

---

# pyQPanda

*Release 1.0.0*

**BYLZ**

Oct 19, 2021



# CONTENTS

<b>1</b>	<b>1 Basic Introduction</b>	<b>1</b>
1.1	1.1 System configuration . . . . .	1
1.2	1.2 Download pyqpanda . . . . .	1
<b>2</b>	<b>2 Advanced Quantum programming</b>	<b>3</b>
2.1	2.1 Quantum logic gate . . . . .	3
2.1.1	2.1.1 Matrix form of common quantum logic gates . . . . .	5
2.1.1.1	2.1.1.1 Single-qubit quantum logic gate . . . . .	5
2.1.1.2	2.1.1.2 Multi-qubit quantum logic gates . . . . .	7
2.1.2	2.1.2 Interface introduction . . . . .	9
2.1.3	2.1.3 Example . . . . .	10
2.2	2.2 Quantum circuit . . . . .	11
2.2.1	2.2.1 Quantum algorithm circuit diagram . . . . .	11
2.2.2	2.2.2 Interface introduction . . . . .	12
2.2.3	2.2.3 Example . . . . .	12
2.3	2.3 QWhile . . . . .	13
2.3.1	2.3.1 Interface introduction . . . . .	13
2.3.2	2.3.2 Example . . . . .	14
2.4	2.4 QIf . . . . .	15
2.4.1	2.4.1 Interface introduction . . . . .	15
2.4.2	2.4.2 Example . . . . .	15
2.5	2.5 Quantum program . . . . .	16
2.5.1	2.5.1 Interface introduction . . . . .	16
2.5.2	2.5.2 Example . . . . .	17
2.6	2.6 Quantum simulator . . . . .	18
2.6.1	2.6.1 Full-amplitude quantum simulator . . . . .	18
2.6.1.1	2.6.1.1 Interface introduction . . . . .	18
2.6.1.2	2.6.1.2 Example 1 . . . . .	20
2.6.1.3	2.6.1.3 Example 2 . . . . .	21
2.6.2	2.6.2 Noise inclusive quantum simulator . . . . .	22

2.6.2.1	2.6.2.1 Introduction to noise models . . . . .	22
2.6.2.1.1	(1) DAMPING_KRAUS_OPERATOR . . . . .	22
2.6.2.1.2	(2) DEPHASING_KRAUS_OPERATOR . . . . .	22
2.6.2.1.3	(3) DECOHERENCE_KRAUS_OPERATOR . . . . .	23
2.6.2.1.4	(4) DEPOLARIZING_KRAUS_OPERATOR . . . . .	23
2.6.2.1.5	(5) BITFLIP_KRAUS_OPERATOR . . . . .	23
2.6.2.1.6	(6) BIT_PHASE_FLIP_OPERATOR . . . . .	23
2.6.2.1.7	(7) PHASE_DAMPING_OPERATOR . . . . .	24
2.6.2.1.8	(8) DOUBLE-GATE NOISE MODEL . . . . .	24
2.6.2.2	2.6.2.2 Interface introduction . . . . .	24
2.6.2.3	2.6.2.3 Example . . . . .	27
2.6.3	2.6.3 Single-amplitude quantum simulator . . . . .	28
2.6.3.1	2.6.3.1 Instructions . . . . .	28
2.6.4	2.6.4 Partial-amplitude quantum simulator . . . . .	30
2.6.4.1	2.6.4.1 Instructions . . . . .	30
2.6.5	2.6.5 Tensor network quantum simulator . . . . .	32
2.6.5.1	2.6.5.1 Applications . . . . .	33
2.6.5.2	2.6.5.2 Instructions . . . . .	34
2.6.5.3	2.6.5.3 Complete example code . . . . .	35
2.7	2.7 Qubit pool . . . . .	36
2.7.1	2.7.1 Introduction . . . . .	36
2.7.2	2.7.2 Interface description . . . . .	36
2.7.3	2.7.3 Example . . . . .	37
2.8	2.8 Quantum measurement . . . . .	39
2.8.1	2.8.1 Interface introduction . . . . .	39
2.8.2	2.8.2 Example . . . . .	40
2.9	2.9 Probability measurement . . . . .	41
2.9.1	2.9.1 Interface introduction . . . . .	41
2.9.2	2.9.2 Example . . . . .	42
2.10	2.10 OriginQ Cloud service . . . . .	43
2.10.1	2.10.1 Real chip computing service . . . . .	43
2.10.1.1	2.10.1.1 Origin Wuyuan superconducting chip . . . . .	43
2.10.2	2.10.2 Origin high-performance computing cluster cloud service . . . . .	49
2.10.2.1	2.10.2.1 Full-amplitude simulation cloud computing . . . . .	50
2.10.2.2	2.10.2.2 Partial-amplitude simulation cloud computing . . . . .	51
2.10.2.3	2.10.2.3 Single-amplitude cloud computing . . . . .	51
2.10.2.4	2.10.2.4 Noise simulation cloud computing . . . . .	51
2.10.2.5	2.10.2.5 Get the quantum state tomography results . . . . .	52
<b>3</b>	<b>3 Quantum program information</b>	<b>55</b>
3.1	3.1 NodeIter . . . . .	55
3.1.1	3.1.1 Interface introduction . . . . .	55

