'
```



```
'X|L>'
```



**Exercise** 6.6.2 Show that $X|+\rangle = |+\rangle$ and $X|-\rangle = -|-\rangle$. This means that $X$ does not change $|\pm\rangle$ except for the phase factor.

### 6.6.3. Important Properties

$$X^2 = I$$

This means that

1. $X^2$ does not do any thing on the qubit.
2. $X$ is self-inverse, that is $X^{-1} = X$.
3. $X$ is self-adjoint ($X^\dagger = X$) since $X$ is unitary ($X^\dagger = X^{-1}$) by definition.

Property 1 was deomnstrated in Qiskit Example 6.6.1.

Last modified: 07/09/2022

## 6.7. YGate

This gate is not popular since it can be replaced with other common operators. Hence, this section only briefly covers its properties.

API References: YGate

### 6.7.1. Definition

**Transformation**

$$Y|0\rangle = i|1\rangle, \qquad Y|1\rangle = -i|0\rangle$$

**Matrix expression**

$$Y \doteq \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \tag{6.5}$$

**U gate expression**

$$Y = U\left(\pi, \frac{\pi}{2}, \frac{\pi}{2}\right) \tag{6.6}$$

**R gate expression**

$$Y = iR_y(\pi)$$

The qiskit circuit symbol is `y` and it appears in quantum circuit as

```python
from qiskit import QuantumCircuit
qc=QuantumCircuit(1)
qc.y(0)
qc.draw()
```

```
q: ┤ Y ├
```

## 6.7.2. Acting on a superposition state

When Ygate is applied to a super position state the coefficient is swapped. That is

$$Y\left(c_0|0\rangle + c_1|1\rangle\right) = -i(c_1|0\rangle - c_0|1\rangle) \tag{6.7}$$

If the global phase factor is omitted, we have

$$Y\left(c_0|0\rangle + c_1|1\rangle\right) \simeq c_1|0\rangle - c_0|1\rangle \tag{6.8}$$

Like Xgate, the coefficients are swapped. In addition the relative phase of $\pi$ is applied.

## 6.7.3. Important Properties

$$Y^2 = I$$

This means that

1. $Y^2$ does not do any thing on the qubit.
2. $Y$ is self-inverse, that is $Y^{-1} = Y$.
3. $Y$ is self-adjoint ($Y^\dagger = Y$) since $Y$ is unitary ($Y^\dagger = Y^{-1}$) by definition.

**Exercise** 6.7.1 Show that $Y|+\rangle = -i|-\rangle$ and $Y|-\rangle = i|+\rangle$. Apart from the phase factor, $Y$ flips $|\pm\rangle$.

Late modified xxx

## 6.8. ZGate

This gate is popular since it reverses the relative phase.

API References: ZGate

### 6.8.1. Definition

**Transformation**

$$Z|0\rangle = |0\rangle, \qquad Z|1\rangle = -|1\rangle$$

$Z$ gate preserves $|0\rangle$ but flips the phase of $|1\rangle$. This is a phase gate with prefixed phase change, “-1”.

**U gate expression**

$$Z = U\left(0, \frac{\pi}{2}, \frac{\pi}{2}\right) \tag{6.9}$$

**R gate expression**

$$z = iR_z(\pi)$$

The Qiskit circuit symbol is `z` and it appears in quantum circuit as

The standard symbol is $Z$ and it appears in quantum circuit as

```
from qiskit import QuantumCircuit
qc=QuantumCircuit(1)
qc.z(0)
qc.draw()
```

```
   ┌───┐
q: ┤ Z ├
   └───┘
```

### 6.8.2. Acting on a superposition state

When ZGate is applied to a super position state the relative phase changes by $\pi$. That is

$$Z\left(c_0|0\rangle + c_1|1\rangle\right) = c_0|0\rangle - c_1|1\rangle) \tag{6.10}$$

In Bloch sphere representation,

$$Z\left(\cos(\theta/2)|0\rangle + \sin(\theta/2)e^{i\phi}|1\rangle\right) = \cos(\theta/2)|0\rangle + \sin(\theta/2)e^{i(\phi+\pi)}|1\rangle$$

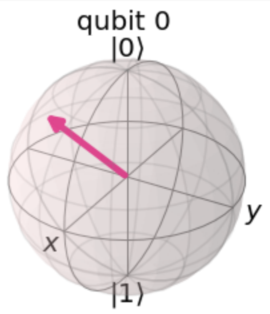
This suggests that the Bloch vector rotates around $z$ axis by $\pi$.

**Exercise** 6.8.1 Show that $Z|+\rangle = |-\rangle$ and $Z|-\rangle = |+\rangle$. ZGate flips $|\pm\rangle$ exactly (YGate flips only up to the global phase). This property is useful in quantum computation.

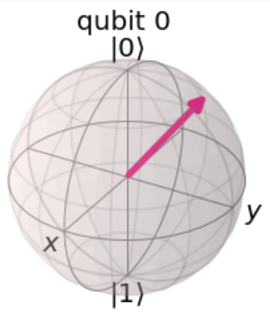
**Qiskit Example** 6.8.1 Let us demonstrate the effect of ZGate using Qiskit. Using the BLoch sphere representation, ZGate transform $(\theta, \phi)$ to $(\theta, \phi + \pi)$. Try $\theta = \pi/4$ and $\phi = -\pi/4$. Check that he Bloch vector rotates around $z$ axis by $\pi$.

```
# import QuatumCircuit and QuantumRegister classes.
from qiskit import QuantumCircuit, QuantumRegister
from qiskit.opflow import Zero, One

# import STatevector class
from qiskit.quantum_info import Statevector

# import numpy
import numpy as np

theta=np.pi/4
phi=-np.pi/4

ket0=np.cos(theta/2)*Zero + np.sin(theta/2)*np.exp(phi*1j)*One

# Preparation
qr=QuantumRegister(1,'q') # create a single qubit with name 'q'.
qc=QuantumCircuit(qr)  # create a quantum circuit

# set the qubit to |L>
qc.initialize(ket0.to_matrix())

# apply Xgate
qc.z(0)

# Final state
ket1=Statevector(qc)

# Compare the final state with |R> in Bloch sphere.
from qiskit.visualization import plot_bloch_multivector

# Generate Bloch vectors
bloch0 = plot_bloch_multivector(ket0)
bloch1 = plot_bloch_multivector(ket1)

from IPython.display import display

# Compare |psi> and Z|psi>.  They are equivalent in the Bloch sphere.
display("Original |psi>",bloch0,"Z|psi>",bloch1)
```

```
'Original |psi>'
```



```
'Z|psi>'
```



When ZGate acts on a superposition state in $\{|+\rangle, |-\rangle\}$, what will be the outcome?

### 6.8.3. Important Properties

$$Z^2 = I$$

This means that

1. $Z^2$ does not do any thing on the qubit.
2. $Z$ is self-inverse, that is $Z^{-1} = Z$.
3. $Z$ is self-adjoint ($Z^\dagger = Z$) since $Z$ is unitary ($Z^\dagger = Z^{-1}$) by definition.

## 6.9. Hadamard gate

This is another very important one-qubit gate. It changes basis set between $\{|0\rangle, |1\rangle\}$ and $\{|+\rangle, |-\rangle\}$ . We shall call this gate simply HGate.

API References: HGate

### 6.9.1. Definition

**Transformation**

$$H|0\rangle = |+\rangle, \qquad H|1\rangle = |-\rangle \tag{6.11}$$

This is the standard way to generate the $x$-basis ((5.2) from the computational basis.

**Matrix expression**

$$H \doteq \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{6.12}$$

**U gate expression**

$$H = U\left(\frac{\pi}{2}, 0, \pi\right) \tag{6.13}$$

**R Gate expression**

$$H = i\, R_x(\pi) \cdot R_y\left(\frac{\pi}{2}\right)$$

The qiskit circuit symbol is $h$ and it appears in quantum circuit as

```python
from qiskit import QuantumCircuit
qc=QuantumCircuit(1)
qc.h(0)
qc.draw()
```

```
   ┌───┐
q: ┤ H ├
   └───┘
```

**Exercise** 6.9.1 Show that $H^{-1} = H$. This means that the $x$-basis is transformed to the computational basis by $H$ gates.

$$H|+\rangle = |0\rangle, \quad H|-\rangle = |1\rangle \tag{6.14}$$

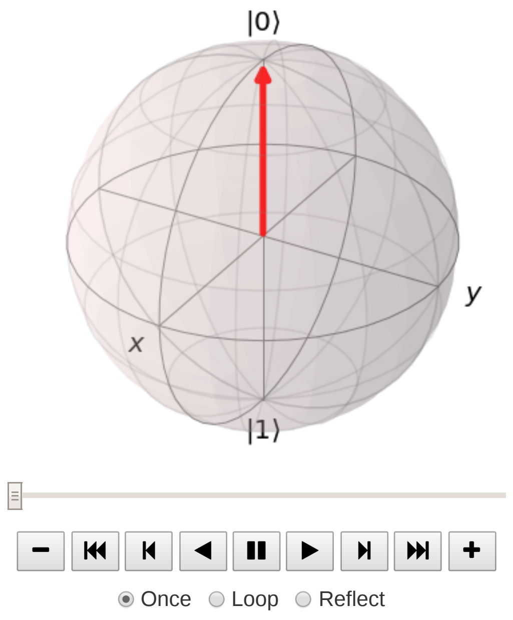
**Qiskit Example** 6.9.1 A common use of the Hadamard gate is to switch the basis set. In the quantum simulation of coin tossing (Qiskit Example 5.3.1) we have already used a Hadamard gate to generate $+\rangle$. Here we flip $|0\rangle$ to $|1\rangle$ via the $x$-basis. First, we switch the basis from $|0\rangle$ to $|+\rangle$ by HGate. Flip $|+\rangle$ to $|-\rangle$ by ZGate. Then, switch back to the original basis by HGate. The final state is $|1\rangle$.

$$|0\rangle \xrightarrow{H} |+\rangle \xrightarrow{Z} |-\rangle \xrightarrow{H} |1\rangle$$

This means $X = H \cdot Z \cdot H$. In this example, the following process is visualized with Qiskit.

```
%%capture
from qiskit import *
from qiskit.visualization import visualize_transition

qc=QuantumCircuit(1)

qc.h(0)
qc.z(0)
qc.h(0)

movie=visualize_transition(qc,fpg=50, spg=1)
```

```
movie
```



## 6.9.2. Acting on a superposition state

The $H$ gate transforms a super position state in an interesting way:

$$H\left(c_0|0\rangle + c_1|1\rangle\right) = c_0|+\rangle + c_1|-\rangle) = \frac{1}{\sqrt{2}}(c_0 + c_1)|0\rangle + \frac{1}{\sqrt{2}}(c_0 - c_1)|1\rangle \quad (6.15)$$

Apart from the normalization constant $\frac{1}{\sqrt{2}}$, it computes addition and subtraction of the coefficients simultaneously, which is actually a Fourier transform (See Section 9.5.). In therms of physics, the Hadamard gate “computes” constructive and destructive interference simultaneously. We will be using this property in may applications.

## 6.9.3. Qiskit example:

We demonstrate Eq. (6.15) using Qiskit. Suppose that the qubit is in a superposition state $\cos(\pi/6)|0\rangle + \sin(\pi/6)|1\rangle$. We want to find $\cos(\pi/6) + \sin(\pi/6)$ and $\cos(\pi/6) - \sin(\pi/6)$ using the Hadamard gate. First, we construct a quantum circuit and test it with Statevector function. Recall that this is not a quantum computation since we cheat by using `initialization` and `Statevector` function. I next example, we try to find the solution by quantum computation.

```python
# import QuatumCircuit and QuantumRegister classes.
from qiskit import *
from qiskit.opflow import Zero, One

# import STatevector class
from qiskit.quantum_info import Statevector

# import numpy
import numpy as np

theta=np.pi/3

c0=np.cos(theta/2)
c1=np.sin(theta/2)

ket0=c0*Zero +c1*One

# Preparation
qr=QuantumRegister(1,'q') # create a single qubit with name 'q'.
qc=QuantumCircuit(qr)  # create a quantum circuit

# set the qubit to |L>
qc.initialize(ket0.to_matrix(),0)

# apply Xgate
qc.h(0)

# Final state
ket1=Statevector(qc).data*np.sqrt(2)

# COmpare
print("c0+c1: Hadamard = ",ket1[0],"   Direct calculation",c0+c1)
print("c0-c1: Hadamard = ",ket1[1],"   Direct calculation",c0-c1)
```

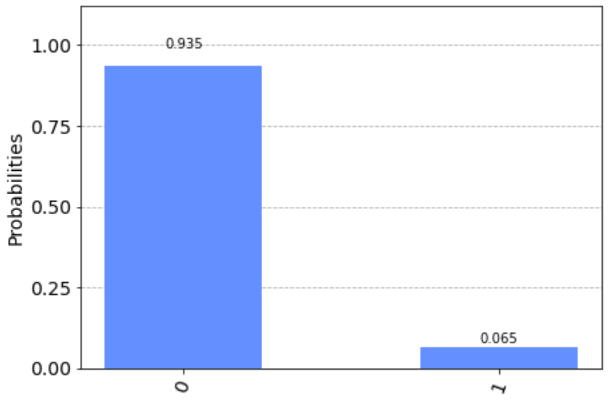
```
c0+c1: Hadamard =  (1.3660254037844384+0j)    Direct calculation
1.3660254037844386
c0-c1: Hadamard =  (0.3660254037844388+0j)    Direct calculation
0.36602540378443876
```

The above calculation mathematically confirms that the Hadamard gate computes the addition and subtraction. Now, we want to solve it using quantum computation. Although there is no advantage over classical computation, we can see how quantum computation calculate two things, addition and subtraction, simultaneously (quantum parallelism). Using the Born rule, the probability that the outcome of measurement is $|0\rangle$ is given by $p_0 = (c_0 + c_1)^2/2$ and similarly for $|1\rangle$ $p_1 = (c_0 - c_1)^2/2$. By repeating quantum computation, many times, we can estimate $p_0$ and $p_1$. From the probabilities we obtain $|c_0 + c_1| = \sqrt{2p_0}$ and $|c_0 - c_1| = \sqrt{2p_1}$. This approach give us only the modulus of the target quantities. We can run the quantum calculation only finite times, the result is not exact. Nevertheless, the results good enough with 10000 tries.

```python
from qiskit import *  # import qiskit

# Preparation
qr=QuantumRegister(1,'q') # create a single qubit named 'q'.
cr=ClassicalRegister(1,'c') # create a single classical bit named 'c'
qc=QuantumCircuit(qr,cr)  # create a quantum circuit

# create the desired superpositionstate using RyGate
qc.ry(theta,0)

# apply Hadamard gate
qc.h(0)

# measure the qubit state
qc.measure(qr,cr)

# The quantum circuit has been constructed.
qc.draw()
```

```
   ┌─────────┐┌───┐┌─┐
q: ┤ Ry(π/3) ├┤ H ├┤M├
   └─────────┘└───┘└╥┘
c: 1/═══════════════╩═
                    0 
```

```
# Now we execute the circuit

# Chose a general quantum simulator without noise.
# The simulator behaves as an ideal quantum computer.
backend = Aer.get_backend('qasm_simulator')

# set number of tries
nshots=10000

# execute the quantum circuit and store the outcome
job = backend.run(qc,shots=nshots)

# extract the result
result = job.result()

# count the outcome
counts = result.get_counts()

# Visualize the outcome
# import histgram plotting function
from qiskit.visualization import plot_histogram
plot_histogram(counts)
```



```
# post analysis
# the outcome of counting is stored in  dict data type
# compute the probabilities
p0=counts['0']/nshots
p1=counts['1']/nshots

# estimate the addition and subtraction from the probabilities
# (this part is classical computation)
add=np.sqrt(2*p0)
sub=np.sqrt(2*p1)

# Compare the results
print("c0+c1: quantum = {:6.3f}   classical {:6.3f}".format(add,c0+c1))
print("c0-c1: quantum = {:6.3f}   classical {:6.3f}".format(sub,c0-c1))
```

```
c0+c1: quantum =  1.367   classical  1.366
c0-c1: quantum =  0.361   classical  0.366
```

## 6.9.4. Important Properties

$$H^2 = I$$

This means that

1. $H^2$ does not do any thing on the qubit.
2. $H$ is self-inverse, that is $H^{-1} = H$.
3. $H$ is self-adjoint ($H^\dagger = H$) since $H$ is unitary ($H^\dagger = H^{-1}$) by definition.

**Exercise** 6.9.2 Knowing that $X = H \cdot Z \cdot H$, show that the following relations are true.

1. $Z = H \cdot X \cdot H$
2. $Z \cdot H = H \cdot X$
3. $H \cdot Z = X \cdot H$

These relations are used to simplify quantum circuits.

**Exercise** 6.9.3 Prove that $H \cdot Y \cdot H = -Y$. Essentially, $Y$ acts in the same way in both the computational basis and $|\pm\rangle$ basis.

Last modified: 07/09/2022

## 6.10. SX and SXdg Gates

SX gate is a native gate of IBMQ hardware. Do not get confused with $S$ times $X$.

API References: SXGate
API References: SXdgGate

### 6.10.1. Definition

**Transformation**

$$\begin{aligned} SX|0\rangle &= e^{i\pi/4}|R\rangle, & SX|1\rangle &= e^{i\pi/4}|L\rangle \\ SX^\dagger|0\rangle &= e^{-i\pi/4}|L\rangle, & SX^\dagger|1\rangle &= e^{-i\pi/4}|R\rangle \end{aligned}$$

**Matrix expression**

$$SX \doteq \frac{1}{2}\begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix}, \quad SX^\dagger \doteq \frac{1}{2}\begin{bmatrix} 1-i & 1+i \\ 1+i & 1-i \end{bmatrix} \tag{6.16}$$

**U gate expression**

$$SX = e^{i\pi/4}\, U\left(\frac{\pi}{2}, -\frac{\pi}{2}, \frac{\pi}{2}\right)$$

**R gate expression**

$$SX = e^{i\pi/4} R_x(\pi/2), \quad SX^\dagger = e^{-i\pi/4} R_x(-\pi/2)$$

Notice that $SX^2 = (SX^\dagger)^2 = X$. Hence, $SX$ and $SX^\dagger$ are square roots of $X$ and they are often expressed as $SX = X^{1/2}$ and $SX^\dagger = X^{-1/2}$.

The Qiskit circuit symbols are `sx` and `sxdg`, respectively. They appear in quantum circuits as

```python
from qiskit import QuantumCircuit
qc=QuantumCircuit(1)
qc.sx(0)
qc.sxdg(0)
qc.draw()
```

```
   ┌────┐┌───────┐
q: ┤ √X ├┤ √Xdg ├
   └────┘└───────┘
```

### 6.10.2. Transformation of other basis kets

- $x$-basis

$$\begin{aligned} SX|+\rangle &= |+\rangle, & SX|-\rangle &= i|-\rangle \\ SX^\dagger|+\rangle &= |+\rangle, & SX^\dagger|-\rangle &= -i|-\rangle \end{aligned}$$

- $y$-basis

$$\begin{aligned} SX|L\rangle &= e^{i\pi/4}|0\rangle, & SX|R\rangle &= e^{i\pi/4}|1\rangle \\ SX^\dagger|L\rangle &= e^{-i\pi/4}|1\rangle, & SX^\dagger|R\rangle &= e^{-i\pi/4}|0\rangle \end{aligned}$$

### 6.10.3. Acting on a superposition state

When SXGate and SXdgGate are applied to a super position state the coefficient to $|0\rangle$ remains the same but that of $|1\rangle$ gets additional phase factor. That is

$$SX\left(c_0|0\rangle + c_1|1\rangle\right) = \frac{1}{\sqrt{2}}\left(e^{i\pi/4}c_0 + e^{-i\pi/4}c_1\right)|0\rangle + \frac{1}{\sqrt{2}}\left(e^{-i\pi/4}c_0 + e^{i\pi/4} \tag{6.17}$$

$$SX^\dagger\left(c_0|0\rangle + c_1|1\rangle\right) = \frac{1}{\sqrt{2}}\left(e^{-i\pi/4}c_0 + e^{i\pi/4}c_1\right)|0\rangle + \frac{1}{\sqrt{2}}\left(e^{i\pi/4}c_0 + e^{-i\pi/} \tag{6.18}$$

In $\{|+\rangle, |-\rangle\}$ basis,

$$SX\left(c_0|+\rangle + c_1|-\rangle\right) = c_0|+\rangle + ic_1|-\rangle \tag{6.19}$$

$$SX^\dagger\left(c_0|+\rangle + c_1|-\rangle\right) = c_0|+\rangle - ic_1|-\rangle \tag{6.20}$$

Comparing these relations with Eqs. (6.23) and (6.24), we find that SXGate and SGate work in the same way but in different basis sets.

### 6.10.4. Additional useful Properties

$SX \cdot SX = SX^\dagger \cdot SX^\dagger = X$ implies that

1. $SX = X \cdot SX^\dagger = SX^\dagger \cdot X, \quad SX^\dagger = X \cdot SX = SX \cdot X.$
2. $X = SX \cdot X \cdot SX^\dagger = SX^\dagger \cdot X \cdot SX$
3. $SX = X \cdot SX \cdot X, \quad SX^\dagger = X \cdot SX^\dagger \cdot X.$

## 6.11. S Gate and S$^\dagger$ Gate

We shall call $S$ and $S^\dagger$ gates SGate and SdgGate, respectively.

API References: SGate
API References: SdgGate

### 6.11.1. Definition

**Transformation**

$$\begin{aligned} S|0\rangle &= |0\rangle, & S|1\rangle &= i|1\rangle \\ S^\dagger|0\rangle &= |0\rangle, & S^\dagger|1\rangle &= -i|1\rangle \end{aligned} \tag{6.21}$$

**Matrix expression**

$$S \doteq \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}, \quad S^\dagger \doteq \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix} \tag{6.22}$$

**U gate expression**

$$S = U\left(0, 0, \frac{\pi}{2}\right), \quad S^\dagger = U\left(0, 0, -\frac{\pi}{2}\right)$$

(Sgate-U)

**R gate expression**

$$S = e^{i\pi/4} R_z(\pi/2), \quad S^\dagger = e^{-i\pi/4} R_z(-\pi/2)$$

The qiskit circuit symbols are `s` and `sdg`, respectively. They appear in quantum circuits as