3.1.2	3.1.2 Example . . . . .	56
3.2	3.2 Logic gate statistics . . . . .	57
3.2.1	3.2.1 Introduction . . . . .	57
3.2.2	3.2.2 Interface introduction . . . . .	58
3.2.3	3.2.3 Example . . . . .	58
3.3	3.3 Counting quantum program clock cycle . . . . .	59
3.3.1	3.3.1 Introduction . . . . .	59
3.3.2	3.3.2 Interface introduction . . . . .	59
3.3.3	3.3.3 Example . . . . .	60
3.4	3.4 Get the corresponding matrix of the quantum circuit . . . . .	60
3.4.1	3.4.1 Example . . . . .	61
3.5	3.5 Judge whether quantum logic gate matches with the quantum topology . . . . .	62
3.5.1	3.5.1 Interface introduction . . . . .	62
3.6	3.6 Get adjacent quantum logic gates at the designated position. . . . .	64
3.6.1	3.6.1 Example . . . . .	64
3.7	3.7 Judge whether two quantum logic gates are interchanged for their positions . . . . .	66
3.7.1	3.7.1 Example . . . . .	67
3.8	3.8 Judge whether the logic gates are of a set of quantum logic gates supported by the quantum chip . . . . .	68
3.9	3.9 Character drawing of quantum circuit . . . . .	70
3.9.1	3.9.1 Example . . . . .	70
3.10	3.10 Quantum volume . . . . .	74
3.10.1	3.10.1 Introduction . . . . .	74
3.10.2	3.10.2 Interface description . . . . .	74
3.10.3	3.10.3 Example . . . . .	74
3.11	3.11 Random benchmark . . . . .	75
3.11.1	3.11.1 Introduction . . . . .	75
3.11.2	3.11.2 Interface description . . . . .	75
3.11.3	3.11.3 Example . . . . .	76
3.12	3.12 Cross-entropy benchmark . . . . .	77
3.12.1	3.12.1 Introduction . . . . .	77
3.12.2	3.12.2 Interface description . . . . .	77
3.12.3	3.12.3 Example . . . . .	77
<b>4</b>	<b>4 Compiling of quantum program . . . . .</b>	<b>79</b>
4.1	4.1 Conversion of QASM by a quantum program . . . . .	79
4.1.1	4.1.1 QASM introduction . . . . .	79
4.1.2	4.1.2 Example . . . . .	80
4.2	4.2 Conversion into a quantum program by QASM . . . . .	81
4.2.1	4.2.1 QASM introduction . . . . .	81
4.2.2	4.2.2 Example . . . . .	82
4.3	4.3 Conversion into Quil by a quantum program . . . . .	83
4.3.1	4.3.1 Introduction . . . . .	83