```
from qiskit import QuantumCircuit
qc=QuantumCircuit(1)
qc.s(0)
qc.sdg(0)
qc.draw()
```

```
   ┌───┐┌─────┐
q: ┤ S ├┤ Sdg ├
   └───┘└─────┘
```

## 6.11.2. Acting on a superposition state

When SGate and SdgGate are applied to a super position state the coefficient to $|0\rangle$ remains the same but that of $|1\rangle$ gets additional phase factor. That is

$$S\left(c_0|0\rangle + c_1|1\rangle\right) = c_0|0\rangle + ic_1|1\rangle \tag{6.23}$$

$$S^\dagger\left(c_0|0\rangle + c_1|1\rangle\right) = c_0|0\rangle - ic_1|1\rangle \tag{6.24}$$

In Bloch sphere representation,

$$S\left(\cos(\theta/2)|0\rangle + \sin(\theta/2)e^{i\phi}|1\rangle\right) = \cos(\theta/2)|0\rangle + \sin(\theta/2)e^{i(\phi+\pi/2)}|1\rangle$$

$$S^\dagger\left(\cos(\theta/2)|0\rangle + \sin(\theta/2)e^{i\phi}|1\rangle\right) = \cos(\theta/2)|0\rangle + \sin(\theta/2)e^{i(\phi-\pi/2)}|1\rangle$$

This suggests that the Bloch vector rotates around $z$ axis by $\pm\pi/2$. Notice that the relative phase changes by $e^{\pm i\pi/2} = \pm i$. Recall that $Z$ changes the relative phase by $e^{i\pi} = -1$. Further notice that $(e^{\pm i\pi/2})^2 = e^{i\pi}$. Hence, $S^2 = (S^\dagger)^2 = Z$. Because of this relation, $S$ is sometimes expressed as $\sqrt{Z}$ or $Z^{1/2}$ and $S^\dagger = Z^{-1/2}$.

Setting $c_0 = c_1 = \frac{1}{\sqrt{2}}$, we find basis transformation

$$S|+\rangle = |L\rangle, \quad S|-\rangle = |R\rangle \tag{6.25}$$

$$S^\dagger|+\rangle = |R\rangle, \quad S^\dagger|-\rangle = |L\rangle \tag{6.26}$$

Since $S$ is unitary $SS^\dagger = I$ and thus $S^{-1} = S^\dagger$ and $(S^\dagger)^{-1} = S$. Now the inverse of (6.25) and `SdgGate- fwd` are

$$S^\dagger|L\rangle = |+\rangle, \quad S^\dagger|R\rangle = |-\rangle \tag{6.27}$$

$$S|L\rangle = |-\rangle, \quad S|R\rangle = |+\rangle \tag{6.28}$$

Combining HGate and SGate, we can transform the computational basis $\{|0\rangle, |1\rangle\}$ to $\{|L\rangle, |R\rangle\}$ by $S \cdot H$ and its inverse is $H \cdot S^\dagger$. Now, we know we can move from one basis to another by $H$, $S$, and $S \cdot H$.

---

**Qiskit Example** 6.11.1 We demonstrate the above basis set transformation using Qiskit. First, we construct a quantum circuit corresponding to the following transformation

$$|0\rangle \xrightarrow{H} |+\rangle \xrightarrow{S} |L\rangle \xrightarrow{S^\dagger} |+\rangle \xrightarrow{H} |0\rangle$$

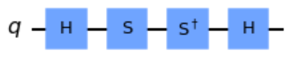
Notice that the whole operation can be written as $H \cdot S^\dagger \cdot S \cdot H|0\rangle$ it can be simplified as

$$H \cdot S^\dagger \cdot S \cdot H|0\rangle = H \cdot (S^\dagger \cdot S) \cdot H = H \cdot I \cdot H = H^2 = I$$

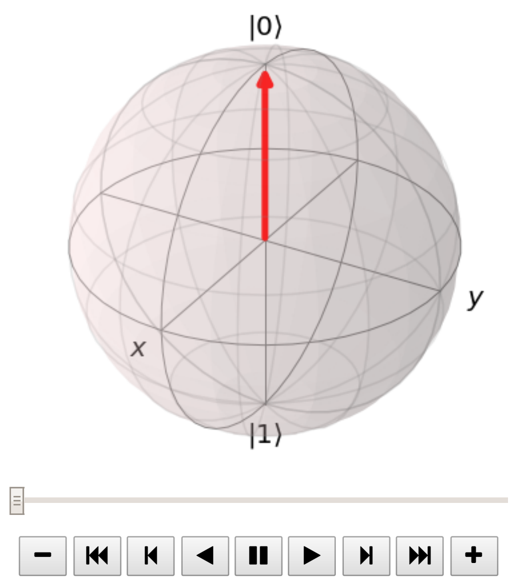
Hence, the whole operation does nothing at all. In order to avoid unnecessary computation like this, we need to understand the properties of gates. You will surprise that a long circuit can be significantly shortened by contracting gates.

```
%%capture
from qiskit import *
from qiskit.visualization import visualize_transition

qc=QuantumCircuit(1)

qc.h(0)
qc.s(0)
qc.sdg(0)
qc.h(0)

movie=visualize_transition(qc,fpg=50, spg=1)
```

```
qc.draw('mpl')
```



```
movie
```



[✓ Once] Loop Reflect

### 6.11.3. Additional useful Properties

$S \cdot, S = S^\dagger \cdot S^\dagger = Z$ implies that

1. $S = Z \cdot S^\dagger = S^\dagger \cdot Z, \quad S^\dagger = Z \cdot S = S \cdot Z.$
2. $Z = S \cdot Z \cdot S^\dagger = S^\dagger \cdot Z \cdot S$
3. $S = Z \cdot S \cdot Z, \quad S^\dagger = Z \cdot S^\dagger \cdot Z.$

---

Last modified 07/11/2022

## 6.12. P Gate

P gate is also known as *phase gate*. This gate contains a parameter.

[API References: PhaseGate]

### 6.12.1. Definition

**Transformation**

$$\begin{aligned} P(\theta)|0\rangle &= |0\rangle, \qquad & P(\theta)|1\rangle &= e^{i\theta}|1\rangle \\ P^\dagger(\theta)|0\rangle &= |0\rangle, \qquad & P^\dagger(\theta)|1\rangle &= e^{-i\theta}|1\rangle \end{aligned}$$

**Matrix expression**

$$P(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}, \qquad P^\dagger(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{-i\theta} \end{bmatrix}$$

**U gate expression**

$$P = U(0,0,\theta) = U(0,\theta,0), \quad P^\dagger = U(0,0,-\theta) = U(0,-\theta,0)$$

**R gate expression**

$$P = e^{i\theta/2} R_z(\theta), \quad P^\dagger = e^{-i\theta/2} R_z(-\theta)$$

The qiskit circuit symbol is `p`. It appears in quantum circuits as

```python
from qiskit.circuit import QuantumCircuit, Parameter
t=Parameter('\u03B8')
qc=QuantumCircuit(1)
qc.p(t,0)
qc.draw()
```

```
   ┌──────┐
q: ┤ P(θ) ├
   └──────┘
```

$Z$, $S$, $T$ and $P$ all rotates the Bloch vector around the $z$ axis. gates are special cases of $P$ gate: $Z = P(\pi)$, $S = P(\pi/2)$ and $T = P(\pi/4)$. While these relations are mathematically exact and $P$ gate can replace them, the parameter-free gates should be used when the rotation angle

### 6.12.2. Acting on a superposition state

When PGate is applied to a superposition state, the coefficient to $|0\rangle$ remains the same but that of $|1\rangle$ gets additional phase factor. That is

$$P(t)\left(c_0|0\rangle + c_1|1\rangle\right) = c_0|0\rangle + e^{it}c_1|1\rangle \tag{6.29}$$

In Bloch sphere representation,

$$P(t)\left(\cos(\theta/2)|0\rangle + \sin(\theta/2)e^{i\phi}|1\rangle\right) = \cos(\theta/2)|0\rangle + \sin(\theta/2)e^{i(\phi+t)}|1\rangle$$

This suggests that the Bloch vector rotates around $z$ axis by $t$.

## 6.13. Change of basis sets

Although the computational basis ($z$-basis) $\{|0\rangle, |1\rangle\}$ is the primary basis set, in some cases $x$-basis $\{|+\rangle, |-\rangle\}$ and $y$-basis $\{|L\rangle, |R\rangle\}$ are more convenient.

We can change from a basis set to another by $H$, $S$, and $S \cdot H$.

$$\begin{aligned} H|0\rangle &= |+\rangle, & H|1\rangle &= |-\rangle \\ S|+\rangle &= |L\rangle, & S|-\rangle &= |R\rangle \\ (S\cdot H)|0\rangle &= |L\rangle, & (S\cdot H)|1\rangle &= |R\rangle \end{aligned}$$

and their inverse transformation are

$$\begin{aligned} H|+\rangle &= |0\rangle, & H|-\rangle &= |1\rangle \\ S^\dagger|L\rangle &= |+\rangle, & S^\dagger|R\rangle &= |-\rangle \\ (H\cdot S^\dagger)|L\rangle &= |0\rangle, & (H\cdot S^\dagger)|R\rangle &= |1\rangle \end{aligned}$$

### 6.13.1. Transformation of gates

Consider the following three transofrmations by gates, $G_z$, $G_x$, and $G_y$.

$$\begin{aligned} G_z\left(a_0|0\rangle + a_1|1\rangle\right) &= b_0|0\rangle + b_1|1\rangle \\ G_x\left(a_0|+\rangle + a_1|-\rangle\right) &= b_0|+\rangle + b_1|-\rangle \\ G_y\left(a_0|L\rangle + a_1|R\rangle\right) &= b_0|L\rangle + b_1|R\rangle \end{aligned}$$

The difference is only the basis set and the coefficients are the same in all three cases. Then, there are simple relation among the gates.

$$G_z = H\cdot G_x \cdot H, \quad G_y = S\cdot G_x\cdot S^\dagger = (S\cdot H)\cdot G_z\cdot(H\cdot S^\dagger)$$

**Example 1** Swapping coefficients

Suppose that a state vector is given in $\{|+\rangle, |-\rangle\}$ basis as $c_0|+\rangle + c_1|-\rangle$. We want to swap the coefficients.

$$c_0|+\rangle + c_1|-\rangle \quad \overset{?}{\to} \quad c_1|+\rangle + c_0|-\rangle$$

What gate do we need? In Section 6.6, we learned that XGate swaps the coeeficients in the computational basis. Recalling that HGate switches basis set between the computational basis and the $x$-basis. Then, $H \cdot X \cdot H$ should do the job. Let us try it.

$$c_0|+\rangle + c_1|-\rangle \quad \overset{H}{\longrightarrow} \quad c_0|0\rangle + c_1|1\rangle \quad \overset{X}{\longrightarrow} \quad c_1|0\rangle + c_0|1\rangle \quad \overset{H}{\longrightarrow} \quad c_1|+\rangle + c_0|-\rangle$$

REcall that we derived $Z = H \cdot X \cdot H$ in Section 6.9. $Z$ is the gate we wanted.

**Exercise** 6.13.1 We want to swap the coefficients in the $y$-basis. That is $c_0|L\rangle + c_1|R\rangle \quad \overset{?}{\to} \quad c_1|L\rangle + c_0|R\rangle$. Find an gate for this transformation.

**Exercise** 6.13.2 In Section 6.10, we noticed that $S$ gate and $SX$ gate work in the same way but in different basis sets. Show that $SX = H \cdot S \cdot H$ and $SX^\dagger = H \cdot S^\dagger \cdot H$.

**Example 2** addition and subtraction of the coefficients

We want to find a gate that transforms a state vector as

$$c_0|+\rangle + c_1|-\rangle \quad \overset{?}{\to} \quad \frac{1}{\sqrt{2}}(c_0 + c_1)|+\rangle + \frac{1}{\sqrt{2}}(c_0 - c_1)|-\rangle$$

Recalling that $H$ gate does the same type of transformation for the computational basis. The $H$ gate itself is also used to change the basis set. Hence, $H \cdot H \cdot H$ will do the job. However, we recall that $H^2 = I$. Hence, $H \cdot H \cdot H = H$. Interestingly, A single $H$ works in the same way on the two different basis sets.

Similarly, we want to find a gate that transforms a state vector in the $y$-basis as

$$c_0|L\rangle + c_1|R\rangle \quad \overset{?}{\to} \quad \frac{1}{\sqrt{2}}(c_0 + c_1)|L\rangle + \frac{1}{\sqrt{2}}(c_0 - c_1)|R\rangle$$

The $S$ and $S^\dagger$ gates swap the basis set between the $x$-basis and the $y$-basis. We already know that the $H$ gate works between the computational gate and the $x$ gate. Hence, $S \cdot H \cdot S^\dagger$ should achieve the task.

**Exercise** 6.13.3 Show that $S \cdot H \cdot S^\dagger$ works as desired.

## 6.13.2. Qiskit Example: Measuring quantum phase.

This is the extension of Section 6.9.3.

Consider a superposition state,

$$|\phi\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + e^{i\phi}|1\rangle\right)$$

The corresponding Bloch vector is $\vec{r} = \cos\phi\,\hat{x} + \sin\phi\,\hat{y}$.

We want to find out the value of the phase angle $\phi$. If we measure this state, we only get the modulus of the coefficients and thus no information on the phase is obtained. In physics, the phase difference between two waves is measured by the interference pattern. Here, we use the same method.

First, we write the state vector with the $x$-basis.

$$|\phi\rangle = \frac{1}{2}\left[(1 + e^{i\phi})|+\rangle + (1 - e^{i\phi})|-\rangle\right]$$

Notice that the modulus square of coefficients are

$$p_0 = \left|\frac{1}{2}(1+e^{i\phi})\right|^2 = \frac{1}{2}(1+\cos\phi), \qquad p_1 = \left|\frac{1}{2}(1-e^{i\phi})\right|^2 = \frac{1}{2}(1-\cos\phi)$$

If these quantities are measured, we find the value of $\cos\phi = p_0 - p_1$. However, the measurement must be done in the computational basis. Hence, we change the basis set by $H$.

$$H|\phi\rangle = \frac{1}{2}\left[(1+e^{i\phi})|0\rangle + (1-e^{i\phi})|1\rangle\right]$$

Measuring $|0\rangle$ and $|1\rangle$, we find the probabilities $p_0$ and $p_1$ from which we find $\cos\phi$. Notice that this is the $x$ component of the Bloch vector.

In order to determine $e^{i\phi}$, we also have to find $\sin\phi$, that is the $y$ component of the Bloch vector. We rewrite the state vector again but in the $y$-basis.

$$\psi\rangle = \frac{1}{2}\left[(1-ie^{i\phi})|L\rangle + (1+ie^{i\phi})|R\rangle\right]$$

The modulus square of coefficients are now

$$q_0 = \left|\frac{1}{2}(1-ie^{i\phi})\right|^2 = \frac{1}{2}(1+\sin\phi), \qquad q_1 = \left|\frac{1}{2}(1+ie^{i\phi})\right|^2 = \frac{1}{2}(1-\sin\phi)$$

To measure these quantities, the $y$-basis must be transformed to the computational basis. It is done by $H \cdot S^\dagger$.

$$(H \cdot S^\dagger)|\psi\rangle = \frac{1}{2}\big(1-ie^{i\phi}\big)|0\rangle + \frac{1}{2}\big(1+ie^{-i\phi}\big)|1\rangle$$

The measurement determines he probabilities $q_0$ and $q_1$ and we find $\sin\phi = q_0 - q_1$.
Now we have full information on $e^{i\phi} = \cos\phi + i\sin\phi$.

Let us try this method using Qiskit.

First we create the superposition state using $H$ and $P$ gates. We set $\phi = \pi/3$. The goal of the quantum computation is to obtain this angle by quantum measurement. In the following quantum circuit, we use two qubits, one for $\cos\phi$ and the other for $\sin\phi$.

```
import numpy as np

from qiskit import *

# two classical registers
cr=ClassicalRegister(2,'c')

# two quantum registers (qubits)
qr=QuantumRegister(2,'q')

# set the quantum circuit
qc=QuantumCircuit(qr,cr)

# Generation of the superposition state
qc.h([0,1])
qc.p(np.pi/3,[0,1])

# separate the preparation part from the phase determination
# by placing a barrier
qc.barrier([0,1])

# real part
qc.h(0)

# imaginary part
qc.sdg(1)
qc.h(1)

# measure the both qubits
qc.measure(qr,cr)

# show the circuit
qc.draw()
```

```
     ┌───┐┌────────┐ ░   ┌───┐      ┌─┐
q_0: ┤ H ├┤ P(π/3) ├─░───┤ H ├──────┤M├───
     ├───┤├────────┤ ░ ┌─┴───┴┐┌───┐└╥┘┌─┐
q_1: ┤ H ├┤ P(π/3) ├─░─┤ Sdg ├┤ H ├─╫─┤M├
     └───┘└────────┘ ░ └─────┘└───┘ ║ └╥┘
c: 2/═══════════════════════════════╩══╩═
                                    0  1
```

```python
# Chose a general quantum simulator without noise.
# The simulator behaves as an ideal quantum computer.
backend = Aer.get_backend('qasm_simulator')

# set number of tries
nshots=8192

# execute the quantum circuit and store the outcome
job = backend.run(qc,shots=nshots)

# extract the result
result = job.result()

# count the outcome
counts = result.get_counts()

# get the marginal counts
from qiskit.result import marginal_counts
counts_x = marginal_counts(result,indices=[0]).get_counts()
counts_y = marginal_counts(result,indices=[1]).get_counts()

# calculate the marginal probability for the sine part
q0=counts_y['0']/nshots
q1=counts_y['1']/nshots
dy=q0-q1

# calculate the marginal probability for the cosine part
p0=counts_x['0']/nshots
p1=counts_x['1']/nshots
dx=p0-p1

# evaluate the phase angle
phi=np.arctan2(dy,dx)

# print out the results
print("measured phi = {:6.3f}".format(phi))
print("exact answer = {:6.3f}".format(np.pi/3))
print("error =  {:6.3f}".format(phi - np.pi/3))
```

```
measured phi =  1.044
exact answer =  1.047
error =  -0.003
```

The result of quantum computing is quite close to the original phase angle.

Last modified: 07/09/2022

## 6.14. State tomography

State *tomography* is an experimental method to determine the state of a quantum system. In this section, we try to determine the pure state of a qubit by quantum computation. Measurement of a single qubit cannot determine it due to the stochastic nature of quantum measurement. We must prepare many qubits in the same state.

In Section 6.13.2, we developed a quantum circuit that determines the quantum phase of a qubit. In that example, we had only one parameter, $\phi$, to be determined. To obtain the full information (up to the global phase) of the state vector, we have to determine two parameters. Recall that the state of a qubit can be expressed as

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + \sin\left(\frac{\theta}{2}\right)e^{i\phi}|1\rangle$$

Measuring this state directly, we obtain only $\left|\cos\left(\frac{\theta}{2}\right)\right|$ and $\left|\sin\left(\frac{\theta}{2}\right)\right|$. Neither $\theta$ nor $\phi$ can be determined from them. We need a quantum algorithm similar to the method used in Section 6.13.2. See the following Strategy box on State Tomography.

**Strategy - State Tomography**

To determine the Bloch vector corresponding to a pure state $\psi\rangle$, change the $x$-, $y$-, $z$-basis to the computational basis and measure the state in the computational basis.

- $x$ component
  Change the basis from the $x$-basis to the computational basis by the $H$ gate.
  Measure $H|\psi\rangle$. Then, $r_x = \sin\theta\cos\phi = p_0 - p_1$.
- $y$ component
  Change the basis from the $y$-basis to the computational basis by the $H \cdot S^\dagger$ gate.
  Measure $(H \cdot S^\dagger)|\psi\rangle$. Then, $r_y = \sin\theta\sin\phi = p_0 - p_1$.
- $z$ component
  Since the $z$-basis and the computational basis are the same, no need to change the basis.
  Measure $|\psi\rangle$. Then, $r_z = \cos\theta = p_0 - p_1$.

One the Bloch vector is measured, we can calculate the angles by $\theta = \arccos(r_z)$ and $\phi = \text{arctan2.}\,(r_y, r_x)$.

```python
import numpy as np

from qiskit import *

# two classical registers
cr=ClassicalRegister(3,'c')

# two quantum registers (qubits)
qr=QuantumRegister(3,'q')

# set the quantum circuit
qc=QuantumCircuit(qr,cr)

# set parameters
theta=np.pi/7
phi=np.pi/3

# Generation of the state
qc.u(theta,phi,0,[0,1,2])

# separate the preparation part from the phase determination
# by placing a barrier
qc.barrier([0,1,2])

# x-component
qc.h(0)

# y-component
qc.sdg(1)
qc.h(1)

# z-component
# no gate for this

# measure the both qubits
qc.measure(qr,cr)

# show the circuit
qc.draw()
```

```
     ┌──────────────┐ ░  ┌───┐          ┌─┐   
q_0: ┤ U(π/7,π/3,0) ├─░──┤ H ├──────────┤M├───
     ├──────────────┤ ░ ┌┴───┴┐┌───┐   └╥┘┌─┐
q_1: ┤ U(π/7,π/3,0) ├─░─┤ Sdg ├┤ H ├────╫─┤M├
     ├──────────────┤ ░ └┬─┬──┘└───┘    ║ └╥┘
q_2: ┤ U(π/7,π/3,0) ├─░──┤M├────────────╫──╫─
     └──────────────┘ ░  └╥┘            ║  ║ 
c: 3/═════════════════════╩═════════════╩══╩═
                          2             0  1 
```

```python
# Chose a general quantum simulator without noise.
# The simulator behaves as an ideal quantum computer.
backend = Aer.get_backend('qasm_simulator')

# set number of tries
nshots=8192

# execute the quantum circuit and store the outcome
job = backend.run(qc,shots=nshots)

# extract the result
result = job.result()

# count the outcome
counts = result.get_counts()

from qiskit.result import marginal_counts

# marginal counts of each component
counts_x = marginal_counts(result,indices=[0]).get_counts()
counts_y = marginal_counts(result,indices=[1]).get_counts()
counts_z = marginal_counts(result,indices=[2]).get_counts()

# get the Bloch vector components
rx=(counts_x['0']-counts_x['1'])/nshots
ry=(counts_y['0']-counts_y['1'])/nshots
rz=(counts_z['0']-counts_z['1'])/nshots

# evaluate the phase angle
theta_qc=np.arccos(rz)
phi_qc=np.arctan2(ry,rx)


# print out the results
print("measured   phi = {:6.3f} (exact  = {:6.3f} )".format(phi_qc, phi))
print("measured theta = {:6.3f} (exact  = {:6.3f} )".format(theta_qc,theta))
```

```
measured   phi =  1.050 (exact  =  1.047 )
measured theta =  0.450 (exact  =  0.449 )
```

# 7. Composite Systems

A single qubit is a smallest information carrying unit similar to classical bit. Despite that a qubit can take infinitely many different states on the surface of a Bloch sphere, it carries the same amount of information as a classical bit since measurement of a qubit produces only two different values like a classical bit. A practically useful quantum computer needs many qubits. Desktop computers process 64 bits of information inside a CPU and store information in giga bits of memory. At least similar number of qubits are needed for practical quantum computation. However, the states of multiple qubits are much more complicated that the same number of classical bits. There are a special kind of superposition states known as *quantum entanglement*, which cannot realized in classical bits. It turns out the entanglement is the most powerful resource for quantum computation.

In this chapter, we study mathematical properties of a composite system consisting of multiple qubits and two-qubit gates which act on two qubits in such a way that the transformation of one qubit depends on the other qubit and vice versa.

## 7.1. Two qubits

### 7.1.1. Hilbert space and computational basis

First, let us recall that a single qubit is in a two-dimensional Hilbert space $\mathbb{C}^2$ and the computational basis vectors $0\rangle$ and $|1\rangle$ span the space. Now we consider two qubits $q_0$ and $q_1$. The composite system lives in a four-dimensional Hilbert space. The dimension $4$ is not $2+2$ but $2 \times 2$. We write it $\mathbb{C}^2 \otimes \mathbb{C}^2$ where $\otimes$ indicates *tensor product*. This mathematical structure is very important. Using the computational basis of each $\mathbb{C}^2$, we can construct four *computational basis* vectors for the composite system as products $|q_1\rangle \otimes |q_0\rangle$:

$$|00\rangle \equiv |0\rangle \otimes |0\rangle, \quad |01\rangle \equiv |0\rangle \otimes |1\rangle, \quad |10\rangle \equiv |1\rangle \otimes |0\rangle, \quad |11\rangle \equiv |0\rangle \otimes |0\rangle. \tag{7.1}$$

The physical meaning of each basis vector is clear. For example, $|01\rangle$ means $q_0$ is in $\rangle$ and $q_1$ in $|0\rangle$.

Other basis sets are also used. The $xx$ basis uses $|\pm\rangle$ for both qubits

$$|++\rangle \equiv |+\rangle \otimes |+\rangle, \quad |+-\rangle \equiv |+\rangle \otimes |-\rangle, \quad |-+\rangle \equiv |-\rangle \otimes |+\rangle, \quad |--\rangle \equiv |-\rangle \otimes |-\rangle$$

The mixed basis in which $q_0$ and $q_1$ use different basis kets is also allowed.

$$|0+\rangle \equiv |0\rangle \otimes |+\rangle, \quad |0-\rangle \equiv |0\rangle \otimes |-\rangle, \quad |1+\rangle \equiv |1\rangle \otimes |+\rangle, \quad |1-\rangle \equiv |1\rangle \otimes |-\rangle$$

**Exercise** 7.1.1 Show that the above basis sets are all orthonormal.

## 7.1.2. Superposition states

The state of two-qubit system is expressed as

$$c_{00}|00\rangle + c_{01}|01\rangle + c_{10}|10\rangle + c_{11}|11\rangle.$$

What does this state physically means? To see that, we assume that Alice has qubit $q_0$ and Bob has $q_1$ . They are separated by a large distance. Even when they are far apart, the above state is still valid. Alice can measure her qubit but not Bob's qubit. Similarly, Bob can measure his qubit but not Alice's. This type of measurement is known as *local measurement*. Now Alice measures her qubit $q_0$ before Bob measures $q_1$. Suppose that the outcome is $|0\rangle$. The state now collapses to

$$\frac{1}{N}(c_{00}|00\rangle + c_{10}|10\rangle) = (c_0|0\rangle + c_1|1\rangle) \otimes |0\rangle.$$

where $N = \sqrt{|c_{00}|^2 + |c_{01}|^2}$, $c_0 = c_{00}/N$, and $c_1 = c_{10}/N$. After the measurement, Alice's qubit is now in definite state $0\rangle$ but Bob's qubit remains in a superposition state. Next, Bob measures his qubit and found it in $|1\rangle$. Then, the state further collapses to $|1\rangle \otimes |0\rangle$.

Recall that the outcome of quantum measurement is completely random and there is no way to predict it. You can calculate the probability to find each state based on the Born rule. The probability that Alice obtain $|0\rangle$ is $p_A(0) = |c_{00}|^2 + |c_{10}|^2$ because Alices qubit is $|0\rangle$ in $|00\rangle = |0\rangle \otimes |0\rangle$ and $|10\rangle = |1\rangle \otimes |0\rangle$. In other words, when Alice obtained $|0\rangle$, Bob's qubit can be still $|0\rangle$ or $|1\rangle$, hence we need to take into account both.

The probability that Bob gets $|1\rangle$ after Alice get $|0\rangle$ is $p_B(1|0) = |c_1|^2$, which is the *conditional probability* after Alice gets $|0\rangle$. The net probability to find $|10\rangle$ is

$$p_A(0)p_B(1|0) = \left(|c_{00}|^2 + |c_{01}|^2\right)|c_1|^2 = |c_{10}|^2$$

which is the probability to obtain $|10\rangle$. , The Born rule worked well for the composite case.

**Exercise** 7.1.1 Suppose that Alice measured her qubit before Bob did and got $|1\rangle$. Bob measured his qubit after Alice and obtained $|1\rangle$. What is the state after their measurement and what is the probability to get it?

Now an important question arises. If Bob measures before Alice, does the probability remain the same?

Suppose that Bob measured before Alice and got $|1\rangle$. The probability that happens is $p_B(1) = |c_{10}|^2 + |c_{11}|^2$, which is different from the previous case despite that Bob gets the same outcome. This is weird but let us move on. After Bob's measurement the state collapse to

$$\frac{1}{N}(c_{10}|10\rangle + c_{11}|11\rangle) = |1\rangle \otimes (c_0|0\rangle + c_1|1\rangle) \tag{7.2}$$

where $N = \sqrt{|c_{10}|^2 + |c_{11}|^2}$, $c_0 = c_{10}/N$, and $c_1 = c_{11}/N$. Next, Alice measured her qubit and obtained $0\rangle$ with the probability $p_A(0|1) = |c_0|^2$. The final state is $|10\rangle$. The net probability is

$$p_B(1)p_A(0|1) = \left(|c_{10}|^2 + |c_{11}|^2\right)|c_1|^2 = |c_{10}|^2$$

in agreement with the previous case. This agreement is guaranteed by the *Bayes' theorem* $p_A(x)p_B(y|x) = p_B(y)p_A(x|y)$.

While the net probability does not depend on the order of the measurement, the probabilities of individual observers experience depend on it. Despite that Alice and Bob are separated by a far distance, the act of Alice on her qubit changed the state of Bob's qubit instantaneously. This conclusion is quite counter intuitive and needs to be examined more carefully.

### 7.1.3. Product states

Let us consider a special case where the two qubit state is

$$|10\rangle = |1\rangle \otimes |0\rangle \tag{7.3}$$

that is $c_{00} = c_{01} = c_{11} = 0$ and $c_{10} = 1$ in Eq (7.2). Alice always finds her qubit in $|0\rangle$ (probability=1) and Bob always find his in $|1\rangle$ without exception (probability=1). In this case, the order of measurement does not affect the individual measurements. There seems no controversy.

Next we consider

$$|+-\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle). \tag{7.4}$$

This state is closer to the general case (7.2). However, the probability that Alice gets $|0\rangle$ is $\frac{1}{2}$ and the probability Bob gets $|1\rangle$ is also $\frac{1}{2}$ regardless of who measures first.

The state of the composite systems in these two examples can be written as *product state* $|q_1\rangle \otimes |q_0\rangle$. In Eq. (7.3), $|q_0\rangle = |0\rangle$ and $|q_1\rangle = |1\rangle$. In Eq. (7.4), $|q_0\rangle = |-\rangle$ and $|q_1\rangle = |+\rangle$. When a composite system is in a product state, the measurements of $q_0$ and $q_1$ are completely independent and thus the order does not matter.

### 7.1.4. Entangled states

When the state of a composite system cannot be written in a form of product $|q_1\rangle \otimes |q_0\rangle$, we say that the system is *entangled*. *Quantum entanglement* is unique to quantum systems. When the qubits are entangled, they are not independent. Measurement on one qubit change the state of the other qubit.

A pair of qubits interact at a certain place and an entangle state

$$\frac{1}{\sqrt{2}}(|01\rangle + |10\rangle) \tag{7.5}$$

is formed. Alice takes $q_0$ and travels to a distant place with it. Bob takes the other qubit and travels in the opposite direction. Alice and Bob are separated by a large distance.

The entangled state (7.5) indicates that Alice's qubit is either $|0\rangle$ or $|1\rangle$ and so is Bob's. When Alice's measurement results in $|0\rangle$, then Bob's qubit necessarily in $|1\rangle$ since $|00\rangle$ is absent. On the other hand, if Alice finds $|1\rangle$ then Bob's qubit must be in $|0\rangle$. Alice immediately knows what Bob's will find. Hence, their measurement is perfectly correlated.

This correlation is not necessarily controversial since classical correlation exits. Consider a pair of grabs. Each side of the grab is placed in a separate box. Alice took a box and Bob the other. They don't know which side of the grab is in their of box. They open the box at their home. When Alice finds the left grab, the Bob must find the right grab. This is an example of perfect classical correlation. When Bob opened the box before Alice, the outcome remain the same.

The quantum correlation by entanglement seems the same as the classical case but it is actually quite different. Before measurement neither spin has a definite state. Both qubits are mixture of $|0\rangle$ and $|1\rangle$. Alice's measurement destroys the mixture of Bob's qubit *instantaneously* over *distance*. How can Alice's action on her qubit changes the state of Bob's qubit instantaneously over an arbitrarily long distance? This is known as *EPR paradox*. Einstein called it "Spooky action at a distance" and claimed that the theory of quantum mechanics is incomplete. However, every experimental observation is found to be consistent with the standard theory of quantum mechanics. Nicolas Gisin's book [2] explains the issue nicely without complicated mathematics. Anyone interested in the quantum correlation, the book is highly recommended.

Once David Mermin explained the common attitude toward the theory of quantum mechanics as “Shut up and calculate”. [3]. For the moment, we set aside the incomprehensible aspect of quantum entanglement and just accept its extraordinary properties. It turns out that entanglement carries very useful properties that classical physics cannot offer. We exploit them in quantum computation. In fact, the success of quantum computation relies on quantum entanglement.

## 7.1.5. Bell basis

The Hilbert space of two qubits is four-dimensional and spanned by four basis vectors. The computational basis (7.1) is based on the product states. In many cases, basis set based on entangled states is desired. The most popular entangled basis is *Bell basis* defined by four Bell states:

$$
\begin{aligned}
|\Phi^{\pm}\rangle &= \frac{1}{\sqrt{2}}(|00\rangle \pm |11\rangle) \\
|\Psi^{\pm}\rangle &= \frac{1}{\sqrt{2}}(|01\rangle \pm |10\rangle)
\end{aligned}
$$

Among them, $|\Psi^{-}\rangle$, known as *singlet state*, plays a particularly important role in quantum information theory.

**Qiskit note: Visualizing two-qubit states**

Visualizing the state of composite systems is challenging. For small composite systems, `plot_state_qsphere` draw a sphere and places the computational basis vectors on it as small circles. See the above example. The size of each circle represents the magnitude of coefficient to the basis such as $|c_{00}|$. In the above example, $|01\rangle$ and $|10\rangle$ are not shown because $c_{01} = c_{10} = 0$. The color of the circle show the phase $e^{i\theta}$, blue for $\theta = 0$ and yellow for $\theta = \pi$.

**Exercise** 7.1.2 Generate $|\Psi^{\pm}\rangle$ and visualize the results using Qiskit. (HINT: You can filp one of qubits by `x` gate.

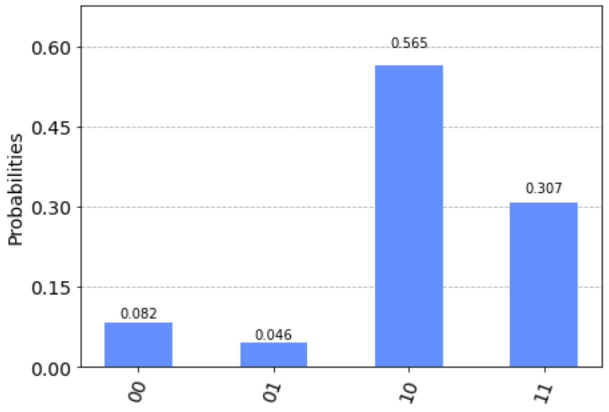
## 7.1.6. Two-qubit measurements in computational basis

We want to determine $|c_{ij}|$ in (7.2) by measurement. It can be done easily in Qiskit.

```
from qiskit import *
import numpy as np

cr=ClassicalRegister(2,'c')
qr=QuantumRegister(2,'q')
qc=QuantumCircuit(qr,cr)

# randomly oriented qubits
a=np.pi*np.random.rand()
b=np.pi*np.random.rand()
c=np.pi*np.random.rand()
qc.u(a,b,c,0)
a=np.pi*np.random.rand()
b=np.pi*np.random.rand()
c=np.pi*np.random.rand()
qc.u(a,b,c,1)

qc.barrier()
qc.measure(0,0)
qc.measure(1,1)

qc.draw()
```

```
     ┌─────────────────────────┐ ░ ┌─┐
q_0: ┤ U(1.2759,1.7262,1.9598) ├─░─┤M├───
     ├─────────────────────────┤ ░ └╥┘┌─┐
q_1: ┤ U(2.418,2.8657,0.40626) ├─░──╫─┤M├
     └─────────────────────────┘ ░  ║ └╥┘
c: 2/═══════════════════════════════╩══╩═
                                    0  1
```

```
# Chose a general quantum simulator without noise.
# The simulator behaves as an ideal quantum computer.
backend = Aer.get_backend('qasm_simulator')

# set number of tries
nshots=8192

# execute the quantum circuit and store the outcome
job = backend.run(qc,shots=nshots)

# extract the result
result = job.result()

# count the outcome
counts = result.get_counts()

from qiskit.visualization import plot_histogram
plot_histogram(counts)
```



## 7.1.7. Entanglement Measure

How do you know that a given state is entangled or not? It is not a trivial question and no simple method is known at present. However, the the state is pure (expressed as a state vector), the presence of bipartite entanglement can be determined by a rather simple method.

Consider a bipartite system of $\mathcal{H}_A \otimes \mathcal{H}_B$. The system is in a pure state $|\psi\rangle$.

1. Write the state in a form of density matrix: $\rho = |\psi\rangle\langle\psi|$.
2. Take partial trace of $\rho$ over the subspace $\mathcal{H}_B$. $\rho_A == \mathrm{Tr}_B \rho$.
3. Evaluate the von Neumann entropy of the reduced density. $S_A = -\mathrm{Tr}_A\left(\rho_A \ln \rho_A\right)$.
4. If $S_A = 0$, then there is no entanglement. Otherwise, there is entanglement.

**EXample**

Let us try to find if a $\psi\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$ is entangled or not. WE use matrix representation.

Step 1: $$\rho = \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{bmatrix}$$

Step 2: $$\rho_A = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

Step 3:
To evalue the von Neumann entropy, we need to find the eigenvalues of $\rho_A$. They are $\lambda_1 = 0$ and $\lambda_2 = 1$.

$$S_A = -\lambda_0 \ln \lambda_0 + \lambda_1 \ln \lambda_1 = -0 \ln 0 = -1 \ln 1 = 0$$

Remember that $0 \ln 0 = 0 \times \infty = 0$.

Step 4: Since $S_A = 0$, there is no entanglement. In fact, the state can be written as a product $|\psi\rangle = |+\rangle \otimes |+\rangle$.

Consider another state $|\phi\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + i|11\rangle)$ which is quite similar to the previous case but notice "i" on the last term.

Step 1: \$$\rho = \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{i}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{i}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{i}{4} \\ \frac{-i}{4} & \frac{-i}{4} & \frac{-i}{4} & \frac{1}{4} \end{bmatrix}$\$

Step 2: \$$\rho_A = \begin{bmatrix} \frac{1}{2} & \frac{1+i}{4} \\ \frac{1-i}{4} & \frac{1}{2} \end{bmatrix}$\$

Step 3:
\$$\lambda_1 = \frac{1}{4}\left(2 - \sqrt{2}\right), \quad \lambda_2 = \frac{1}{4}\left(2 - \sqrt{2}\right)$\$

$$S_A = -\lambda_0 \ln \lambda_0 + \lambda_1 \ln \lambda_1 \approx 0.416$$

Step 4. $S_A > 0$, thus the state is entangled.

## 7.2. More qubits

Two qubits are too few for useful quantum computation and we need many more. In this section, we consider $n$ qubits where $n > 2$. Each qubit is in $\mathbb{C}^2$ and thus ther system of $n$ qubits is in $2^n$ dimensional Hilbert space $\mathbb{C}^2 \otimes \cdots \otimes \mathbb{C}^2$, which spanned by $2^n$ basis vectors.

### 7.2.1. Notation

In this book, we write the computational basis of $n$ qubits as

$$|q_{n-1}\, q_{n-2}\, \cdots\, q_1\, q_0\rangle \equiv |q_{n-1}\rangle \otimes |q_{n-2}\rangle \otimes \cdots \otimes |q_1\rangle \otimes |q_0\rangle$$

Notice that the qubits are ordered from the right to the left. For example, $|01011\rangle$ is a computational basis ket for five qubits representing $|0\rangle \otimes |1\rangle \otimes |0\rangle \otimes |1\rangle \otimes |1\rangle$.

For large $n$, the expression becomes very long. We use a shorthand expression or an index based on integers in classical binary strings:

$$|j\rangle_n = |j_{n-1}\, j_{n-2}\, \cdots\, j_1\, j_0\rangle$$

where $j_k \in \{0, 1\}$ and

$$j = 2^{n-1} j_{n-1} + 2^{n-2} j_{n-2} + \cdots + 2 j_1 + j_0 = \sum_{k=0}^{n-1} 2^k j_k.$$

For $n = 3$, we have eight basis vectors

$$\begin{array}{llll} |0\rangle_3 = |000\rangle, & |1\rangle_3 = |001\rangle, & |2\rangle_3 = |010\rangle, & |3\rangle_3 = |011\rangle \\ |4\rangle_3 = |100\rangle, & |5\rangle_3 = |101\rangle, & |6\rangle_3 = |110\rangle, & |7\rangle_3 = |111\rangle \end{array}$$

---

**Exercise** For the system of 5 qubits, find what computational basis set $|13\rangle_5$ means.

### 7.2.2. Entanglement

Recall the system is entangled if the state vector of a composite system cannot be written as a product of individual state vectors. For two qubits, this definition is clear. For three qubits, what does "product" state mean? Do all three qubits have to be separated like $|q_2\rangle \otimes |q_1\rangle \otimes |q_0\rangle$? If $q_1$ and $q_2$ are entangled but not with $q_0$ the state vector can be written as

$$|\text{entangled}\rangle_2 \otimes |q_0\rangle$$

Is this a product state? In one sense this is a product state and thus the whole system is not entangled. However, a part of the whole system is entangled. When the state vector of a whole system cannot be written as any form of product state, we say that the whole system is entangled. For example, the state known as the *GHZ* state:

$$|\mathrm{GHZ}\rangle = \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$$

cannot be written as any form of product state. We say that the state is maximally entangled. To understand the effect of entanglement, qubits $q_0$, $q_1$ and $q_2$ are delivered to Alice, Bob, and Charlie, respectively. Before any measurement, all of them have a equal chance to get $|0\rangle$ or $|1\rangle$. Alice measures her qubit before others and get $|1\rangle$. Immediately, the state collapse to $|111\rangle$ and she knows that Bob's and Charlie's qubits are both in $|1\rangle$. Now, they have no chance to get $|0\rangle$ (but they don't know it.) This happens even when they are far apart. In one sense, this entanglement is strong because measurement of a single qubit removes the uncertainty in two others. On the other hand, the entanglement is not robust since the measurement of a single qubit destroys the entanglement entirely. The GHZ state is used in various quantum algorithms including Quantum Byzantine agreement.

There is another maximally entangled state known as the *W* state

$$|\mathrm{W}\rangle = \frac{1}{\sqrt{3}}(|001\rangle + |010\rangle + |100\rangle).$$

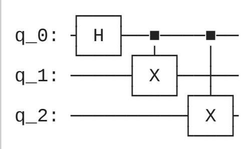
Again, Alice measures her qubit first but gets $|0\rangle$ this time. The state collapses to $\frac{1}{\sqrt{2}}(01\rangle + |10\rangle) \otimes |0\rangle$. This entanglement is a little bit more robust than the GHZ state since even after Alice measured, entanglement between Bob and Charie remains.

Finding all maximally entangled states for bigger composite systems is not a trivial task. We will not discuss it here.

**Quiskit example: creation of GHZ state**

The GHZ state can be created essentially in the same way as the Bell state $|\Phi^+\rangle$. The following circuit creates it using `H` and `CX` gates.

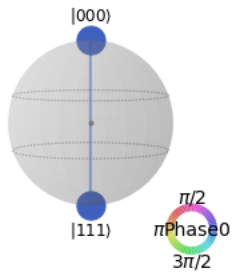
```
from qiskit import *

qr=QuantumRegister(3,'q')
qc=QuantumCircuit(qr)

qc.h(0)
qc.cx(0,1)
qc.cx(0,2)

qc.draw()
```



```
from qiskit.quantum_info import Statevector

psi=Statevector(qc)
psi.draw('latex')
```

$$\frac{\sqrt{2}}{2}|000\rangle + \frac{\sqrt{2}}{2}|111\rangle$$