4.3.2	4.3.2 Interface introduction . . . . .	84
4.3.3	4.3.3 Example . . . . .	84
4.4	4.4 Serialization of quantum programs . . . . .	85
4.4.1	4.4.1 Introduction . . . . .	85
4.4.2	4.4.2 Interface introduction . . . . .	85
4.4.3	4.4.3 Example . . . . .	86
4.5	4.5 Parse quantum program binary files . . . . .	87
4.5.1	4.5.1 Introduction . . . . .	87
4.5.2	4.5.2 Interface introduction . . . . .	87
4.5.3	4.5.3 Example . . . . .	87
4.6	4.6 Conversion into a quantum program by OriginIR . . . . .	88
4.6.1	4.6.1 OriginIR . . . . .	88
4.6.2	4.6.2 Example . . . . .	89
4.7	4.7 Conversion of OriginIR by a quantum program . . . . .	91
4.7.1	4.7.1 OriginIR introduction . . . . .	91
4.7.1.1	4.7.1.1 Qubit . . . . .	91
4.7.1.2	4.7.1.2 Classical register . . . . .	91
4.7.1.3	4.7.1.3 Quantum logic gate . . . . .	91
4.7.1.4	4.7.1.4 Transposed conjugate operation . . . . .	93
4.7.1.5	4.7.1.5 Adding control qubit operation . . . . .	93
4.7.1.6	4.7.1.6 QIF . . . . .	93
4.7.1.7	4.7.1.7 QWHILE . . . . .	94
4.7.1.8	4.7.1.8 Classical expression . . . . .	94
4.7.1.9	4.7.1.9 MEASURE operation . . . . .	95
4.7.1.10	4.7.1.10 RESET operation . . . . .	95
4.7.1.11	4.7.1.11 BARRIER operation . . . . .	95
4.7.1.12	4.7.1.12 QGATE operation . . . . .	96
4.7.1.13	4.7.1.13 OriginIR program example . . . . .	97
4.7.2	4.7.2 Example . . . . .	98
4.8	4.8 Quantum program matching topology . . . . .	99
4.8.1	4.8.1 Interface description . . . . .	99
4.8.2	4.8.2 Example . . . . .	100
<b>5</b>	<b>5 Utility tool</b>	<b>103</b>
5.1	5.1 Quantum circuit query and replacement . . . . .	103
5.1.1	5.1.1 Introduction to interface used . . . . .	103
5.1.2	5.1.2 Example . . . . .	103
5.2	5.2 Filling QProg by gate I . . . . .	105
5.2.1	5.2.1 Example . . . . .	105
5.3	5.3 Unitary matrix decomposition . . . . .	106
5.3.1	5.3.1 Objective of algorithm . . . . .	107
5.3.2	5.3.2 Overview of algorithm . . . . .	107