```
from qiskit.visualization import plot_state_qsphere
# it's an entangled state.  Use qsphere.
plot_state_qsphere(psi,figsize=(4,4))
```



Generating W state is a little bit more complicated. The following circuit creates the W state.

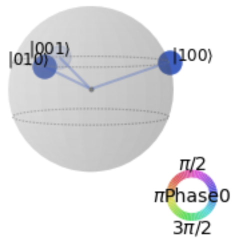
```python
from qiskit import *
import numpy  as np

qr=QuantumRegister(3,'q')
qc=QuantumCircuit(qr)

theta = 2*np.arccos(1./np.sqrt(3))

qc.ry(theta,0)
qc.ch(0,1)
qc.cx(1,2)
qc.cx(0,1)
qc.x(0)

qc.draw()
```

```
     ┌────────────┐                   ┌───┐
q_0: ┤ Ry(1.9106) ├──■─────────────■──┤ X ├
     └────────────┘┌─┴─┐         ┌─┴─┐└───┘
q_1: ──────────────┤ H ├──■──────┤ X ├─────
                   └───┘┌─┴─┐    └───┘
q_2: ───────────────────┤ X ├──────────────
                        └───┘
```

```python
from qiskit.quantum_info import Statevector

psi=Statevector(qc)
psi.draw('latex')
```

$$\frac{\sqrt{3}}{3}|001\rangle + \frac{\sqrt{3}}{3}|010\rangle + \frac{\sqrt{3}}{3}|100\rangle$$

```python
plot_state_qsphere(psi,figsize=(4,4))
```



## 7.3. Encoding numbers

If a quantum computation involves numbers, first we need to encode them.The situation is not much different from classical computing. First, we have only finite number of qubits, we cannot encode all numbers. Secondly, strictly speaking continuous numbers cannot be expressed in digital computers. In this section, we study how to encode integer numbers and real numbers.

### 7.3.1. Integers

Suppose that we encode integers in $n$ qubits. We just use the same encoding scheme as classical encoding. That is to express intgers in binary and map them to bit string. Recall the inter indexing of the computational basis introduced in Section 7.2.1 which also uses the binary string. Hence, integers from $0$ to $2^n - 1$ can be expressed by the computational basis

$$j \quad \Rightarrow \quad |j\rangle_n$$

For example, $151 \Rightarrow |151\rangle_8 = |10010111\rangle$. You can also encode the same number with more qubits. Using $16$ qubits, $151 \Rightarrow |151\rangle_{16} = |0000000010010111\rangle$. The lower half of the binary string matches to the 8-qubit expression.

The following Qiskit example encode 151 in eight qubits.

```python
from qiskit import *

qr=QuantumRegister(8,'q')
qc=QuantumCircuit(qr)

# create integer 151
qc.x([0,1,2,4,7])

from qiskit.quantum_info import Statevector

Statevector(qc).draw('latex')
```

$$|10010111\rangle$$

## 7.3.2. Real numbers with floating point arithmetic

Real numbers are continuous and thus there are uncountable many numbers even upper and lower bounds are limited. The situation is same in classical computation. THerefore, we express real numbers approximately using the *floating-point arithmetic* method. Suppose that we write a real number in a scientific notation *mantissa* $\times$ *scale*, for example in base 10 $0.3452 \times 10^7$. We can always make the mantissa less than 1 by adjusting the scaling. We can also make the mantissa integer as $3452 \times 10^3$. Therefore, the scientific notation can be written as integer times scaling. Now, we use the similar expression in binary. Consider a binary fractional number $0.10011 \times 2^{10}$. The mantissa can be written as

$$0.10011 = \frac{1}{2} + \frac{1}{2^4} + \frac{1}{2^5} = \frac{1}{2^5}\left(2^4 \times 1 + 2 \times 1 + 1\right) = \frac{1}{2^5} \times 10011$$

Hence, $0.10011 \times 2^{10} = 10011 \times 2^5$.

In general binary fractional number expressed with $n$ qubits is written as

$$0.j_{n-1}\,j_{n-2}\ldots j_1\,j_0 = \frac{1}{2^n}\left(2^{n-1}j_{n-1} + 2^{n-2}j_{n-2} + \cdots + 2j_1 + j_0\right) = \frac{1}{2^n}\sum_{k=0}^{n-1} 2^k j_k$$

which indicate that the mantissa can be written as an integer divided by $2^n$. Real numbers smaller than 1 can be approximately encoded with $n$ qubit as integer devided by $2^n$. In practice, $0 < x < 1$ is encoded as $|2^n x\rangle = |j\rangle_n$. At the end of computation, we can find $x$ by computing $j/2^n$ on a classical computer.

Using 3 qubits, we can encode eight real numbers, $\frac{0}{8} = 0$, $\frac{1}{8} = 0.125$, $\frac{2}{8} = 0.25$, $\frac{3}{8} = 0.375$, $\frac{4}{8} = 0.5$, $\frac{5}{8} = 0.625$, $\frac{6}{8} = 0.75$, $\frac{7}{8} = 0.875$. The gap between two adjacent values is $0.125$, too big as approximated real number. However, for some applications, this is good enough. See for examples, Section 9.5 and Section 9.6. If $64$ qubits are available, we can encode $2^{64} = 18446744073709551616$ different real numbers between 0 and 1. That means the gap is approximately $5.42101086 \times 10^{-20}$, which is small enough for most of applications. Unfortunately, the currently available quantum computer do not have enough qubits to use floating point arithmetic.

```
from qiskit import *

qr=QuantumRegister(3,'q')
qc=QuantumCircuit(qr)

# create |101>
qc.x([0,2])

# find corresponding real number
# extract output
psi=Statevector(qc).to_dict()
psi=list(psi.keys())[0]
x = int(psi[2])/2. + int(psi[1])/2**2.+ int(psi[0])/2**3.

print("|{0:s}> corresponds to x={1:5.3f}".format(psi,x))
```

```
|101> corresponds to x=0.625
```

### 7.3.3. Real numbers encoded in coefficients

Unlike classical bits, quantum states naturally contains continuous numbers in superposition states. Consider general qubit state

$$|\psi\rangle = \cos\left(\frac{\pi x}{2}\right)|0\rangle + \sin\left(\frac{\pi x}{2}\right)e^{2\pi y}|1\rangle$$

where $0 < x < 1$ and $0 < y < 1$. We can store two real numbers $x$ and $y$ in a single qubit. Their value can be determined by the method discussed in Section 6.14. It is relatively easy to encode and decode the data. However, it is not trivial to use $x$ and $y$ in actual calculation. However, this method works in certain application nicely, especially the real numbers represent angles.

let us calculate $\cos(A)\cos(B)\cos(C),\ \cos(A)\cos(B)\sin(C), \cdots \sin(A)\sin(B)\sin(C)$ all at once where $0 < A, B, C < \pi/2$. Using three qubits, we construct a state

$$\begin{aligned} |\psi\rangle &= (\cos(A)|0\rangle + \sin(A)|1\rangle) \otimes (\cos(B)|0\rangle + \sin(B)|1\rangle) \otimes (\cos(C)|0\rangle + \sin(C)|1\rangle) \\ &= \cos(A)\cos(B)\cos(C)|000\rangle + \cos(A)\cos(B)\sin(C)|001\rangle + \cdots + \sin(A)\sin(B)\sin(C) \end{aligned}$$

By measuring all qubits, we obtain probabilities $p_{000}, \cdots, p_{111}$. Then, $\cos(A)\cos(B)\cos(C) = \sqrt{p_{000}}, \cdots$. Now we calculated all combination of three trig functions simultaneously.

```
import numpy as np
from qiskit import *

cr=ClassicalRegister(3,'c')
qr=QuantumRegister(3,'q')
qc=QuantumCircuit(qr,cr)

A=np.pi/3
B=np.pi/4
C=np.pi/6

qc.ry(2*A,0)
qc.ry(2*B,1)
qc.ry(2*C,2)

qc.measure([0,1,2],[0,1,2])

# Chose a general quantum simulator without noise.
# The simulator behaves as an ideal quantum computer.
backend = Aer.get_backend('qasm_simulator')

# set number of tries
nshots=8192

# execute the quantum circuit and store the outcome
job = backend.run(qc,shots=nshots)

# extract the result
result = job.result()

# count the outcome
counts = result.get_counts()

for k in counts.keys():
    sqrtp = np.sqrt(counts[k]/nshots)
    term=(k.replace('0',' cos')).replace('1',' sin')
    print("{0:s}  = {1:3.3f}".format(term,sqrtp))
```

```
 cos sin cos  = 0.306
 sin sin sin  = 0.301
 cos sin sin  = 0.531
 cos cos cos  = 0.308
 cos cos sin  = 0.531
 sin cos sin  = 0.303
 sin sin cos  = 0.174
 sin cos cos  = 0.184
```

## 8. Two-qubits Gates

When two gates $A$ and $B$ are applied to qubits $q_0$ and $q_1$, respectively, we can write the combined gates as $C = A \otimes B$. We do not call it 2-qubit gate. In fact, $A$ and $B$ are not necessarily applied to the qubits at the same time. A 2-quibit gate cannot be expressed as a tensor product of two 1-qubit gates and it must act simultaneously on two qubits.

Commonly used 2-qubit gates are *controoled* gates, which are actually conditional 1-qubit gates. A 1-qubit gate such as `X` is applied to $q_1$ only when $q_0$ is in $|1\rangle$. Otherwise, no action is taken.
There are other kind of 2-qubit gates such as `Swap` gate.

Most of common 2-qubit gates are spectial cases of more general canonical gates. Many models used in condensed matter physics are directly expressed by canonical gates, understandin the relation between canonical gates and common 2-qubit gates are particluarly important.

| Generaic Name | Qiskit Class Name | Qiskit Ciruit Name | Symbols |
|---|---|---|---|
| Controlled X | CXGate | cx or cnot | `CX` or `CNOT` |
| Controlled Y | CYGate | cy | `CY` |
| Controlled Z | CZGate | cz | `CZ` |
| Controlled Hadamard | CHGate | ch | `CH` |
| Controlled SX | CSXGate | csx | `CSX` |
| Controlled Phase | CPhaseGate | cp | `CP` |
| Controlled Rotaion X | CRXGate | crx | $\texttt{CR}_x$ |
| Controlled Rotaion Y | CRYGate | cry | $\texttt{CR}_y$ |
| Controlled Rotaion Z | CRZGate | crz | $\texttt{CR}_z$ |
| Controlled Unitary 1 | CU1Gate | cu1 | $\texttt{CU}_1$ |
| Controlled Unitary 3 | CU3Gate | cu3 | $\texttt{CU}_3$ |
| Controlled Unitary | CUGate | cu | `CU` |

**Others**

- `Swap`
- $\texttt{H}_2$

**Canonical Gates**

$\texttt{Can}(t_x, t_y, t_z) = \exp\left[-i\frac{\pi}{2}(t_x X \otimes X + t_y Y \otimes Y + t_z \otimes ZZ)\right]$

## 8.1. CX gate

CX (Controlled-X), known also as `CNOT`, is one of the most essential gates for quantum computation.

### 8.1.1. Definition

**Operational Definition**

When gate $\mathrm{CX}_{q_0}^{q_1}$ acts on $|q_1\, q_0\rangle$, X is applied to $q_1$ if $q_0 = 1$ and nothing is done otherwise.[1] Qubit $q_0$ serves as source and $q_1$ as target. Mathematically, it is expressed as

$$\mathrm{CX}_{q_0}^{q_1} = \mathrm{I} \otimes |0\rangle\langle 0| + \mathrm{X} \otimes |1\rangle\langle 1|$$

Switching source and target qubits,

$\mathrm{CX}_{q_1}^{q_0}|q_1\, q_0\rangle$ means “Apply X to $q_0$ if $q_1 = 1$ and do nothing otherwise.” Mathematically, it is expressed as

$$\mathrm{CX}_{q_1}^{q_0} = |0\rangle\langle 0| \otimes \mathrm{I} + |1\rangle\langle 1| \otimes \mathrm{X}$$

**Transformation**

CX transforms computational basis as follows:

$$\begin{aligned}
\mathrm{CX}_{q_0}^{q_1}|00\rangle &= |00\rangle \\
\mathrm{CX}_{q_0}^{q_1}|01\rangle &= |11\rangle \\
\mathrm{CX}_{q_0}^{q_1}|10\rangle &= |10\rangle \\
\mathrm{CX}_{q_0}^{q_1}|11\rangle &= |01\rangle
\end{aligned}$$

$$\begin{aligned}
\mathrm{CX}_{q_0}^{q_1}|00\rangle &= |00\rangle \\
\mathrm{CX}_{q_1}^{q_0}|01\rangle &= |01\rangle \\
\mathrm{CX}_{q_1}^{q_0}|10\rangle &= |11\rangle \\
\mathrm{CX}_{q_1}^{q_0}|11\rangle &= |10\rangle
\end{aligned}$$

**Matrix representation**

$$\mathrm{CX}_{q_0}^{q_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

which is default CX in Qiskit. In many literature, `CX` corresponds to:

$$\mathrm{CX}_{q_1}^{q_0} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Don't get confused. Our notation avoids the confusion.

The Qiskit circuit symbol is `cx` and it appears in circuit as:

```python
from qiskit import QuantumCircuit
qc=QuantumCircuit(2)

# CX_{q0,q1}
qc.cx(0,1)
# CX_{q1,q0}
qc.cx(1,0)

print(qc)
```

```
          ┌───┐
q_0: ──■──┤ X ├
     ┌─┴─┐└─┬─┘
q_1: ┤ X ├──■──
     └───┘
```

## 8.1.2. Acting on superposition state

$\mathrm{CX}_{q_0}^{q_1}$ swaps the coefficients of $|01\rangle$ and $|11\rangle$.

$$\mathrm{CX}_{q_0}^{q_1}\left(c_{00}|00\rangle + c_{01}|01\rangle + c_{10}|10\rangle + c_{11}|11\rangle\right) = c_{00}|00\rangle + c_{11}|01\rangle + c_{10}|10\rangle + c_{01}|11\rangle$$

Similarly, $\mathrm{CX}_{q_1}^{q_0}$ swaps the coefficients of $|10\rangle$ and $|11\rangle$.

$$\mathrm{CX}_{q_1}^{q_0}\left(c_{00}|00\rangle + c_{01}|01\rangle + c_{10}|10\rangle + c_{11}|11\rangle\right) = c_{00}|00\rangle + c_{01}|01\rangle + c_{11}|10\rangle + c_{10}|11\rangle$$

The following Qiskit example demonstrates it.

```python
from qiskit import *

qr=QuantumRegister(2,'q')
qc=QuantumCircuit(qr)

qc.x([0,1])
qc.h([0,1])

qc.draw()
```

```
     ┌───┐┌───┐
q_0: ┤ X ├┤ H ├
     ├───┤├───┤
q_1: ┤ X ├┤ H ├
     └───┘└───┘
```

```python
from qiskit.quantum_info import Statevector
# state before applying CX
print("State before applying CX.")
Statevector(qc).draw('latex')
```

```
State before applying CX.
```

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

```
qc.cx(0,1)
qc.draw()
```

```
     ┌───┐┌───┐     
q_0: ┤ X ├┤ H ├──■──
     ├───┤├───┤┌─┴─┐
q_1: ┤ X ├┤ H ├┤ X ├
     └───┘└───┘└───┘
```

```
# state after applyiong CX
print("State after applying CX.")
Statevector(qc).draw('latex')
```

```
State after applying CX.
```

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

Compare the states before and after applying `cx` and find how the coefficients changed.

**Source qubit not necessarily preserved by control gates**

The definition of CX seems indicating that only the state of the target qubit is modified and the state of the source qubit remains the same. Surprisingly, that is not true. In some cases, the state of the source qubit also changes.

Let us look at the above example more carefully. The state before applying $\mathrm{CX}_{q_0}^{q_1}$ is

$$\frac{1}{2}(|00\rangle-|01\rangle-|10\rangle+|11\rangle)=\frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)\otimes\frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)=|-\rangle\otimes|-\rangle$$

and after applying `cx`$_{q_0}^{q_1}$ ,

$$\frac{1}{2}(|00\rangle+|01\rangle-|10\rangle-|11\rangle)=\frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)\otimes\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)=|-\rangle\otimes|+\rangle$$

Notice that the source qubit is transformed from $|-\rangle$ to $|+\rangle$. This kind of transformation is called *phase kickback* since the phase of the source qubit changed. The phase kickback is a ubiquitous strategy in quantum algorithms. See the next subsection.

## 8.1.3. Phase kickback

Let us look at the action of CX on z-basis $|\pm\pm\rangle$. We assume that $q_0$ is the control qubit and $q_1$ is the target.

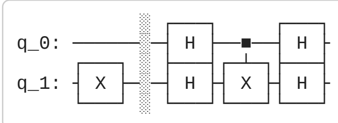
$$\begin{aligned}
\mathrm{CX}_{q_0}^{q_1}|++\rangle &= \frac{1}{2}\mathrm{CX}_{q_0}^{q_1}(|00\rangle+|01\rangle+|10\rangle+|11\rangle) &&= \frac{1}{2}(|00\rangle+|11\rangle+|10\rangle+|01\rangle)\\
\mathrm{CX}_{q_0}^{q_1}|+-\rangle &= \frac{1}{2}\mathrm{CX}_{q_0}^{q_1}(|00\rangle-|01\rangle+|10\rangle-|11\rangle) &&= \frac{1}{2}(|00\rangle-|11\rangle+|10\rangle-|11\rangle)\\
\mathrm{CX}_{q_0}^{q_1}|-+\rangle &= \frac{1}{2}\mathrm{CX}_{q_0}^{q_1}(|00\rangle+|01\rangle-|10\rangle-|11\rangle) &&= \frac{1}{2}(|00\rangle+|11\rangle-|10\rangle-|01\rangle)\\
\mathrm{CX}_{q_0}^{q_1}|--\rangle &= \frac{1}{2}\mathrm{CX}_{q_0}^{q_1}(|00\rangle-|01\rangle-|10\rangle+|11\rangle) &&= \frac{1}{2}(|00\rangle-|11\rangle-|10\rangle+|01\rangle)
\end{aligned}$$

This transformation can be described as “Apply Z to $q_0$ if $q_1=-$ and do nothing otherwise.” Interestingly, now $q_1$ is the control qubit and $q_0$ is the target. The gate has not change and this operation is still “Apply X to $q_1$ if $q_0=1$ and do nothing otherwise.” The two operations are equivalent. The reversal of control-target relation is known as signature of *phase kickback*. Other controlled gates also show phase kickback.

Remembering that the Hadamard gate changes basis set from z-basis to x-basis and vice versa. Applying H gate before CX we can realize the phase kickback shown above. After the kickback, we can go back to z-basis by another H gate. Then, we have reversed CX.

In the following example, we start with an initial state $|10\rangle$. In the first example, the basis is switched to x-basis by H gate and apply $\mathrm{CX}_{q_0}^{q_1}$. In the second computation, $\mathrm{CX}_{q_1}^{q_0}$ is directly applied. Both get the same result. Try other initial conditions and confirm that the two circuits do the same transformation.

```python
from qiskit import *

qr = QuantumRegister(2,'q')
qc = QuantumCircuit(qr)

qc.x(1)
qc.barrier()
qc.h([0,1])
qc.cx(0,1)
qc.h([0,1])
qc.draw()
```

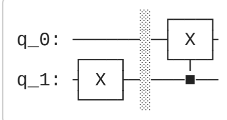


```python
from qiskit.quantum_info import Statevector
Statevector(qc).draw('latex')
```

$$None|01\rangle|11\rangle$$

However, this must be equivalent to

```python
qc = QuantumCircuit(qr)
qc.x(1)
qc.barrier()
qc.cx(1,0)
qc.draw()
```



```python
Statevector(qc).draw('latex')
```

$$|11\rangle$$

## 8.1.4. Change of basis

Suppose that we want to flip $q_1$ if $q_0$ is $|-\rangle$ and do nothing otherwise. We change the basis of $q_0$ using H gate, apply $\mathrm{CX}_{q_0}^{q_1}$, and apply H again to $q_0$. Note that the third step bring $q_0$ back to the original state. We can generate superposition of Bell states using the circuit.

$$\mathrm{H}_{q_0} \cdot \mathrm{CX}_{q_0}^{q_1} \cdot \mathrm{H}_{q_0}|00\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle - |11\rangle) = \frac{1}{\sqrt{2}}\left(\Phi^- + \Psi^+\right)$$
$$\mathrm{H}_{q_0} \cdot \mathrm{CX}_{q_0}^{q_1} \cdot \mathrm{H}_{q_0}|01\rangle = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle + |11\rangle) = \frac{1}{\sqrt{2}}\left(\Phi^+ - \Psi^-\right)$$
$$\mathrm{H}_{q_0} \cdot \mathrm{CX}_{q_0}^{q_1} \cdot \mathrm{H}_{q_0}|10\rangle = \frac{1}{2}(|00\rangle + |01\rangle - |10\rangle + |11\rangle) = \frac{1}{\sqrt{2}}\left(\Phi^+ + \Psi^-\right)$$
$$\mathrm{H}_{q_0} \cdot \mathrm{CX}_{q_0}^{q_1} \cdot \mathrm{H}_{q_0}|11\rangle = \frac{1}{2}(-|00\rangle + |01\rangle + |10\rangle + |11\rangle) = \frac{1}{\sqrt{2}}\left(-\Phi^- + \Psi^+\right)$$

If we want to flip the target qubit also in x-basis, H gate is applied on $q_1$ before and after CX. as well. However, this circuit is identical to just a single CX since it does the same transformation simply in a different basis.

The following example generates $\frac{1}{\sqrt{2}}\left(\Phi^- + \Psi^+\right)$.