5.3.3	5.3.3 Instructions . . . . .	107
5.3.4	5.3.4 Example . . . . .	107
<b>6</b>	<b>6 Component</b>	<b>111</b>
6.1	6.1 Class of Pauli operators . . . . .	111
6.1.1	6.1.1 Interface introduction . . . . .	112
6.1.2	6.1.2 Example . . . . .	114
6.2	6.2 Class of Fermionic operators . . . . .	115
6.2.1	6.2.1 Example . . . . .	115
6.3	6.3 Optimization algorithm (direct search) . . . . .	116
6.3.1	6.3.1 Interface introduction . . . . .	116
6.3.2	6.3.2 Example . . . . .	117
<b>7</b>	<b>7 VQC</b>	<b>121</b>
7.1	7.1 Variable . . . . .	121
7.1.1	7.1.1 Interface introduction . . . . .	121
7.1.2	7.1.2 Example . . . . .	122
7.2	7.2 Operator . . . . .	123
7.3	7.3 Variational quantum logic gate . . . . .	126
7.3.1	7.3.1 Interface introduction . . . . .	126
7.3.2	7.3.2 Example . . . . .	127
7.4	7.4 Variational quantum logic gate (VQG) . . . . .	128
7.4.1	7.4.1 Interface introduction . . . . .	129
7.4.2	7.4.2 Dynamic modification of parameters . . . . .	132
7.4.2.1	7.4.2.1 Example: . . . . .	132
7.5	7.5 Optimization algorithm (gradient descent) . . . . .	134
7.5.1	7.5.1 Interface introduction . . . . .	134
7.5.2	7.5.2 Example . . . . .	135
7.6	7.6 Comprehensive example . . . . .	138
7.6.1	7.6.1 QAOA . . . . .	138
<b>8</b>	<b>8 Basis of quantum algorithm</b>	<b>145</b>
8.1	8.1 Review of basic concepts . . . . .	145
8.1.1	8.1.1 Basic definitions . . . . .	145
8.1.2	8.1.2 pyQPanda interface function . . . . .	145
8.1.3	8.1.3 Example . . . . .	146
8.2	8.2 Experimental state preparation and quantum entanglement . . . . .	147
8.2.1	8.2.1 Experimental state preparation . . . . .	147
8.2.1.1	8.2.1.1 Maximum superposition state . . . . .	148
8.2.2	8.2.2 Quantum entanglement . . . . .	148
8.2.3	8.2.3 Maximum superposition state preparation . . . . .	149
8.3	8.3 Hadamard Test and SWAP Test . . . . .	150
8.3.1	8.3.1 Hadamard Test . . . . .	150

	8.3.1.1	8.3.1.1 Output results and generalization . . . . .	150
	8.3.1.2	8.3.1.2 Code example . . . . .	151
8.3.2		8.3.2 SWAP Test . . . . .	152
	8.3.2.1	8.3.2.1 Code example . . . . .	152
8.4		8.4 Amplitude magnification . . . . .	153
	8.4.1	8.4.1 Background of algorithm . . . . .	154
	8.4.2	8.4.2 Code example . . . . .	155
8.5		8.5 Quantum Fourier transform . . . . .	156
	8.5.1	8.5.1 Basic definition . . . . .	156
	8.5.2	8.5.2 Construction of quantum circuit . . . . .	156
	8.5.2.1	8.5.2.1 Sum form and tensor product form of QFT . . . . .	156
	8.5.2.2	8.5.2.2 Binary expansion and quantum state preparation . . . . .	157
	8.5.3	8.5.3 Code implementation . . . . .	158
8.6		8.6 Quantum phase estimation . . . . .	159
	8.6.1	8.6.1 Overview of structure of quantum circuit . . . . .	159
	8.6.2	8.6.2 Construction of quantum circuit . . . . .	159
	8.6.2.1	8.6.2.1 Eigen quantum state and eigenvalue phase extraction . . . . .	159
	8.6.2.2	8.6.2.2 Transfer of eigenvalue phase from amplitude to base vector . . . . .	160
	8.6.3	8.6.3 Quantum circuit diagram and code implementation . . . . .	160
8.7		8.7 For operations of quantum . . . . .	162
	8.7.1	8.7.1 Background of adder algorithm . . . . .	162
	8.7.1.1	8.7.1.1 Component of MAJ quantum circuit . . . . .	163
	8.7.1.2	8.7.1.2 Component of UMA quantum circuit . . . . .	164
	8.7.2	8.7.2 For operations of quantum . . . . .	166
	8.7.2.1	8.7.2.1 Quantum adder . . . . .	166
	8.7.2.2	8.7.2.2 Quantum subtracter . . . . .	166
	8.7.2.3	8.7.2.3 Quantum multiplier . . . . .	166
	8.7.2.4	8.7.2.4 Quantum divider . . . . .	166
	8.7.3	8.7.3 Code implementation and use instructions . . . . .	167
	8.7.3.1	8.7.3.1 Quantum adder . . . . .	167
	8.7.3.2	8.7.3.2 Quantum subtracter . . . . .	167
	8.7.3.3	8.7.3.3 Quantum multiplier . . . . .	167
	8.7.3.4	8.7.3.4 Quantum divider . . . . .	168
	8.7.4	8.7.4 Example . . . . .	169
8.8		8.8 HHL algorithm . . . . .	170
	8.8.1	8.8.1 Overview of background . . . . .	170
	8.8.2	8.8.2 Principle of algorithm . . . . .	170
	8.8.2.1	8.8.2.1 Extraction of eigenvalue through QPE . . . . .	171
	8.8.2.2	8.8.2.2 Transfer of eigenvalue through controlled rotation . . . . .	172
	8.8.2.3	8.8.2.3 Output of resulting quantum state through inverse QPE . . . . .	172
	8.8.3	8.8.3 Quantum circuit diagram and reference code . . . . .	172
8.9		8.9 Grover algorithm and Quantum Counting algorithm . . . . .	174