```python
from qiskit import *

qr=QuantumRegister(2,'q')
qc=QuantumCircuit(qr)

qc.h(0)
qc.cx(0,1)
qc.h(0)

qc.draw()
```

```
     ┌───┐     ┌───┐
q_0: ┤ H ├──■──┤ H ├
     └───┘┌─┴─┐└───┘
q_1: ─────┤ X ├─────
          └───┘
```

```python
from qiskit.quantum_info import Statevector

Statevector(qc).draw('latex')
```

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

### 8.1.5. Simple applications

As the above example shows, CX gate is used to create Bell states. In Section 9, many quantum strategies that use CX extensively will be discussed. Here are a few small applications.

#### Generating Bell states

The No one-qubit gate can generate an entangled state. The CX gate is commonly used to generate entangled states from product states. As discussed in Section 7.1, Bell states play important roles in quantum computation. We can generate all Bell states by by applying CX_{q_0}^{q_1} \cdot H_{q_0}$ to the computational basis vectors.

$$\begin{aligned}
CX_{q_0}^{q_1} \cdot H_{q_0}|00\rangle &= |\Phi^+\rangle \\
CX_{q_0}^{q_1} \cdot H_{q_0}|01\rangle &= |\Phi^-\rangle \\
CX_{q_0}^{q_1} \cdot H_{q_0}|10\rangle &= |\Psi^+\rangle \\
CX_{q_0}^{q_1} \cdot H_{q_0}|11\rangle &= -|\Psi^-\rangle
\end{aligned} \tag{8.1}$$

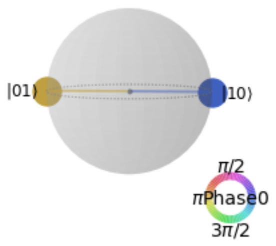
The last one has unwanted phase “-”. We can get rid of it by applying Z on both qubits. In the following Qiskit code, we generate the singlet state $|\Psi^-$.

**Exercise** 8.1.1 Generate $|\Psi^{\pm}\rangle$ and visualize the results using Qiskit. (HINT: You can flip one of qubits by X gate.

```python
from qiskit import *
from qiskit.quantum_info import Statevector

qr=QuantumRegister(2,'q')
qc=QuantumCircuit(qr)

# generate |11>
qc.x([0,1])

# apply CX*H
qc.h(0)
qc.cx(0,1)

# adjust phase
qc.z([0,1])

psi=Statevector(qc)
psi.draw('latex')
```

$$\frac{\sqrt{2}}{2}|01\rangle - \frac{\sqrt{2}}{2}|10\rangle$$

```python
from qiskit.visualization import plot_state_qsphere
# it's an entangled state.  Use qsphere.
plot_state_qsphere(psi,figsize=(4,4))
```



**Exercise** 8.2.1 Generate $|\Phi^{\pm}\rangle$ and visualize the results using Qiskit.

### Bell state measurement

We can expand any two qubit states in the Bell basis as

$$|\psi\rangle = c_{\Phi^+}|\Phi^+\rangle + c_{\Phi^-}|\Phi^-\rangle + c_{\Psi^+}|\Psi^+\rangle + c_{\Psi^-}|\Phi^-\rangle. \tag{8.2}$$

Now, we want find the probabilities to find the Bell states. The process is called *Bell measurement*. After the measurement, the state collapses to a Bell state depending on the outcome of the measurement. The Bell measurement is commonly used in quantum information processes, such as *quantum teleportation*. However, the Bell measurement is not trivial. Any local measurement fails and thus the standard measurement based on the computational basis does not help.

In the example above, we transformed the computational basis vectors to the Bell states using $\mathrm{CX}_{q_0}^{q_1}\cdot \mathrm{H}_{q_0}$. By inverting the transformation, we can convert the Bell states to the computational basis. That is applying $\mathrm{H}_{q_0}\cdot \mathrm{CX}_{q_0}^{q_1}$ to the Bell states

$$\begin{aligned}
\mathrm{H}_{q_0}\cdot\mathrm{CX}_{q_0}^{q_1}|\Phi^+\rangle &= |00\rangle\\
\mathrm{H}_{q_0}\cdot\mathrm{CX}_{q_0}^{q_1}|\Phi^-\rangle &= |01\rangle\\
\mathrm{H}_{q_0}\cdot\mathrm{CX}_{q_0}^{q_1}|\Psi^+\rangle &= |10\rangle\\
\mathrm{H}_{q_0}\cdot\mathrm{CX}_{q_0}^{q_1}|\Phi^-\rangle &= -|11\rangle
\end{aligned}$$

Applying $(H\otimes I)\cdot CX$ to the superposition (8.2), we obtain

$$(H\otimes I)\cdot CX|\psi\rangle = c_{\Phi^+}|00\rangle + c_{\Phi^-}|10\rangle + c_{\Psi^+}|01\rangle - c_{\Psi^-}|11\rangle. \tag{8.3}$$

The resulting state is a superposition in computational basis but the expansion coefficients are the same as before the transformation. Now, the standard measurement in the computational basis determine the probabilities of finding the corresponding Bell state. The actual measurement collapses the state to one of computational basis vectors but the Bell measurement should result in one of the Bell state. The computational basis vector obtained from the measurement can be transformed back to the corresponding Bell state.

### Addition modulo 2

We want to compute modulo-2 addition of two bits $x$ and $y$. We write it $x\otimes y$. Its truth table is

| $x$ | $y$ | $x\oplus y$ |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

In classical computer, XOR gate calculate it. Unlike classical XOR, quantum computation must be reversible. Thus we need to retain the values of $x$ and $y$. The out needs one qubit. In total, we need three qubits $|z\,y\,x\rangle = |z\rangle\otimes|y\rangle\otimes|x\rangle$ where $z$ contains $x\oplus y$ at the end. We assume that $z=0$

initially. We want to construct a quantum circuit which transforms $|0\,y,x\rangle$ to $|(x\oplus y)\,y\,x\rangle$. Writing it explicitly.

$$\begin{array}{ccc} |000\rangle & \Rightarrow & |000\rangle \\ |001\rangle & \Rightarrow & |101\rangle \\ |010\rangle & \Rightarrow & |110\rangle \\ |011\rangle & \Rightarrow & |011\rangle \end{array}$$

The second transformation can be done by CX$_{q_0}^{q_1}$ and the third transformation can be done by a CX$_{q_1}^{q_2}$. These two gates also works for the first transformation. However, it does not work for the last transformation. It takes more than one step. It would be nice if $z$ is directly correlated to $y$ or $x$. Let us try CX$_{q_0}^{q_1}$.

$$\begin{array}{l} \mathrm{CX}_{q_0}^{q_1}|000\rangle = |000\rangle \\ \mathrm{CX}_{q_0}^{q_1}|001\rangle = |011\rangle \\ \mathrm{CX}_{q_0}^{q_1}|010\rangle = |010\rangle \\ \mathrm{CX}_{q_0}^{q_1}|011\rangle = |001\rangle \end{array}$$

Now , the output $z$ and $y$ are perfectly correlated. We can now apply CX$_{q_1}^{q_2}$ to find $z$.

$$\begin{array}{l} \mathrm{CX}_{q_1}^{q_2}\mathrm{CX}_{q_0}^{q_1}|000\rangle = \mathrm{CX}_{q_1}^{q_2}|000\rangle = |000\rangle \\ \mathrm{CX}_{q_1}^{q_2}\mathrm{CX}_{q_0}^{q_1}|001\rangle = \mathrm{CX}_{q_1}^{q_2}|011\rangle = |111\rangle \\ \mathrm{CX}_{q_1}^{q_2}\mathrm{CX}_{q_0}^{q_1}|010\rangle = \mathrm{CX}_{q_1}^{q_2}|010\rangle = |110\rangle \\ \mathrm{CX}_{q_1}^{q_2}\mathrm{CX}_{q_0}^{q_1}|011\rangle = \mathrm{CX}_{q_1}^{q_2}|001\rangle = |001\rangle \end{array}$$

The value of $z$ is correct but $x$ and $y$ are not preserved. That is not good. Next,we try to recover them by X$_{q_0}^{q_1}$.

$$\begin{array}{l} \mathrm{CX}_{q_0}^{q_1}\mathrm{CX}_{q_1}^{q_2}\mathrm{CX}_{q_0}^{q_1}|000\rangle = \mathrm{CX}_{q_0}^{q_1}|000\rangle = |000\rangle \\ \mathrm{CX}_{q_0}^{q_1}\mathrm{CX}_{q_1}^{q_2}\mathrm{CX}_{q_0}^{q_1}|001\rangle = \mathrm{CX}_{q_0}^{q_1}|111\rangle = |101\rangle \\ \mathrm{CX}_{q_0}^{q_1}\mathrm{CX}_{q_1}^{q_2}\mathrm{CX}_{q_0}^{q_1}|010\rangle = \mathrm{CX}_{q_0}^{q_1}|110\rangle = |110\rangle \\ \mathrm{CX}_{q_0}^{q_1}\mathrm{CX}_{q_1}^{q_2}\mathrm{CX}_{q_0}^{q_1}|011\rangle = \mathrm{CX}_{q_0}^{q_1}|001\rangle = |011\rangle \end{array}$$

Now we have the desired output. Three CX gates calculate $x\oplus y$.

```python
from qiskit import *
from qiskit.quantum_info import Statevector
import numpy as np


# loop over all inputs
for x in range(2):
    for y in range(2):

        qr=QuantumRegister(3,'q')
        qc=QuantumCircuit(qr)

        # generate input state
        if x==1:
            qc.x(0)
        if y==1:
            qc.x(1)

        # compute x \oplus y
        qc.cx(0,1)
        qc.cx(1,2)
        qc.cx(0,1)

        # extract output
        psi=Statevector(qc).to_dict()
        psi=list(psi.keys())[0]
        print("x={0:d}, y={1:d}, x+y={2:s}".format(x,y,psi[0]))
```

```
x=0, y=0, x+y=0
x=0, y=1, x+y=1
x=1, y=0, x+y=1
x=1, y=1, x+y=0
```

## Swapping qubits

Swap gate SWAP is defined by SWAP$|q_1\, q_0\rangle = |q_0\, q_1\rangle$. When it acts on a superposition state, the coefficients of $|01\rangle$ and $|10\rangle$ is swapped.

$$\mathrm{SWAP}\left(c_{00}|00\rangle + c_{01}|01\rangle + c_{10}|10\rangle + c_{11}|11\rangle\right) = c_{00}|00\rangle + c_{10}|01\rangle + c_{01}|10\rangle + c_{11}|11\rangle$$

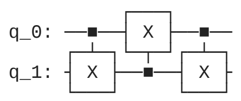
Recall that CX$^{q_1}_{q_0}$ swaps the coefficients of $|01\rangle$ and $|11\rangle$, and CX$^{q_0}_{q_1}$ swaps the coefficients of $|10\rangle$ and $|11\rangle$. Using these gates in series, we can create SWAP.

$$\begin{aligned}
&\mathrm{CX}^{q_1}_{q_0}\cdot\mathrm{CX}^{q_0}_{q_1}\cdot\mathrm{CX}^{q_1}_{q_0}\left(c_{00}|00\rangle + c_{01}|01\rangle + c_{10}|10\rangle + c_{11}|11\rangle\right)\\
=&\mathrm{CX}^{q_1}_{q_0}\cdot\mathrm{CX}^{q_0}_{q_1}\left(c_{00}|00\rangle + c_{11}|01\rangle + c_{10}|10\rangle + c_{01}|11\rangle\right)\\
=&\mathrm{CX}^{q_1}_{q_0}\left(c_{00}|00\rangle + c_{11}|01\rangle + c_{01}|10\rangle + c_{10}|11\rangle\right)\\
=&c_{00}|00\rangle + c_{10}|01\rangle + c_{01}|10\rangle + c_{11}|11\rangle
\end{aligned}$$

which indicates SWAP=$\mathrm{CX}^{q_1}_{q_0}\cdot\mathrm{CX}^{q_0}_{q_1}\cdot\mathrm{CX}^{q_1}_{q_0}$. (See the following circuit.)

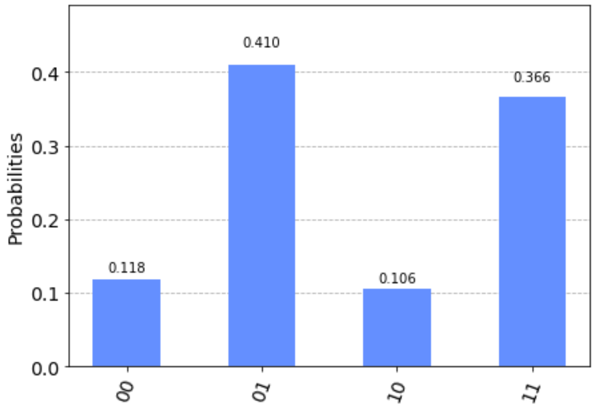
```
from qiskit import *
qr=QuantumRegister(2,'q')
qc=QuantumCircuit(qr)
qc.cx(0,1)
qc.cx(1,0)
qc.cx(0,1)

print("sawpping qubits")
qc.draw()
```

```
sawpping qubits
```



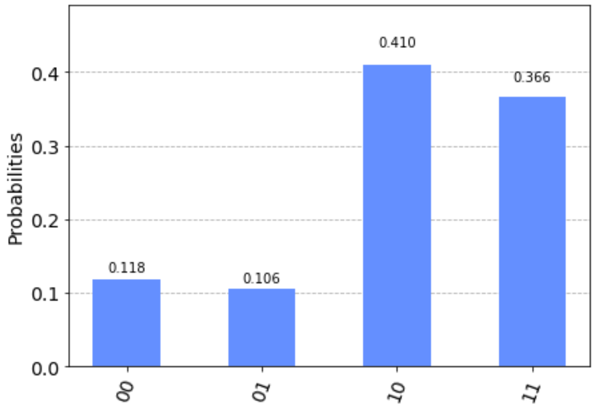
```
from qiskit import *
from qiskit.quantum_info import Statevector
import numpy as np

qr=QuantumRegister(2,'q')
qc=QuantumCircuit(qr)

# randomly oriented qubits
a=np.pi*np.random.rand()
b=np.pi*np.random.rand()
c=np.pi*np.random.rand()
qc.u(a,b,c,0)
a=np.pi*np.random.rand()
b=np.pi*np.random.rand()
c=np.pi*np.random.rand()
qc.u(a,b,c,1)

# get the statevector in dict format
psi=Statevector(qc).to_dict()

# evaluate of probabilities
p0 = dict()
for k in psi:
    p0[k]=abs(psi[k])**2

from qiskit.visualization import plot_histogram
print("probability distribution before swap")
plot_histogram(p0)
```

```
probability distribution before swap
```



```python
qc.cx(0,1)
qc.cx(1,0)
qc.cx(0,1)

psi=Statevector(qc).to_dict()

p1 = dict()
for k in psi:
    p1[k]=abs(psi[k])**2

print("probability distribution after swapping.")
plot_histogram(p1)
```

```
probability distribution after swapping.
```



Last Modified on 08/19/2022.

[1] The notation $\mathtt{cx}_{q_0}^{q_1}$ is not commonly used. In many literature, simply `cx` is used without specifying which qubit is source, causing confusion. If you are confused, see the corresponding circuit diagram.

## 8.2. Controlled- Z, S, T, P gates

Recall that that Z, S, and T gates all change the phase of $|1\rangle$ and that the general phase gate P($\theta$) can replace them as Z = P($\pi$), S = P($\pi/2$), and T = P($\pi/4$) (See Section 6.) In this section, controlled- Z, S, T, and P gates (CZ, CS, CT, and CP, respectively) are introduced. Since all of them work in the similar manner, only controlled-P is explained in most parts.

**Operational Definition**

When gate CP$_{q_0}^{q_1}(\theta)$ acts on $|q_1\, q_0\rangle$, P($\theta$) is applied to $q_1$ if $q_0 = 1$ and nothing is done otherwise. Qubit $q_0$ serves as source and $q_1$ as target. AS shwon bellow, CP$_{q_0}^{q_1}(\theta) =$ CP$_{q_1}^{q_0}(\theta)$. Hence, it is not necessary to specify a source and a target qubit.

Mathematically, it is expressed as

$$\mathrm{CP}^{q_1}_{q_0}(\theta)|q_1\, q_0\rangle = \mathrm{I} \otimes |0\rangle\langle 0| + \mathrm{P}(\theta) \otimes |1\rangle\langle 1|$$

Switching source and target qubits,

$\mathrm{CP}^{q_0}_{q_1}(\theta)|q_1\, q_0\rangle$ applies P($\theta$) to $q_0$ if $q_1 = 1$ and do nothing otherwise. Mathematically, it is expressed as

$$\mathrm{CP}^{q_0}_{q_1}|q_1\, q_0\rangle(\theta) = |0\rangle\langle 0| \otimes \mathrm{I} + |1\rangle\langle 1| \otimes \mathrm{P}(\theta)$$

**Transformation**

$\mathrm{CP}^{q_1}_{q_0}(\theta)$ and $\mathrm{CP}^{q_0}_{q_1}(\theta)$ transforms computational basis as follows:

$$\begin{aligned}
\mathrm{CP}^{q_1}_{q_0}(\theta)|00\rangle &= |00\rangle \\
\mathrm{CP}^{q_1}_{q_0}(\theta)|01\rangle &= |01\rangle \\
\mathrm{CP}^{q_1}_{q_0}(\theta)|10\rangle &= |00\rangle \\
\mathrm{CP}^{q_1}_{q_0}(\theta)|11\rangle &= e^{i\theta}|11\rangle
\end{aligned}$$

Since only $|11\rangle$ is affected, $\mathrm{CP}^{q_1}_{q_0}(\theta) = \mathrm{CP}^{q_0}_{q_1}(\theta)$.

**Matrix representation**

$$\mathrm{CP}^{q_1}_{q_0}(\theta) = \mathrm{CP}^{q_1}_{q_0}(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta} \end{bmatrix}$$

## 8.2.1. Qiskit circuit functions

The Qiskit circuit functions for CZ and CP are `cz` and `cp`, respectively and it appears in a circuit as follows. Unlike CX gate, the source and target are not distinguished.

```python
from qiskit import *
from qiskit.circuit import Parameter
from qiskit.quantum_info import Statevector
t=Parameter('\u03B8')

qr  = QuantumRegister(2)
qc = QuantumCircuit(qr)

# control CZ gate
qc.cz(0,1)

# contrl CP gate
qc.cp(t,0,1)

qc.draw()
```

```
q0_0: ─■──■──────
       │  │P(θ)
q0_1: ─■──■──────
```

Qiskit does not have standard circuit functions for CS and CT but they are predefined in `qiskit.circuit.library.standard_gates`. You just have to create a shorthand expression from the library and use `append` function to add the gate to the circuit. We can always use CP($\pi/2$) and CP($\pi/4$) for CS and CT, respectively.

```python
from qiskit import *
from qiskit.circuit.library.standard_gates import SGate, TGate

cs = SGate().control(1) # the parameter is the amount of control points you want
ct = TGate().control(1)
qr=QuantumRegister(2,'q')
qc = QuantumCircuit(qr)

qc.append(cs, [0, 1])
qc.append(ct, [0, 1])

qc.draw()
```

```
q_0: ──■────■──
     ┌─┴─┐┌─┴─┐
q_1: ┤ S ├┤ T ├
     └───┘└───┘
```

### 8.2.2. Acting on superposition state

$\mathrm{CP}_{q_0}^{q_1}(\theta)$ multiplies phase factor $e^{i\theta}$ only to $|11\rangle$. Other basis vectors are not affected. Note also that the modulus of the coefficients remain the same and thus probabilities of finding the computational basis vectors are not modified by CP.

$$\mathrm{CP}_{q_0}^{q_1}(\theta)\left(c_{00}|00\rangle + c_{01}|01\rangle + c_{10}|10\rangle + c_{11}|11\rangle\right) = c_{00}|00\rangle + c_{11}|01\rangle + c_{10}|10\rangle + e^{i\theta}c_{01}|11\rangle$$

How can we change the phase of a basis vector other than 11\tangle? A possible strategy is to transform the target basis vector to $|11\rangle$, apply CP and transform back to the original basis vector. For example, if we want to multiply $i$ to $|00\rangle$ without changing other basis vector, apply (X\otimeX) $\cdot \mathrm{CS} \cdot (\mathrm{X} \otimes \mathrm{X})$

**Example**

How can we change the phase of a basis vector other than $|11\rangle$? A possible strategy is to transform the target basis vector to $|11\rangle$, apply CP and transform back to the original basis vector. For example, if we want to multiply $i$ to $|00\rangle$ without changing other basis vector, apply $(\mathrm{X} \otimes \mathrm{X}) \cdot \mathrm{CS} \cdot (\mathrm{X} \otimes \mathrm{X})$. In this example, we start with a product state $|+\rangle \otimes |+\rangle$. Then, the above gate is applied to it. Check that $i$ is multiplied only to $|00\rangle$. The final state is entangled. (See Section 7.1.