8.9.1	8.9.1 Overview of background . . . . .	174
8.9.1.1	8.9.1.1 Quantum Counting . . . . .	174
8.9.1.2	8.9.1.2 Search for solution elements . . . . .	175
8.9.2	8.9.2 Principle of algorithm . . . . .	175
8.9.2.1	8.9.2.1 QPE process based on amplitude amplification operator . . . . .	175
8.10	8.10 Shor' s Algorithm . . . . .	179
8.10.1	8.10.1 Background of problem . . . . .	179
8.10.2	8.10.2 Principle of algorithm . . . . .	179
8.10.2.1	8.10.2.1 Pre-lemma . . . . .	180
8.10.2.2	8.10.2.2 Construction of modular multiplication quantum gate . . . . .	180
8.10.2.3	8.10.2.3 Solving of modular exponentiation inverse element . . . . .	181
8.10.3	8.10.3 Quantum circuit diagram and reference code . . . . .	181
8.11	8.11 Quantum imaginary time evolution . . . . .	182
8.11.1	8.11.1 Overview of background . . . . .	183
8.11.2	8.11.2 Principle of algorithm . . . . .	183
8.11.2.1	8.11.2.1 Approximate solution from Schrodinger Equation to differential equation . . . . .	183
8.11.2.2	8.11.2.2 Imaginary time evolution approaching ground state . . . . .	184
8.11.3	8.11.3 Quantum circuit diagram and reference code . . . . .	185



## 1 BASIC INTRODUCTION

For the sake of compatibility efficiency and convenience, QPanda2 is provided with two versions: C++ and Python. This document mainly introduces how to use the Python version. To learn about the use of C++ version, please go to QPanda2.

We use the pybind11 tools to encapsulate the functions and classes in QPanda2 in a direct and concise way, and provide the almost perfect mapping function. The code of the encapsulated part will generate a dynamic library during Qpanda2 compilation, so the encapsulated part can be imported as a Python package.

### 1.1 1.1 System configuration

Pyqpanda takes C++ as the host language, with requirements for the system environment as follows:

software	version
GCC	$\geq 5.4.0$
Python	$\geq 3.6.0$

### 1.2 1.2 Download pyqpanda

If you have installed the python environment and pip tool, please enter the following commands at the terminal or console:

```
pip install pyqpanda
```

---

**Note**

In case of permission problems under linux, add `sudo` .

---



## **2 ADVANCED QUANTUM PROGRAMMING**

### **2.1 2.1 Quantum logic gate**









In classical computing, bit is the most basic unit and logic gate is the most basic control mode. We can control the circuit through combination of logic gates. Similarly, qubit is processed by the quantum logic gate. We consciously evolve quantum states by using quantum logic gates. Therefore, the quantum logic gate is the basis of the quantum algorithm.

The quantum logic gate is denoted with unitary matrix. The most common quantum logic gates operate on one or two qubits, just as the common classical logic gates operate on one or two bits.