```python
from qiskit import *
from qiskit.quantum_info import Statevector
import numpy as np

qr=QuantumRegister(2,'q')
qc = QuantumCircuit(qr)

# prepare a super position state
qc.h([0,1])

print("initial state")
Statevector(qc).draw('latex')
```

```
initial state
```

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

```python
qc.barrier()
qc.x([0,1])
qc.cp(np.pi/2,0,1)
qc.x([0,1])
qc.draw()
```

```
     ┌───┐ ░ ┌───┐         ┌───┐
q_0: ┤ H ├─░─┤ X ├─■───────┤ X ├
     ├───┤ ░ ├───┤ │P(π/2) ├───┤
q_1: ┤ H ├─░─┤ X ├─■───────┤ X ├
     └───┘ ░ └───┘         └───┘
```

```
print("final state")
Statevector(qc).draw('latex')
```

```
final state
```

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

**Exercise**

Construct a circuit that transforms $\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$ to $\frac{1}{2}(|00\rangle + |01\rangle - |10\rangle + |11\rangle)$.

## 8.3. SWAP

| input | | output |
|---|---|---|
| $\|00\rangle$ | $\Rightarrow$ | $\|00\rangle$ |
| $\|01\rangle$ | $\Rightarrow$ | $\|10\rangle$ |
| $\|10\rangle$ | $\Rightarrow$ | $\|01\rangle$ |
| $\|11\rangle$ | $\Rightarrow$ | $\|11\rangle$ |

SWAP gate can be a native gate for certain types of quantum computer through exchange interaction. Here we show a circuit equivalent to `SWAP`.

```
import numpy as np
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, Aer
from qiskit.quantum_info import Statevector
backend = Aer.get_backend('statevector_simulator')
```

Using the built-in SWAP gate: |01⟩⇒|10⟩

```
cr=ClassicalRegister(2)
qr  = QuantumRegister(2)
qc = QuantumCircuit(qr,cr)
qc.x(0)
qc.barrier()
qc.swap(0,1)
qc.measure([0,1],[0,1])
qc.draw()
```

```
        ┌───┐ ░    ┌─┐   
 q0_0: ┤ X ├─░──X─┤M├───
        └───┘ ░  │ └╥┘┌─┐
 q0_1: ──────░──X──╫─┤M├
              ░     ║ └╥┘
 c0: 2/═════════════╩══╩═
                    0  1 
```

```
result=backend.run(qc).result()
(result.get_statevector()).draw('latex')
```

$$|10\rangle$$

- Equivalent circuit to `SWAP` using `CX`

```
qr = QuantumRegister(2)
qc = QuantumCircuit(qr)
qc.x(1)
qc.cnot(0,1)
qc.cnot(1,0)
qc.cnot(0,1)
qc.draw()
```

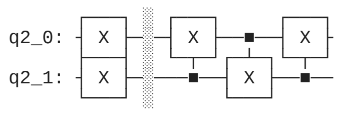
```
                  ┌───┐     
 q1_0: ───────■──┤ X ├──■──
       ┌───┐┌─┴─┐└─┬─┘┌─┴─┐
 q1_1: ┤ X ├┤ X ├──■──┤ X ├
       └───┘└───┘     └───┘
```

```
result=backend.run(qc).result()
(result.get_statevector()).draw('latex')
```

$$|01\rangle$$

Symmetric property: Swapping qubits themselves (not their states) should give the same results.

```
qr = QuantumRegister(2)
qc = QuantumCircuit(qr)
qc.x([0,1])
qc.barrier()
qc.cnot(1,0)
qc.cnot(0,1)
qc.cnot(1,0)
qc.draw()
```

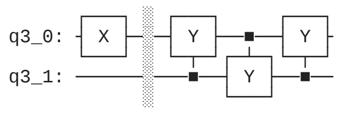


```
result=backend.run(qc).result()
(result.get_statevector()).draw('latex')
```

$$|11\rangle$$

There is no particular reason to use X direction. `CY` should be able to realize `SWAP` as well.

```
qr = QuantumRegister(2)
qc = QuantumCircuit(qr)
qc.x(0)
qc.barrier()
qc.cy(1,0)
qc.cy(0,1)
qc.cy(1,0)
qc.draw()
```



```
result=backend.run(qc).result()
(result.get_statevector()).draw('latex')
```

$$|10\rangle$$

# 9. Algorithms

TBW

## 9.1. Quantum Teleportation

Suppose that Alice has a qubit in an arbitrary state $|\psi\rangle$ carrying some information. She wants to send the information to Bob. In other words, Bob needs to obtain a qubit in the same state $|\psi\rangle$. The simple solution is to deliver the Alice's qubit to Bob. Let us assume that the Alice's qubit is not mobile and must stay with her. What can they do? First of all, Bob needs a qubit. It can be any state but let us assume it us in a "reset state" $|0\rangle$. Further, we assume that the two qubits are initially not entangled. This assumption allows Bob to bring a qubit from anywhere he likes. We apply some unitary operation (gates) to the both qubits so that the state of Bob's qubit is transformed to $|\psi\rangle$. Copying the state from Alice to Bob is not possible due to the no-cloning theorem. Applying local unitary operation (one-qubit gates) to each qubit does not transmit any information and thus the Bob's qubit is independent of the Alice's. We need non-local operation (two-qubit gates). We know such a non-local operation. The SWAP operation makes the Bob's qubit $|\psi\rangle$ and Alice loses the information entirely. Unfortunately, the SWAP operation works only when the two qubits are close to each other, hence one of them must travel to the other.

Is there a way to create a desired state over a distance by communication? Classical communication such as telephone call won't help due to the no-teleportation theorem. We must use a quantum communication. That means we need a messenger qubit that can be sent from one cite to another, such as a photon. It turns out that quantum communication along with classical communication *teleport* the state $|\psi\rangle$ from Alice to Bob over a large distance. We use three qubits, $q_A$ for Alice, $q_B$ for Bob, and $q_M$ messenger (often called *ancila* qubit). The protocol is called *quantum teleportation*.

### 9.1.1. The protocol

Consider a Hilbert space of three qubits $\mathcal{H}_A \otimes \mathcal{H}_M \otimes \mathcal{H}_B$. $q_A$ is in a arbitrary state $\alpha|0\rangle + \beta|1\rangle$, and $q_B$ and $q_M$ are both in $|0\rangle$. Thus the initial state is

$$|\psi_0\rangle = (\alpha|0\rangle + \beta|1\rangle) \otimes |00\rangle$$

We want find a protocol that makes $q_B$ becomes $(\alpha|0\rangle + \beta|1\rangle)$ at the end. We don't care the final state of $q_A$ and $q_M$.

**Step 1**

Assume that $q_M$ is near by $q_B$. Bob applies H and CX on $q_M$ and $q_B$ so that they becomes Bell state $|\Phi^+$.(See (8.1).)

$$|\psi_0\rangle = (\alpha|0\rangle + \beta|1\rangle) \otimes |00\rangle \quad \Rightarrow \quad |\psi_1\rangle = (\alpha|0\rangle + \beta|1\rangle) \otimes |\Phi^+\rangle$$

**Step 2**

Now the messenger qubit travels to Alice. In order to distinguish $\alpha$ and $\beta$, Alice applies CX$^{q_M}_{q_A}$. \$ $|\psi_1\rangle = (\alpha|0\rangle + \beta|1\rangle) \otimes |\Phi^+\rangle \quad \Rightarrow \quad |\psi_2\rangle = \alpha|0\rangle \otimes |\Phi^+\rangle + \beta|1\rangle \otimes |\Psi^+\rangle$\$

Note that $q_M$ and $q_B$ are separated by a distance, entanglement in the Bell states survives.

**Step 3**

Alice applies H on $q_A$ to change the basis of $q_A$ from the $z$-basis to $x$-basis.

$$|\psi_2\rangle = \alpha|0\rangle \otimes |\Phi^+\rangle + \beta|1\rangle \otimes |\Psi^+\rangle \quad \Rightarrow \quad |\psi_3\rangle = \alpha|+\rangle \otimes |\Phi^+\rangle + \beta|-\rangle \otimes |\Psi^+\rangle$$

Now, we rewrite $|\psi_3\rangle$ from the Alice's view

$$|\psi_3\rangle = \frac{1}{2}[|00\rangle \otimes (\alpha|0\rangle + \beta|1\rangle) + |01\rangle \otimes (\alpha|1\rangle + \beta|0\rangle) + |10\rangle \otimes (\alpha|0\rangle - \beta|1\rangle) + |11\rangle \otimes (\alpha|1\rangle$$

**Step 4**

Alice measures $q_A$ and $q_M$ in the computational basis. There are four possible outcomes, $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$ with the equal probability $\frac{1}{4}$. The state vector collapses accordingly. The outcomes of the measurement are listed in Table 9.1.

| Alice's outcome | Bob's state | Classical message |
|---|---|---|
| $\|00\rangle$ | $\alpha\|0\rangle + \beta\|1\rangle$ | Do nothing. |
| $\|01\rangle$ | $\alpha\|1\rangle + \beta\|0\rangle$ | Apply X. |
| $\|10\rangle$ | $\alpha\|0\rangle - \beta\|1\rangle$ | Apply Z. |
| $\|11\rangle$ | $\alpha\|1\rangle - \beta\|0\rangle$ | Apply Z·X |

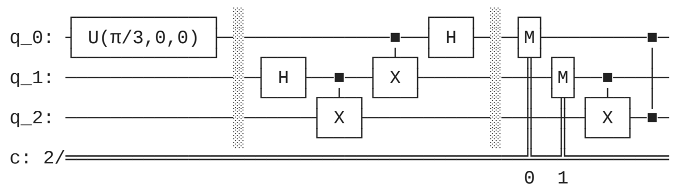
*Table 9.1* The outcome of measurement

**Step 5**

If Alice obtain $00\rangle$, then Bob's qubit is in the desired state. Otherwise, Bob needs to transform his qubit but he does not know what to do. Alice sends a message given in Table 9.1 to Bob through classical channel (such as telephone). In quantum circuit, this final adjustment can be done by applying CX$^{q_B}_{q_M}$ and CZ$^{q_B}_{q_A}$ after the measurement. Note this CX and CZ are just mimicking the classical communication.

Bob follows the message. Now, Bob has the desired state in all cases.

```
import numpy as np
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, Aer
from qiskit.quantum_info import Statevector, partial_trace, purity
backend = Aer.get_backend('statevector_simulator')

cr=ClassicalRegister(2,'c')
qr=QuantumRegister(3,'q')
qc=QuantumCircuit(qr,cr)

# Generate a state Alice has
# set parameters
theta=np.pi/3
phi=0.0
qc.u(theta,phi,0,0)

psi0=Statevector(qc)

qc.barrier()

# Step 1
qc.h(1)
qc.cx(1,2)

# Step 2
qc.cx(0,1)

# Step 3
qc.h(0)
qc.barrier()

# Step 4
qc.measure([0,1],[0,1])

# Step 5
qc.cx(1,2)
qc.cz(0,2)

qc.draw()
```



```
result=backend.run(qc).result()
full_statevector = result.get_statevector()
```

```
# get the density matrix for the first qubit by taking the partial trace
print("Alice's initial state")
rhoA = partial_trace(psi0, [1,2])
print("purity=",purity(rhoA))
psiA = Statevector(np.sqrt(np.diagonal(rhoA)))
psiA.draw('latex')
```

```
Alice's initial state
purity= (1.0000000000000002+0j)
```

$$\frac{\sqrt{3}}{2}|0\rangle+\frac{1}{2}|1\rangle$$

```
# get the density matrix for the first qubit by taking the partial trace
print("Bob's initial state")
rhoB = partial_trace(full_statevector, [0,1])
print("purity=",purity(rhoB))
psiB = Statevector(np.sqrt(np.diagonal(rhoB)))
psiB.draw('latex')
```

```
Bob's initial state
purity= (1.0000000000000002+0j)
```

$$\frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle$$

The result shows that the state of Bob's qubit is pure and identical to the original state of Alice's qubit.

## 9.2. Walsh-Hadamard Transformation

An integer $j$ can be expressed in binary bits $j_{n-1}\, j_{n-2}\, \cdots\, j_0$ such that

$$j = 2^{n-1} j_{n-1} + 2^{n-2} j_{n-2} + \cdots + j_0 = \sum_{k=0}^{n-1} 2^k j_j$$

where $j_k = \in \{0, 1\}$. The smallest and the largest integers expressed with $n$ bits are $0$ and $2^n - 1$. We can encode integers in quantum computation in the same way. Replacing the bits with qubits $j_k$,

$$|j\rangle_n = |j_{n-1}\, j_{n-2}\, \cdots\, j_0\rangle = |j_{n-1}\rangle \otimes |j_{n-2}\rangle \otimes \cdots \otimes |j_0\rangle = \bigotimes_{k=0}^{n-1} |j_k\rangle$$

where $|\cdot\rangle_n$ is a ket in a $n$-dimensional Hilbert space. For example, integers from 0 to 15 can be encoded in with qubits. Integer $9$ is expressed as $|9\rangle_4 = |0\rangle \otimes |1\rangle \otimes |0\rangle \otimes |1\rangle$.

An advantage of quantum computation is quantum parallelism using a super position state. It would be quite useful if we can create a superposition of many intergers

$$|\psi\rangle = \frac{1}{\sqrt{2^n}} \big(|0\rangle_n + |1\rangle_n + \cdots + |2^{n-1}\rangle_n\big).$$

### 9.2.1. Walsh-Hadamard transformation

To find a quantum algorithm, we look at a few small cases. For $n = 1$, the target state is $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. We are already familiar with this state and $H|0\rangle$ is the solution. For $n = 2$, we have

$$\begin{aligned} \frac{1}{2}(|0\rangle_2 + |1\rangle_2 + |2\rangle_2 + |3\rangle_2) &= \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) \\ &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ &= H|0\rangle \otimes H|0\rangle = (H \otimes H)|0\rangle_2 \end{aligned}$$

It is now clear that the desired quantum algorithm for general case is

$$\begin{aligned} |\psi\rangle &= (H \otimes H \otimes \cdots \otimes H)|0\rangle_n \\ &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \cdots \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \end{aligned} \tag{9.1}$$

which is known as *Walsh-Hadamard* transform.

**Example** 9.2.1 The following example calculate the Wals-Hadamard transform for $n = 3$. The result should be $\frac{1}{\sqrt{8}}(|000\rangle + |001\rangle + \cdots + |110\rangle + |111\rangle)$.

```python
import numpy as np
from qiskit import *

qr=QuantumRegister(3,'q')
qc=QuantumCircuit(qr)

qc.h(range(3))

qc.draw()
```

```
     ┌───┐
q_0: ┤ H ├
     ├───┤
q_1: ┤ H ├
     ├───┤
q_2: ┤ H ├
     └───┘
```

```python
# Show the result of Walsh-Hadamard transform
from qiskit.quantum_info import Statevector
psi=Statevector(qc)
psi.draw('latex')
```

$$\frac{\sqrt{2}}{4}|000\rangle + \frac{\sqrt{2}}{4}|001\rangle + \frac{\sqrt{2}}{4}|010\rangle + \frac{\sqrt{2}}{4}|011\rangle + \frac{\sqrt{2}}{4}|100\rangle + \frac{\sqrt{2}}{4}|101\rangle + \frac{\sqrt{2}}{4}|110\rangle + \frac{\sqrt{2}}{4}$$

### 9.2.2. Remarks

1. Recalling that the Hadamard gate changes the basis set, from the computational basis to the $x$-basis. The Walsh-Hadamard transformation is simply $|00\cdots 00\rangle \Rightarrow |++\cdots++\rangle$. Interestingly, the simple product of $|+\rangle$ corresponds to the superposition of integer states. Quantum algorithms use this kind of *tricks* everywhere.
2. The Walsh-Hadamard transformation generates the superposition state where all terms have the same phase. Applying phase shifting gates such as $Z$, $S$, $T$, and $P$, you can modify the phases. Quantum Fourier transform is an example. See Section 9.5.

---

Last modified on 07/23/2022.

## 9.3. A little example of quantum parallelism

Quantum computer can calculate many different instances simultaneously. Here we consider a simple example. Suppose that we want to calculate $z = x \oplus y$ where $x,\ y,\ z \in \{0, 1\}$. The variables are assigned to three qubits as $|zyx\rangle = |z\rangle \otimes |y\rangle \otimes |x\rangle$. Initially, $z = 0$. The computation corresponds to transformation $|0\rangle \otimes |y\rangle \otimes |x\rangle \Rightarrow |x \oplus y\rangle \otimes |y\rangle \otimes |x\rangle$. There are four possible instances:

$$\begin{array}{ccc}
|000\rangle & \Rightarrow & |000\rangle \\
|001\rangle & \Rightarrow & |101\rangle \\
|010\rangle & \Rightarrow & |110\rangle \\
|011\rangle & \Rightarrow & |011\rangle
\end{array}$$

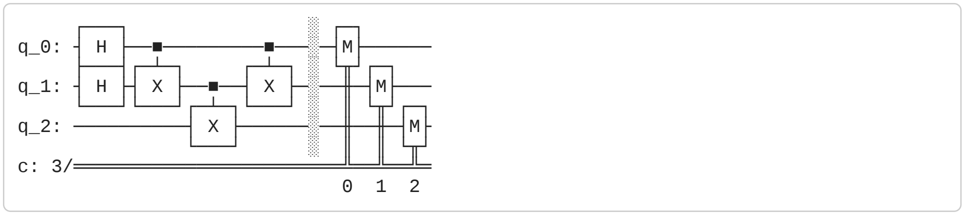
We want to compute all these four cases at once. We use a superposition state of the four input states. First we create four possible input states with equal weight and then we compute $x \oplus y$ for each term:

$$\begin{aligned}
|000\rangle &\Rightarrow |0\rangle \otimes \frac{1}{2}(|0\rangle \otimes |0\rangle + |0\rangle \otimes |1\rangle + |1\rangle \otimes |0\rangle + |1\rangle \otimes |1\rangle) \\
&\Rightarrow \frac{1}{2}(|000\rangle + |101\rangle + |110\rangle + |011\rangle)
\end{aligned}$$

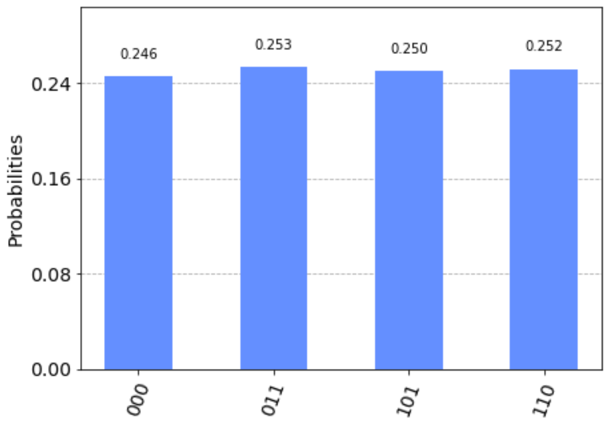
Note that the output does not include all possible states. For example $|111\rangle$ does not exist since $1 \oplus 1 \neq 1$.

The first step can be done with the Walsh-Hadamard transformation. The addition can be done with thee $CX$ gates as shown in XXX. The following circuit calculates four cases simultaneously.