2.1.1 2.1.1 Matrix form of common quantum logic gates








2.1.1.1 2.1.1.1 Single-qubit quantum logic gate

	I	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
	Hadamard	$\begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix}$
	T	$\begin{bmatrix} 1 & 0 \\ 0 & \exp(i\pi/4) \end{bmatrix}$
	S	$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
	Pauli-X	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
	Pauli-Y	$\begin{bmatrix} 0 & -1i \\ 1i & 0 \end{bmatrix}$
	Pauli-Z	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
2.1. 2.1 Quantum logic gate		5
	X1	$\begin{bmatrix} 1/\sqrt{2} & -1i/\sqrt{2} \\ -1i/\sqrt{2} & 1/\sqrt{2} \end{bmatrix}$





## 2.1.1.2 2.1.1.2 Multi-qubit quantum logic gates

	CNOT	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
	CR	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \exp(i\theta) \end{bmatrix}$
	iSWAP	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -i \times \sin(\theta) & 0 \\ 0 & -i \times \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
	SWAP	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
	CZ	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
	CU	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & u0 & u1 \\ 0 & 0 & u2 & u3 \end{bmatrix}$
	Toffoli	$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$
2.1. 2.1 Quantum logic gate		$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$

PyQPanda encapsulates all quantum logic gates as API for the use by users, and can obtain the return value of QGate type. For example, if using Hadamard gate, you can obtain it by the following ways:

```
from pyqpanda import *
import numpy as np
init(QMachineType.CPU)
qubits = qAlloc_many(4)
h = H(qubits[0])
```

The parameter is the target qubit and the return value is the quantum logic gate.

Single-gate without any angle, supported by pyqpanda includes: I, H, T, S, X, Y, Z, X1, Y1, and Z1 .

The application for qubit will be introduced in the part of Quantum simulator.

Single-gate includes logic gate with a rotation angle, and RX gate is taken as an example:

```
rx = RX(qubits[0], np.pi/3)
```

The first parameter is target qubit and the second parameter is rotation angle.

Single-gate supported by pyqpanda includes logic gate with a rotation angle: RX, RY, RZ, U1 and P .

U2, U3 and U4 gates are supported by pyqpanda and used as follows:

```
# U2(qubit, phi, lambda) There are two angles
u2 = U2(qubits[0], np.pi, np.pi/2)

# U3(qubit, theta, phi, lambda) There are three angles
u3 = U3(qubits[0], np.pi, np.pi/2, np.pi/4)

# U4(qubit, alpha, beta, gamma, delta) There are four angles
u4 = U4(qubits[0], np.pi, np.pi/2, np.pi/4, np.pi/2)
```

Except that input parameters are different, two-qubits quantum logic gate and single-qubit quantum logic gate are used by the same measure, and CNOT gate is taken as an example:

```
cnot = CNOT(qubits[0], qubits[1])
```

The first parameter is control qubit and the second parameter is target qubit.

---

### Note

two qubits are different.

---

The double logic gate not provided with angle and supported by pyqpanda includes CNOT, CZ, SWAP, iSWAP, and SqiSWAP.

Except that input parameters are different, two-qubits quantum logic gate and single-qubit quantum logic gate are used by the same measure, and CNOT gate is taken as an example:

```
cnot = CNOT(control_qubit, target_qubit)
```

CNOT gate receives two parameters, in which the first parameter is control qubit and the second parameter is target qubit.

Double logic gate supported by pyqpanda and provided with rotation angle includes CR, CU and CP.

Double-gate with rotation angle, for example, CR gate:

```
cr = CR(qubits[0], qubits[1], np.pi)
```

The first parameter is control qubit, the second parameter is target qubit, and the third parameter is the rotation angle.

The CU gate is supported and used by the following measure:

```
# CU(control, target, alpha, beta, gamma, delta) There are four angles
cu = CU(qubits[0], qubits[1], np.pi, np.pi/2, np.pi/3, np.pi/4)
```

Three-qubit quantum logic gate Toffoli is obtained by the measure:

```
toffoli = Toffoli(qubits[0], qubits[1], qubits[2])
```

In practice, three-qubit quantum logic gate Toffoli is CCNOT gate, and the first two parameters are control qubits and the final parameter is target qubit.