```
from qiskit import *

cr=ClassicalRegister(3,'c')
qr=QuantumRegister(3,'q')
qc=QuantumCircuit(qr,cr)

qc.h([0,1])
qc.cx(0,1)
qc.cx(1,2)
qc.cx(0,1)
qc.barrier()
qc.measure(qr,cr)
qc.draw()
```



```
# Chose a general quantum simulator without noise.
# The simulator behaves as an ideal quantum computer.
backend = Aer.get_backend('qasm_simulator')

# set number of tries
nshots=8192

# execute the quantum circuit and store the outcome
job = backend.run(qc,shots=nshots)

# extract the result
result = job.result()

# count the outcome
counts = result.get_counts()

from qiskit.visualization import plot_histogram
plot_histogram(counts)
```



Only four states are obtained and each shows $z = x \oplus y$. We indeed calculated four different cases at once thanks to the quantum parallelism.

Last modified: 07/31/2022

## 9.4. The Deutsch problem

We are going to solve a kind of classification problem discussed originally by Deutsch [4]. The problem itself is rather trivial and not really useful. However, it demonstrates an advantage of quantum computation over classical computation.

### 9.4.1. The problem

Consider a binary function $x \mapsto f(x)$ where $x \in \{0,1\}$ and $f(x) \in \{0,1\}$. There are only four possible functions, $f_1,\ f_2,\ f_3,\ f_4$.

| **x** | **$f_1$** | **$f_2$** | **$f_3$** | **$f_4$** |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |

An oracle possess one of these functions. We can ask it questions like "what is the value of $f(1)$?" To find what function it has, we need to ask two questions, "what are $f(0)$ and $f(1)$?". Now, let us try a simpler problem. The outputs of $f_1$ and $f_2$ are always the same regardless of the input values. So, we call them *constant* function. On the other hand, the outputs of $f_3$ and $f_4$ contain both $0$ and $1$. We shall call them *balanced* function. Now, our task is to determine which type of the functions the oracle possess, constant or balanced. Hence, this is a classification task. Only we need to find out is if $f(0) = f(1)$ or $f(0) \neq f(1)$. How many questions do we need to ask? Although we don't have to identify the function, it seems that we still need to ask two questions. It turns out that the Deutsch's quantum algorithm requires only one question.

## 9.4.2. Encoding the functions

Before solving the problem, we need to find out how to express a function using gates. Every gate is a unitary operator and thus the operation is reversible. While $f_3$ and $f_4$ are bijective (invertible), $f_1$ and $f_2$ are even not surjective and thus not invertible. Here we use a two-qubit gate $U_f$ acting on $|q_0\rangle \otimes |q_1\rangle$ as defined by

$$U_f |x\rangle \otimes |y\rangle = |x\rangle \otimes |y \oplus f(x)\rangle$$

where $\oplus$ is modulo-2 addition. The qubit $q_0$ contains the value of $x$ and its value is not affected by the gate. The second qubit $q_1$ is transformed from $y$ to $y \otimes f(x)$. When $y = 0$, the output is simply the function value $f(x)$. When $y = 1$, $1 \oplus f(x) = 1 - f(x)$. By knowing the value of $y$ and $y \oplus f(x)$, we can find $f(x)$.

Does such a gate exist? If the gate is invertible, we can construct a unitary operator. Noting that $z \oplus z = 0$ for any $z$,

$$U_f |x\rangle \otimes |y \oplus f(x)\rangle = |x\rangle \otimes |y \oplus f(x) \oplus f(x)\rangle = |x\rangle \otimes |y\rangle$$

indicating that $U_f^{-1} = U_f$. We just make it sure that $U_f$ is self-adjoint. Here we show the actual gates:

---

- $f_1(x)$

Since $f(x) = 0$,

$$U_{f_1} |x\rangle \otimes |y\rangle = |x\rangle \otimes |y \oplus 0\rangle = |x\rangle \otimes |y\rangle$$

Hence, $U_{f_1} = I$ (identity operator).

---

- $f_2(x)$

Since $f(x) = 1$

$$U_{f_2} |x\rangle \otimes |y\rangle = |x\rangle \otimes |y \oplus 1\rangle = |x\rangle \otimes (X|y\rangle) = (I \otimes X)|x\rangle \otimes |y\rangle$$

Hence, $U_{f_2} = I \otimes X$.

---

- $f_3(x)$

Since $f(x) = x$,

$$U_{f_3} |x\rangle \otimes |y\rangle = |x\rangle \otimes |y \oplus x\rangle$$

The output is $|0\rangle \otimes |y\rangle$ if $x = 0$ and $|1\rangle \otimes X|y\rangle >$ if $x = 1$. Hence $x$ is a control bit and $y$ flips only when $x = 1$. This is not the standard control gate. We need to flip $q_0$ before and after applying the control-$X$, which is $U_{f_3} = (X \otimes I) \cdot CX_{q_0,q_1} \cdot (X \otimes I)$.

- $f_4(x)$

Since $f(x) = 1 - x$,

$$U_{f_4}|x\rangle \otimes |y\rangle = |x\rangle \otimes |y \oplus (1-x)\rangle$$

The output is $0\rangle \otimes X|y\rangle$ if $x = 0$ and $|0\rangle \otimes |y\rangle >$ if $x = 1$. Hence $x$ is a control bit and $y$ flips only when $x = 0$. Thus, $U_{f_4} = CX_{q_0,q_1}$.

## 9.4.3. No-so smart algorithm

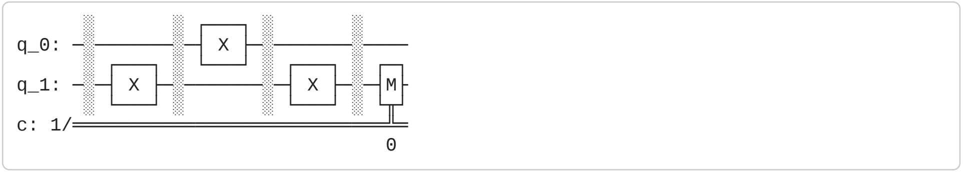
We just showed that how to encode the functions. However, that is not our task. The oracle has one of the above gates $U_{f_i}$. We want to know if it is constant or balanced.

Let's us begin with a not-so-smart method. We just ask two questions. That is to apply $U_f$ twice, the first one for $x = 0$ and the second for $x = 1$.

$$|0\rangle \otimes |0\rangle \xrightarrow{U_f} |0\rangle \otimes |f(0)\rangle \xrightarrow{X\otimes I} |1\rangle \otimes |f(0)\rangle\rangle \xrightarrow{U_f} |1\rangle \otimes |f(0) \oplus f(1)\rangle$$

Now, we measure $q_1$. The outcome is $f(0) \oplus f(1)$ with probability 1 (no error). The oracle has a constant function if the $f(0) \oplus f(1) = 0$ and a balanced function if $f(0) \oplus f(1) = 1$.

```python
# Define functions
import numpy as np

from qiskit import *

cr=ClassicalRegister(1,'c')
qr=QuantumRegister(2,'q')
qc=QuantumCircuit(qr,cr)

#chose a function at random
k=np.random.randint(1,4)
if k==1:
    oracle = "constant"
elif k==2:
    oracle = "constant"
elif k==3:
    oracle = "balanced"
elif k==4:
    oracle = "balanced"

# Define Uf (Only Oracle knows it)
def Uf(k):

    # Uf appears between barriers
    qc.barrier([qr[0],qr[1]])

    if k==1:
        qc.i([qr[0],qr[1]])
        oracle = "constant"
    elif k==2:
        qc.x(qr[1])
        oracle = "constant"
    elif k==3:
        qc.x(qr[0])
        qc.cx(qr[0],qr[1])
        qc.x(qr[0])
        oracle = "balanced"
    elif k==4:
        qc.cx(qr[0],qr(1))
        oracle = "balanced"

    qc.barrier([qr[0],qr[1]])


# Consturct circuit

Uf(k)
qc.x(qr[0])
Uf(k)
qc.measure(qr[1],cr[0])
qc.draw()
```



```python
# Chose a general quantum simulator without noise.
# The simulator behaves as an ideal quantum computer.
backend = Aer.get_backend('qasm_simulator')

# set number of tries
nshots=8192

# execute the quantum circuit and store the outcome
job = backend.run(qc,shots=nshots)

# extract the result
result = job.result()

# count the outcome
counts = result.get_counts()

# evaluate the probability
p0=counts.get('0',0)/nshots
p1=counts.get('1',0)/nshots

if p0 > p1:
    answer = "constant"
elif p0 < p1:
    answer = "balanced"
else:
    answer = "unknown"

print("Quantum computer has found that the function is",answer,".")

if oracle==answer:
    print("Correct! Oracle has",oracle,".")
else:
    print("Wrong! Oracle has",oracle,".")
```

```
Quantum computer has found that the function is constant .
Correct! Oracle has constant .
```

## 9.4.4. The Deutsch's algorithm

Here is a challenge. Can we solve the problem using $U_f$ only once? In the previous algorithm, we use $U_f$ twice, first for $x=0$ and second for $x=1$. The key idea is that $y$ can be in a superposition state. If $|y\rangle = |-\rangle$, what would be the answer from the oracle?

Let us try to find how the oracle responds.

$$U_f|x\rangle \otimes |-\rangle = |x\rangle \otimes \frac{1}{\sqrt{2}}(|f(x)\rangle - |1 \oplus f(x)\rangle) = (-1)^{f(x)}|x\rangle \otimes |-\rangle$$

Depending on the value of $f(x)$, the gate does nothing or changes the phase. This trick is known as *phase kickback*.

Now, we replace $|x\rangle$ with another superposition state $|+\rangle$.

$$U_f|+\rangle \otimes |-\rangle = \frac{1}{\sqrt{2}}\left[(-1)^{f(0)}|0\rangle \otimes -\rangle + (-1)^{f(1)}|1\rangle \otimes |-\rangle\right] \tag{9.2}$$

If the function is constant, $f(0)=f(1)$. Then, Eq. (9.2) becomes

$$U_f|+\rangle \otimes |-\rangle = (-1)^{f(0)}|+\rangle \otimes |-\rangle \tag{9.3}$$

In it is balanced, $f(0) \neq f(1)$, then we have

$$U_f|+\rangle \otimes |-\rangle = \pm|-\rangle \otimes |-\rangle \tag{9.4}$$

Apart from the global phase, the difference between Eqs. (9.3) and (9.4) is the state of $q_0$. If it is $+\rangle$, then the answer is constant. If it is $|-\rangle$ then, the answer is balanced. Since we can measure only in the computational basis, we transofrm $\pm\rangle$ by a Hadamard gate.

Here is the summary of the Deustch's algorithm.

1. Prepare $|+\rangle \otimes |-\rangle$.
2. Apply $U_f$.
3. Apply $H \otimes I$
4. Measure $q_0$

We use $U_f$ only once! Notice that we calculate $x = 0$ and $x = 1$ simultaneously using the super position state.

```
cr=ClassicalRegister(1,'c')
qr=QuantumRegister(2,'q')
qc=QuantumCircuit(qr,cr)

qc.x(qr[1])
qc.h([qr[0],qr[1]])
Uf(k)
qc.h(qr[0])
qc.measure(qr[0],cr[0])

qc.draw()
```

```
     ┌───┐           ░           ░ ┌───┐┌─┐
q_0: ┤ H ├───────────░───────────░─┤ H ├┤M├
     ├───┤┌───┐      ░      ┌───┐░ └───┘└╥┘
q_1: ┤ X ├┤ H ├──────░──────┤ X ├░───────╫─
     └───┘└───┘      ░      └───┘░       ║
c: 1/════════════════════════════════════╩═
                                         0
```

```
# Chose a general quantum simulator without noise.
# The simulator behaves as an ideal quantum computer.
backend = Aer.get_backend('qasm_simulator')

# set number of tries
nshots=8192

# execute the quantum circuit and store the outcome
job = backend.run(qc,shots=nshots)

# extract the result
result = job.result()

# count the outcome
counts = result.get_counts()

p0=counts.get('0',0)/nshots
p1=counts.get('1',0)/nshots

if p0 > p1:
    answer = "constant"
elif p0 < p1:
    answer = "balanced"
else:
    answer = "unknown"

print("The function is found to be",answer, "( oracle actually has",oracle,")")
```

```
The function is found to be constant ( oracle actually has constant )
```

## 9.5. Quantum Fourier Transform

Fourier transform (FT) is a ubiquitous mathematical tool used in science, engineering and beyond and we are using it every day without knowing it. Whenever digital signals are processed, the Fourier transform is most likely used including WiFi, cell phone, digital music, digital picture, …, to name a few. We wish to calculate the Fourier transform on a quantum computer. It turns out that quantum computers are not good at calculating general Fourier transform as explained below. We will focus on a very special case not for the sake of Fourier transform but for other quantum algorithms such as phase estimation.

### 9.5.1. Discrete Fourier Transform

The digital signal processes use a particular form of Fourier transform known as *discrete Fourier transform* defined by

$$y_n = \frac{1}{\sqrt{N}} \sum_{m=0}^{N-1} x_m e^{2\pi i\, n\, m/N} \tag{9.5}$$

or writing ti in a matrix from

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix} = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & e^{2\pi i/N} & \cdots & e^{2\pi i(N-1)/N} \\ \vdots & \vdots & \cdots & \vdots \\ 1 & e^{2\pi i(N-1)/N} & \cdots & e^{2\pi i(N-1)^2/N} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix} \tag{9.6}$$

where $x_m \in \mathbb{C}$ is the original signal and $y_n \in \mathbb{C}$ is its Fourier transform.

Writing the column vectors in kets as

$$|x\rangle \doteq \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix}, \qquad |y\rangle \doteq \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix}$$

and the matrix as an operator

$$\mathcal{F}_N \doteq \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & e^{2\pi i/N} & \cdots & e^{2\pi i(N-1)/N} \\ \vdots & \vdots & \cdots & \vdots \\ 1 & e^{2\pi i(N-1)/N} & \cdots & e^{2\pi i(N-1)^2/N} \end{bmatrix}$$

which we shall call *Fourier operator* or *Fourier gate*. Now, the Fourier transform can be written in an abstract form

$$|y\rangle = \mathcal{F}_N |x\rangle. \tag{9.7}$$

It is easy to check that $\mathcal{F}_N \mathcal{F}_N^\dagger = 1$ and thus $\mathcal{F}_N$ is unitary. The inverse Fourier transform is done by $\mathcal{F}^\dagger$ as

$$|x\rangle = \mathcal{F}_N^\dagger |y\rangle. \tag{9.8}$$

A unitary transformation in $N$-dimensional Hilbert space can be implemented as a quantum transformation by 1) preparing a quantum state $|x\rangle$ using a certain basis set, 2) constructing the unitary operator, and 3) reading out $|y\rangle$.
However, we have a couple of issues here. First of all, quantum measurement only gives us $|y_n|$ through the Born rule. This is not necessarily a fatal limitation since many applications need only the modulus such as power spectrum. Secondly, when $N$ is very large as in the engineering applications, there is no good way to prepare $|x\rangle$ and also finding all $|y_n|$ requires a huge number of quantum measurement. Unfortunately, the quantum version of DFT seems not feasible for traditional applications.

Nevertheless, the quantum Fourier transform has several useful application such as phase estimation and period finding. It is worth developing a quantum algorithm for $\mathcal{F}_N$. To do so, we consider the Fourier transform of the computational basis $|u_j\rangle,\ j = 0, \cdots 2^{n-1}$ where $n$ is the number of qubits. Writing them explicitly in the computational bases,

$$|u_j\rangle = |j_n\, j_{n-1}\, \cdots\, j_1\rangle \equiv |j_n\rangle \otimes |j_{n-1}\rangle \otimes \cdots \otimes |j_1\rangle$$

where the individual qubit takes values $j_i \in \{0, 1\}$ and $j$ is decimal expression of the binary string $j_1\, j_2, \cdots\, j_n$, that is

$$j = 2^{n-1}j_n + 2^{n-2}j_{n-1} + \cdots + j_1 = \sum_{\ell=1}^{n} 2^{\ell-1} j_\ell \tag{9.9}$$

For $n = 2$, there are four basis kets $|u_0\rangle = |00\rangle$, $|u_1\rangle = |01\rangle$, $|u_2\rangle = |10\rangle$, $|u_3\rangle = |11\rangle$.

When the Fourier operator is applied to the computational basis we obtain a new basis

$$|w_j\rangle = \mathcal{F}_N |u_j\rangle. \tag{9.10}$$

We can construct a quantum circuit for the Fourier operator $\mathcal{F}_N$ from Eq. (9.10). Once we obtained the circuit or the Fourier gate, we can apply it to other states.
In literature “quantum Fourier transform” often means Eq.(9.10) or the Fourier gate $\mathcal{F}_N$ rather than the general Fourier transform (9.7).

**Exercise** 9.5.1  Show that $|k_j\rangle$ forms an orthonormal basis set.

### 9.5.2. The QFT algorithm

Now we construct the Foureir operator. First, we compute Eq. (9.10) explicitly in the computational basis.

**N=2**
Let us consider $n = 1$, the simplest QFT transforms the computational basis $0\rangle$ and $|1\rangle$ as

$$\begin{aligned}\mathcal{F}_2|u_0\rangle &= \mathcal{F}_2|0\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle + e^{i0}|1\rangle\big) = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ \mathcal{F}_2|u_1\rangle &= \mathcal{F}_2|1\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle + e^{i\pi}|1\rangle\big) = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle).\end{aligned} \tag{9.11}$$

We want to find a quantum circuit that does the same transformation. Recalling that

$$\begin{aligned}H|0\rangle &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ H|1\rangle &= \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\end{aligned} \tag{9.12}$$

By direct comparison between Eqs. (9.11) and (9.12), it is clear that $\mathcal{F}_2 = H$. We learned in Section 6.9 that the Hadamard gate transforms the computational basis to the $x$-basis. That transformation turned out to be the lowest order QFT and we have already computed it. See Section 6.9.2.

**N=4**

We need more examples to get an idea of $\mathcal{F}_N$.. Let us try $n = 2$.

$$\begin{aligned}
\mathcal{F}_2|u_0\rangle &= \mathcal{F}_2|00\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) \\
&= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\
\mathcal{F}_2|u_1\rangle &= \mathcal{F}_2|01\rangle = \frac{1}{2}\left(|00\rangle + e^{i\pi/2}|01\rangle + e^{i\pi}|10\rangle + e^{i3\pi/2}|11\rangle\right) \\
&= \frac{1}{2}(|00\rangle + i|01\rangle - |10\rangle - i|11\rangle) \\
&= \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle) \\
\mathcal{F}_2|u_2\rangle &= \mathcal{F}_2|10\rangle = \frac{1}{2}\left(|00\rangle + e^{i\pi}|01\rangle + e^{2i\pi}|10\rangle + e^{i3\pi}|11\rangle\right) \\
&= \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle) \\
&= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\
\mathcal{F}_2|u_3\rangle &= \mathcal{F}_2|11\rangle = \frac{1}{2}\left(|00\rangle + e^{i3\pi/2}|01\rangle + e^{3i\pi}|10\rangle + e^{i9\pi/2}|11\rangle\right) \\
&= \frac{1}{2}(|00\rangle - i|01\rangle - |10\rangle + i|11\rangle) \\
&= \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)
\end{aligned} \tag{9.13}$$

The corresponding quantum circuit seems complicated since phase factor $\pm i$ are now involved. We recall that $S$ gate does the job.

$$\begin{aligned}
S \cdot H|0\rangle &= \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle) \\
S \cdot H|1\rangle &= \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)
\end{aligned} \tag{9.14}$$

Using eqs. (9.12) and (9.14), we try the following transformations

$$\begin{aligned}
H \otimes H|00\rangle &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\
(S \cdot H) \otimes H|01\rangle &= \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\
H \otimes H|10\rangle &= \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\
(S \cdot H) \otimes H|11\rangle &= \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)
\end{aligned}$$

The results are close to the Fourier transform but the order of the output bits are wrong. We need to swap the qubits. Another issue is that $S$ gates is applied to $|01\rangle$ and $|11\rangle$. Noting that the second qubit is $1\rangle$. Hence, we can use the controlled-S gate with the second qubit as the control.

$$\begin{aligned}
SWAP \cdot (H \otimes I) \cdot CS \cdot (I \otimes H)|00\rangle &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\
SWAP \cdot (H \otimes I) \cdot CS \cdot (I \otimes H)|01\rangle &= \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle) \\
SWAP \cdot (H \otimes I) \cdot CS \cdot (I \otimes H)|10\rangle &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\
SWAP \cdot (H \otimes I) \cdot CS \cdot (I \otimes H)|11\rangle &= \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)
\end{aligned} \tag{9.15}$$

Now, the Fourier transform (9.13) and The chirchuit (9.15) produces the same outputs. Hence, $\mathcal{F}_4 = SWAP \cdot (H \otimes I) \cdot CS \cdot (I \otimes H)$.

```python
from qiskit import *
from qiskit.circuit.library.standard_gates import SGate
qr=QuantumRegister(2,'q')
qc=QuantumCircuit(qr)
cs=SGate().control(1)
qc.h(1)
qc.append(cs,[0,1])
qc.h(0)
qc.swap(0,1)
qc.draw()
```

```
                 ┌───┐
q_0: ──────■─────┤ H ├─X─
     ┌───┐┌─┴─┐  └───┘ │
q_1: ┤ H ├┤ S ├────────X─
     └───┘└───┘
```

In the following example, all four transformations (9.11) are computed using Qiskit.