Pyqpanda also supports to add the qubit array into the quantum logic gate, that is, all qubits in the array are computed by endowing the same logic gate, and single-gate H is taken as an example:

```
# Returns a quantum circuit
circuit = H(Qvec);
```

Here, Qvec is the array to store qubits. When multiple gates are added with arrays, multiple corresponding arrays are correspondingly imported and the logic gates are computed according to the subscript sequence of the arrays.

## 2.1.2 2.1.2 Interface introduction

As mentioned at the beginning of this chapter, all quantum logic gates are of the unitary matrix, so you can also perform transposed conjugate on quantum logic gates and obtain a quantum logic gate following the quantum logic gate dagger by using the following measure:

```
rx_dagger = RX(qubits[0], np.pi).dagger()
```

Or:

```
rx_dagger = RX(qubits[0], np.pi)
rx_dagger.set_dagger(true)
```

The quantum logic gate can be added with control qubit, where the quantum logic gate controlled by a quantum logic gate may be added by the following measures:

```
qvec = [qubits[0], qubits[1]]
rx_control = RX(qubits[2], np.pi).control(qvec)
```

Or:

```
qvec = [qubits[0], qubits[1]]
rx_control = RX(qubits[2], np.pi)
rx_control.set_control(qvec)
```

pyqpanda also encapsulates some convenient interfaces to simply the operations of some quantum logic gates.

```
cir = apply_QGate(qubits, H)
```

All qubits are added with H gate

### 2.1.3 2.1.3 Example

The following example mainly demonstrates how to use QGate interface

```
from pyqpanda import *

if __name__ == "__main__":
    init(QMachineType.CPU)
    qubits = qAlloc_many(3)
    control_qubits = [qubits[0], qubits[1]]
    prog = create_empty_qprog()

    # Building quantum programs
    prog << H(qubits) \
        << H(qubits[0]).dagger() \
        << X(qubits[2]).control(control_qubits)

    # Perform probability measurements on quantum programs
    result = prob_run_dict(prog, qubits, -1)

    # Print measurement results
    print(result)
    finalize()
```

The computing results are as follows:

```
{'000': 0.249999999999999295, '001': 0.0, '010': 0.249999999999999295, '011': 0.0, '100': 0.249999999999999295, '101': 0.0, '110': 0.249999999999999295, '111': 0.0}
```

## 2.2 Quantum circuit

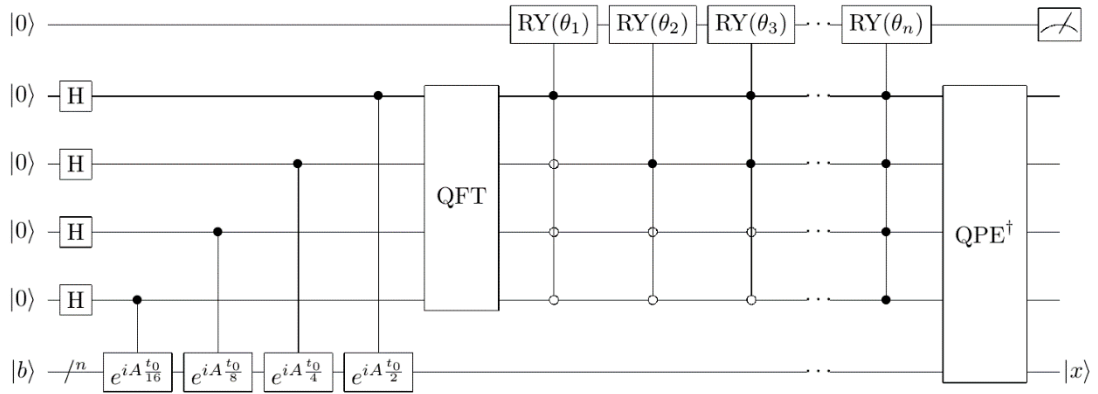
Quantum circuit, also called as quantum logic circuit, is the most common quantum computing model, which represents a circuit to operate qubits in an abstract concept. The quantum circuit is composed of qubits, circuits (timelines), and logic gates. Quantum measurement results often require to be read out.