```python
import numpy as np
from qiskit import *

# qiskit does not have predefined controlled-S gate
# In this application we use an equivalent gate controlled-P gate.

def QFT():
    # define the Fourier gate
    qc.h(1)
    qc.cp(np.pi/2,0,1)
    qc.h(0)
    qc.swap(0,1)
```

```python
# for |00>
qr=QuantumRegister(2,'q')
qc=QuantumCircuit(qr)

# apply the Fourier gate
QFT()

# Show the Fourier transform
from qiskit.quantum_info import Statevector
psi=Statevector(qc)
print("Fourier transform of |00>")
psi.draw('latex')
```

```
Fourier transform of |00>
```

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

```python
# for |01>
qr=QuantumRegister(2,'q')
qc=QuantumCircuit(qr)

qc.x(0)
QFT()
psi=Statevector(qc)
print("Fourier transform of |01>")
psi.draw('latex')
```

```
Fourier transform of |01>
```

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

```python
# for |10>
qr=QuantumRegister(2,'q')
qc=QuantumCircuit(qr)

qc.x(1)
QFT()
psi=Statevector(qc)
print("Fourier transform of |10>")
psi.draw('latex')
```

```
Fourier transform of |10>
```

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

```
# for |11>
qr=QuantumRegister(2,'q')
qc=QuantumCircuit(qr)

qc.x([0,1])
QFT()
psi=Statevector(qc)
print("Fourier transform of |10>")
psi.draw('latex')
```

```
Fourier transform of |10>
```

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

**Higher order QFT**

Now we have some good idea about QFT. Now, we try to find $\mathcal{F}_N$. The derivation is a bit complicated but the extension of the $N = 4$ case. First, we evaluate $\mathcal{F}_N|j\rangle$.

$$\begin{aligned}
\mathcal{F}_N|j\rangle = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi i\, j\, k/2^n} |k\rangle \\
&= \sum_{k=0}^{N-1} e^{2\pi i\, j\, \sum_{\ell=1}^{n} k_\ell/2^\ell} |k_1\, k_2\, \cdots\, k_n\rangle \\
&= \left( \sum_{k_1=0}^{1} e^{2\pi i\, j\, k_1/2} |k_1\rangle \right) \otimes \left( \sum_{k_2=0}^{1} e^{2\pi i\, j, k_2/2^2} |k_2\rangle \right) \otimes \cdots \otimes \left( \sum_{k_n=}^{1} \right. \\
&= \Big( |0\rangle + e^{2\pi i\, j/2} |1\rangle \Big) \otimes \Big( |0\rangle + e^{2\pi i\, j/2^2} |1\rangle \Big) \otimes \cdots \otimes \Big( |0\rangle + e^{2\pi}
\end{aligned}$$

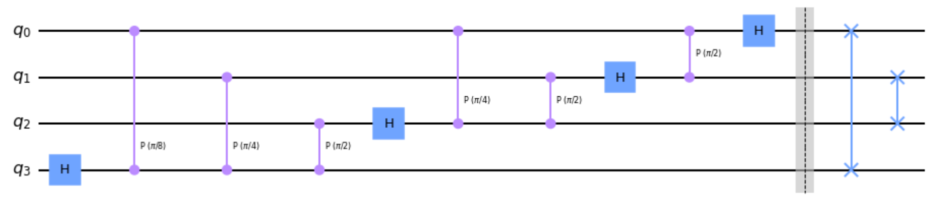
The results look quite similar to Eq. (9.13). Only the difference is the phase angles. The question is how we calculate the phase angle. Let us take a closer look at a phase factor $e^{2\pi i\, j/2^k}$, $1 \leq k \leq n$. Using Eq. (9.9),

$$e^{2\pi i\, j/2^k} = e^{2\pi i\, \sum_{\ell=1}^{n} 2^{\ell-1-k} j_\ell} = e^{2\pi i, j_1/2^k} e^{2\pi i\, j_2/2^{k-1}} \cdots e^{2\pi i\, j_k/2} e^{2\pi i j_{k+1}} \cdots e^{2\pi i\, j_n (n-1-k)} = e^{2\pi i\, j_1/2^k} e$$

We need to apply the phase rotation of angle $k$ different phase angle $2\pi j_\ell/2^{ell-1-k}$, $1 \leq \ell \leq k$. No rotation is needed when $\ell > k$ or $j_\ell = 0$. Hence, we apply the controlled phase gate $CP(2\pi/2^{\ell-1-k})$ only when $j_\ell = 0$. In conclusion, apply $H$ to $k$-th qubit and apply controlled-$P$ gate $k$ times with appropriate angles. Repeat this from $k = n$ to $k = 1$ in the backward so that source qubits of controlled $P$ gate is not modified before the controlled operation. As you saw in $\mathcal{F}_4$, the order of the qubits is wrong after these operations. Hence, we reverse the order by applying swap gates.

The following circuit is equivalent to $\mathcal{F}_8$

```
import numpy as np
from qiskit import *
qr=QuantumRegister(4,'q')
qc=QuantumCircuit(qr)

# qubit 3
qc.h(3)
qc.cp(np.pi/8,0,3)
qc.cp(np.pi/4,1,3)
qc.cp(np.pi/2,2,3)

# qubit 2
qc.h(2)
qc.cp(np.pi/4,0,2)
qc.cp(np.pi/2,1,2)

# qubit 1
qc.h(1)
qc.cp(np.pi/2,0,1)

# qubit 0
qc.h(0)

qc.barrier(range(4))

# reordering
qc.swap(0,3)
qc.swap(1,2)

qc.draw('mpl')
```



Last modified: 07-19-2022

## 9.6. Quantum Phase Estimation

The problem of quantum phase estimation (QPE) is just a special kind of eigenvalue problem. However, it plays a quite important role in quantum computation. It can be considered as a subroutine used in many useful quantum algorithms such as factorization, quantum walks, … So, we need to learn it before other quantum algorithms.

### 9.6.1. The problem

Consider a unitary operator $U$ in a $2n$-dimension Hilbert space $(\mathbb{C}_2)^{\otimes n}$. Its eigenvector $|psi_\lambda\rangle$ satisfies the eigenvalue equation $U|\psi_\lambda\rangle = \lambda|\psi_\lambda\rangle$ where $\lambda$ is the eigenvalue. The adjoint of the eigenvalue equation is $\langle\psi_\lambda|U^\dagger = \langle\psi_\lambda|\lambda^*$. The inner product of the two eigenvalue equations is $\langle\psi_\lambda|U^\dagger U|\psi_\lambda\rangle = |\lambda|^2\langle\psi_\lambda|\psi_\lambda\rangle$. Since $U$ is unitary, $U \dagger U = I$ and thus $|\lambda| = 1$. So, the eigenvalue equation can be written as

$$U|\psi_\theta\rangle = e^{2\pi i\theta}|\psi_\theta\rangle$$

where $|\psi_\theta\rangle$ is the eigenvector and the corresponding eigenvalue is $e^{2\pi i\theta}$. Our task is to find the phase variable $\theta \in [0, 1)$ for a given $|\psi_\theta$. Mathematically, it is a trivial problem. Assuming $|\psi_\theta\rangle$ is normalized, $e^{2\pi i\theta} = \langle\psi_\theta|U|\psi_\theta\rangle$, which current quantum computer cannot compute. Can we find $\theta$ without computing the inner product?

### 9.6.2. Encoding a continuous number between 0 and 1

The value we want to find is not integer. How can we encode a continuous number between 0 and 1 in qubits? Since it is not possible to encode a true continuous number in digital computers, we approximate it. In Section 9.2 we encoded integers between $0$ and $2^n - 1$ in $n$ qubits as

$|j\rangle_n = |j_{n-1}\, j_{n-2}\, \cdots\, j_0\rangle$. Noting that $0 \le 2^n\theta < 2^n$, we can encode $2^n\theta$ as an integer state $|j\rangle_n$. Then, $\theta = j/2^n$, which is not continuous and there is a gap of $2^{-n}$. The gap decreases quickly as $n$ increases. If an accurate value of $\theta$ is needed, we must use a large number of qubits.

Our task is now to find the integer state $|j\rangle_n$ corresponding to $\theta$.

## 9.7.

### 9.7.1. The algorithm

The QPE algorithm was developed by Kitaev in 1995 [5].

We use $n+1$ qubits. The first $n$ qubits are used to from the computational basis $\ell\rangle_n\, \ell = 0, \cdots, n-1$ and $theremainingqubitstores$|psi\rangle$. The first step of our strategy is to create the following state:

$$|\Phi\rangle_{n+1} = \frac{1}{\sqrt{2^n}}\left(|0\rangle_n + |1\rangle_n + \cdots |n-1\rangle_n\right) \otimes |\psi\rangle = \sum_{\ell=0}^{n-1} |\ell\rangle_n \otimes |\psi\rangle \tag{9.16}$$

We try to find a quantum algorithm that transforms it to $|j\rangle_n \otimes |\psi\rangle$. In other words, the algorithm eliminates the integer states expect for $|j\rangle_n$. Since there is one term at the end, measurement detects $|j\rangle_n$ with high certainty.

We can generate (9.16) using the Walsh-Hadamard transformation discussed in Section 9.2. Furthermore, we know that Eq. (9.16) is equivalent to

$$|\Phi\rangle_{n+1} = \frac{1}{\sqrt{2^{n+1}}}\left(|0\rangle + |1\rangle\right) \otimes \cdots \otimes \left(|0\rangle + |1\rangle\right) \otimes |\psi\rangle$$

Next we consider a controlled gate $CU^{2^i}$ which applys the unitary operator $U\ 2^i$ times on $\psi\rangle$ when $i$-th qubit is in $1\rangle$ and does nothing otherwise. For example, the $i$-th qubit is transformed as (only the $i$-qubit and $|psi\rangle$ are shown)

$$\begin{aligned} CU^{2^i}\left(|0\rangle + |1\rangle\right)_i \otimes |\psi\rangle &= |0\rangle \otimes |\psi\rangle + |1\rangle \otimes \left(U^{2^i}|\psi\rangle\right) = |0\rangle \otimes |\psi\rangle + |1\rangle \otimes \left(e^{2\pi i\,\theta\,2^i}|\psi\rangle\right) \\ &= \left(|0\rangle + e^{2\pi i\,\theta\,2^i}|1\rangle\right) \otimes |\psi\rangle \end{aligned}$$

Applying the gate to all qubits (except for $|\psi\rangle$),

$$\begin{aligned} \bigotimes_{i=0}^{n-1} CU^{2^i}|\Psi\rangle_{n+1} &= \frac{1}{\sqrt{2^{n+1}}}\left(|0\rangle + e^{2\pi i\,\theta\,2^{n-1}}|1\rangle\right) \otimes \cdots \otimes \left(|0\rangle + e^{2\pi i\,\theta\,2^0}|1\rangle\right) \otimes |\psi\rangle \\ &= \frac{1}{\sqrt{2^{n+1}}}\sum_{\ell=0}^{n-1} e^{2\pi i\,\theta\,\ell}|\ell\rangle \otimes |\psi\rangle = \frac{1}{\sqrt{2^{n+1}}}\sum_{\ell=0}^{n-1} e^{2\pi i\,2^n\theta\,\ell/2^n}|\ell\rangle \otimes |\psi\rangle \\ &= \mathcal{F}_n|2^n\theta\rangle \otimes |\psi\rangle \end{aligned}$$

Recall that $2^n\theta$ is an integer as discussed in Section 9.6.2. Hence, we have Fourier transform of a computational basis.

The final step is to apply inverse Fourier transform.

$$\mathcal{F}_n^\dagger \cdot \bigotimes_{i=0}^{n-1} CU^{2^i}|\Psi\rangle_{n+1} = \mathcal{F}_n^\dagger\mathcal{F}_n|2^n\theta\rangle \otimes |\psi\rangle = |2^n\theta\rangle \otimes |\psi\rangle.$$

We have obtained a desired state. Suppose that we obtain $|j\rangle$ up on the measurement with high probability, then $\theta = j/2^n$.

If the number of qubits is not big enough, $2^n\theta$ can deviate from any integer. Nevertheless, the chance to find the nearest integer $j$ is much high than other integers. Thence, this algorithm gives us a good estimate of $\theta$. See the discussion of the error in [6].

### 9.7.2. Example

Consider a unitary operator $U = T^3$. $|1\rangle$ is known to be its eigenket. We want to find the phase of angle $\theta$ of the corresponding eigenvalue. Since the value of $\theta$ is not known, we don't know how many qubits are needed. Let us try three qubits for the expression of $\theta$ and another qubit for $|psi\rangle$. We need also three classical bits for measurement. $T^3$ can be realized by the pase gate $P(3\pi/4)$.

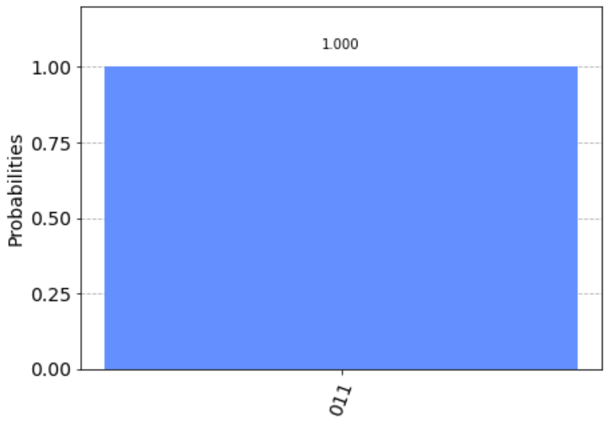
```python
import numpy as np
from qiskit import *

cr=ClassicalRegister(3,'c')
qr=QuantumRegister(4,'q')
qc=QuantumCircuit(qr,cr)

# Walsh-Hadamard transformation for qubit 0-2
qc.h(range(3))

# X for qubit 3 to create |1>
qc.x(3)

qc.barrier()

# qubit 0
qc.cp(np.pi*3/4,0,3)

# qubit 1
qc.cp(np.pi*3/4,1,3)
qc.cp(np.pi*3/4,1,3)

# qubit 2
qc.cp(np.pi*3/4,2,3)
qc.cp(np.pi*3/4,2,3)
qc.cp(np.pi*3/4,2,3)
qc.cp(np.pi*3/4,2,3)

qc.barrier()

qc.draw()
```

```
     ┌───┐ ░                                                                    »
q_0: ┤ H ├─░──■─────────────────────────────────────────────────────────────────»
     ├───┤ ░  │                                                                 »
q_1: ┤ H ├─░──┼─────────■──────────■────────────────────────────────────────────»
     ├───┤ ░  │         │          │                                            »
q_2: ┤ H ├─░──┼─────────┼──────────┼─────────■──────────■──────────■────────────»
     ├───┤ ░  │P(3π/4)  │P(3π/4)   │P(3π/4)  │P(3π/4)   │P(3π/4)   │P(3π/4)     »
q_3: ┤ X ├─░──■─────────■──────────■─────────■──────────■──────────■────────────»
     └───┘ ░                                                                    »
c: 3/═══════════════════════════════════════════════════════════════════════════»
                                                                                »
«                  ░ 
«q_0: ─────────────░─
«                  ░ 
«q_1: ─────────────░─
«                  ░ 
«q_2: ─■───────────░─
«      │P(3π/4)    ░ 
«q_3: ─■───────────░─
«                  ░ 
«c: 3/═══════════════
«                    
```

```python
# Inverse Fourier transform

# SWAP
qc.swap(0,2)

# qubit 0
qc.h(0)

# qubit 1
qc.cp(-np.pi/2,1,0)
qc.h(1)

# qubit 2
qc.cp(-np.pi/4,2,0)
qc.cp(-np.pi/2,2,1)
qc.h(2)

qc.barrier()

qc.measure(0,0)
qc.measure(1,1)
qc.measure(2,2)

qc.draw()
```

```
     ┌───┐ ░                                                                 »
q_0: ┤ H ├─░──■──────────────────────────────────────────────────────────────»
     ├───┤ ░  │                                                              »
q_1: ┤ H ├─░──┼─────────■──────────■─────────────────────────────────────────»
     ├───┤ ░  │         │          │                                         »
q_2: ┤ H ├─░──┼─────────┼──────────┼─────────■──────────■──────────■─────────»
     ├───┤ ░  │P(3π/4)  │P(3π/4)   │P(3π/4)  │P(3π/4)   │P(3π/4)   │P(3π/4)  »
q_3: ┤ X ├─░──■─────────■──────────■─────────■──────────■──────────■─────────»
     └───┘ ░                                                                 »
c: 3/════════════════════════════════════════════════════════════════════════»
                                                                             »
«                 ░    ┌───┐                                           ░ ┌─┐      
«q_0: ────────────░──X─┤ H ├─■───────────────■─────────────────────────░─┤M├──────
«                 ░  │ └───┘ │P(-π/2) ┌───┐  │                         ░ └╥┘┌─┐   
«q_1: ────────────░──┼───────■────────┤ H ├──┼──────────■──────────────░──╫─┤M├───
«                 ░  │                └───┘  │P(-π/4)   │P(-π/2) ┌───┐ ░  ║ └╥┘┌─┐
«q_2: ─■──────────░──X───────────────────────■──────────■────────┤ H ├─░──╫──╫─┤M├
«      │P(3π/4)   ░                                              └───┘ ░  ║  ║ └╥┘
«q_3: ─■──────────░────────────────────────────────────────────────────░──╫──╫──╫─
«                 ░                                                    ░  ║  ║  ║ 
«c: 3/════════════════════════════════════════════════════════════════════╩══╩══╩═
«                                                                         0  1  2 
```

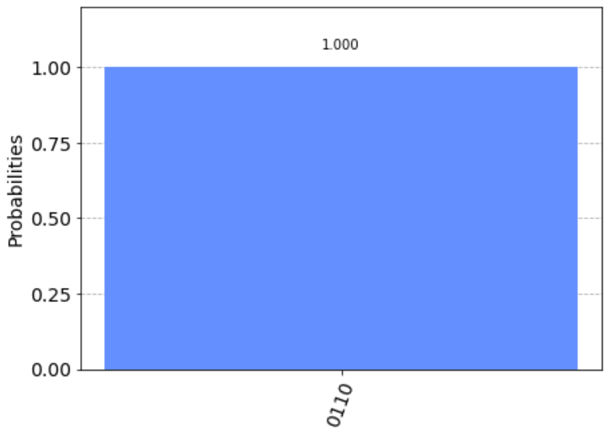
```python
# Chose a general quantum simulator without noise.
# The simulator behaves as an ideal quantum computer.
backend = Aer.get_backend('qasm_simulator')

# set number of tries
nshots=8192

# execute the quantum circuit and store the outcome
job = backend.run(qc,shots=nshots)

# extract the result
result = job.result()

# count the outcome
counts = result.get_counts()

from qiskit.visualization import plot_histogram
plot_histogram(counts)
```



We obtain $|011\rangle = |3\rangle_3$ with unit probability. The quantum algorithm has selected $j = 3$ out of 8 integers. Hence, $\theta = 3/2^3 = 3/8$. The result is in perfect agreement with the actual eigenvalue $e^{i\pi 3/4} = e^{2\pi i 3/8}$.

What will happen if we use more qubits than necessary? Let us try 4 qubits. There are 16 integer numbers to chose. If $j = 6$ is selected, we should get the same answer. The following code indeed finds 6 as the answer.

```python
import numpy as np
from qiskit import *

cr=ClassicalRegister(4,'c')
qr=QuantumRegister(5,'q')
qc=QuantumCircuit(qr,cr)

# Walsh-Hadamard transformation for qubit 0-2
qc.h(range(4))

# X for qubit 3 to create |1>
qc.x(4)

qc.barrier()

for i in range(4):

# qubit 0
    for _ in range(2**i):
        qc.cp(np.pi*3/4,i,4)

qc.barrier()

# SWAP
qc.swap(0,3)
qc.swap(1,2)

# qubit 0
qc.h(0)

# qubit 1
qc.cp(-np.pi/2,1,0)
qc.h(1)

# qubit 2
qc.cp(-np.pi/4,2,0)
qc.cp(-np.pi/2,2,1)
qc.h(2)

# qubit 3
qc.cp(-np.pi/8,3,0)
qc.cp(-np.pi/4,3,1)
qc.cp(-np.pi/2,3,2)
qc.h(3)

qc.barrier()

qc.measure(0,0)
qc.measure(1,1)
qc.measure(2,2)
qc.measure(3,3)

# Chose a general quantum simulator without noise.
# The simulator behaves as an ideal quantum computer.
backend = Aer.get_backend('qasm_simulator')

# set number of tries
nshots=8192

# execute the quantum circuit and store the outcome
job = backend.run(qc,shots=nshots)

# extract the result
result = job.result()

# count the outcome
counts = result.get_counts()

from qiskit.visualization import plot_histogram
plot_histogram(counts)
```

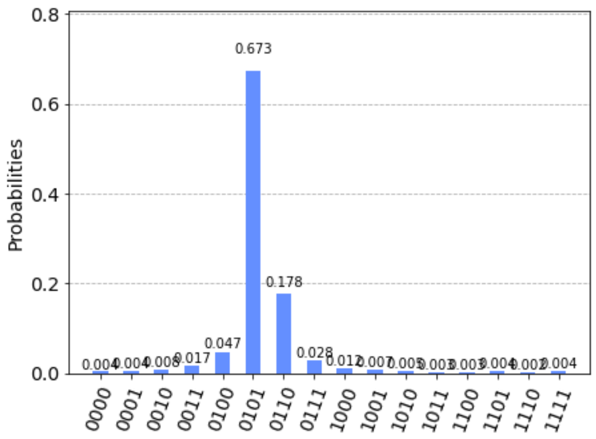


Now we got $|0110\rangle = |6\rangle$ from which we find $\theta = 6/2^4 = 3/8$. The result does not change.

### 9.7.3. Irrational value

If the eigenvalue is $e^{2\pi i/3}$, $theta = 1/3$ cannot be expressed by the finite binary fractions and thus it is not possible to get the perfect answer using the current algorithm. Let us consider $U = P(2\pi/3)$ and $\psi\rangle = |1\rangle$.

```python
import numpy as np
from qiskit import *

cr=ClassicalRegister(4,'c')
qr=QuantumRegister(5,'q')
qc=QuantumCircuit(qr,cr)

# Walsh-Hadamard transformation for qubit 0-2
qc.h(range(4))

# X for qubit 3 to create |1>
qc.x(4)

qc.barrier()

for i in range(4):

# qubit 0
    for _ in range(2**i):
        qc.cp(np.pi*2/3,i,4)

qc.barrier()

# SWAP
qc.swap(0,3)
qc.swap(1,2)

# qubit 0
qc.h(0)

# qubit 1
qc.cp(-np.pi/2,1,0)
qc.h(1)

# qubit 2
qc.cp(-np.pi/4,2,0)
qc.cp(-np.pi/2,2,1)
qc.h(2)

# qubit 3
qc.cp(-np.pi/8,3,0)
qc.cp(-np.pi/4,3,1)
qc.cp(-np.pi/2,3,2)
qc.h(3)

qc.barrier()

qc.measure(0,0)
qc.measure(1,1)
qc.measure(2,2)
qc.measure(3,3)

# Chose a general quantum simulator without noise.
# The simulator behaves as an ideal quantum computer.
backend = Aer.get_backend('qasm_simulator')

# set number of tries
nshots=8192

# execute the quantum circuit and store the outcome
job = backend.run(qc,shots=nshots)

# extract the result
result = job.result()

# count the outcome
counts = result.get_counts()

from qiskit.visualization import plot_histogram
plot_histogram(counts)
```



Clearly $|0101\rangle = |5\rangle$ is dominant. It corresponds to $\theta = 5/2^4 = 0.3125$. The exact answer is $1/3 = 0.3333\ldots$. The agreement is not bad and the error can be reduced by using more qubits.

Last modified on 07/24/2022.

## References

1 Jürgen Audretsch. *Entangled Systems - New Directions in Quantum Physics*. Wiley-VHC, 2007. ISBN 978-3-527-40684-5.

2 Nicolas Gisin. *Quantum Chance*. Springer International Publishing, 2014. doi:10.1007/978-3-319-05473-5.

3 N. David Mermin. What's wrong with this pillow? *Physics Today*, 42(4):9–11, April 1989. URL: https://doi.org/10.1063/1.2810963, doi:10.1063/1.2810963.

4 David Deutsch and Roger Penrose. Quantum theory, the church-turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 400(1818):97–117, July 1985. URL: https://doi.org/10.1098/rspa.1985.0070, doi:10.1098/rspa.1985.0070.

5 A. Yu. Kitaev. Quantum measurements and the abelian stabilizer problem. *arXiv*, pages quant–ph/9511026, 1995. doi:10.48550/ARXIV.QUANT-PH/9511026.

6 M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000. ISBN 0-521-63503-9.