Distinguished from a traditional circuit that is connected by metal wires to transmit voltage or current signals, the quantum circuit is connected by timeline, that is, the natural evolution in qubit state occurs along with time, and follows the instructions of Hamiltonian operators until they are operated by logic gates.

Each quantum logic gate to constitute a quantum circuit is a `unitary` operator, and therefore the entire quantum circuit is also a large unitary operator.

### 2.2.1 Quantum algorithm circuit diagram

In the current theoretical study of quantum computing, the quantum algorithms are commonly indicated by quantum circuits, such as the quantum circuit diagram of HHL algorithm listed below.



## 2.2.2 Interface introduction

In pyQPanda, QCircuit is a container type and only carries quantum logic gates. Besides, it is also a type of QNode. One QCircuit object can be initialized by the following two measures:

```
cir = QCircuit()
```

Or:

```
cir = create_empty_circuit()
```

You can populate nodes into the end of QCircuit with the following ways, in which pyqpanda reloads << operators as a way to insert the quantum circuit.

```
cir << node
```

Node may be either QGate or QCircuit.

We can also obtain the quantum circuit after the transposed conjugate of QCircuit by the following way:

```
cir_dagger = cir.dagger()
```

If you want to replicate the current quantum circuit and add control qubits to the replicated quantum circuit, the following ways can be adopted:

```
qvec = [qubits[0], qubits[1]]
cir_control = cir.control(qvec)
```

---

### Note

- No error is reported when QPorg, QIf and Measure are inserted into QCircuit, but unexpected errors may occur during the operation.
  - One constituted QCircuit cannot directly conduct quantum computing and simulation, which requires to be further constituted into QProg.
- 

## 2.2.3 Example

```
from pyqpanda import *

if __name__ == "__main__":

    init(QMachineType.CPU)
```

(continues on next page)

(continued from previous page)

```

qubits = qAlloc_many(4)
cbits = cAlloc_many(4)

# Building quantum programs
prog = QProg()
circuit = create_empty_circuit()

circuit << H(qubits[0]) \
    << CNOT(qubits[0], qubits[1]) \
    << CNOT(qubits[1], qubits[2]) \
    << CNOT(qubits[2], qubits[3])

prog << circuit << Measure(qubits[0], cbits[0])

# The quantum program runs 1000 times and returns the measurement
result = run_with_configuration(prog, cbits, 1000)

# The number of times the printed quantum state appears in the result
# of multiple runs of the quantum program
print(result)
finalize()

```

Running results:

```
{'0000': 486, '0001': 514}
```

## 2.3 2.3 QWhile

The operations are controlled circularly by the quantum program; the input parameters are considered as the conditional judgment expression, with the function to execute the while loop operation.

### 2.3.1 2.3.1 Interface introduction

In pyQPanda, QWhileProg is used to execute the while loop operation of the quantum program, and also regarded as a type of QNode; one QWhileProg object can be initialized by the following two measures:

```
qwile = QWhileProg(ClassicalCondition, QNode)
```

Or

```
qwile = create_while_prog(ClassicalCondition, QNode)
```

The abovementioned functions need to have two types of parameters, namely quantum expression of ClassicalCondition and QNode, where passable QNode includes QProg, QCircuit, QGate, QWhileProg, QIfProg, and QMeasure.

### 2.3.2 2.3.2 Example

```
from pyqpanda import *

if __name__ == "__main__":

    init(QMachineType.CPU)
    qubits = qAlloc_many(3)
    cbits = cAlloc_many(3)
    cbits[0].set_val(0)
    cbits[1].set_val(1)

    prog = QProg()
    prog_while = QProg()

    # Build the loop branch of QWhile
    prog_while << H(qubits[0]) << H(qubits[1]) << H(qubits[2]) \
        << assign(cbits[0], cbits[0] + 1) \
<< Measure(qubits[1], cbits[1])

    # Build a QWhile
    qwhile = create_while_prog(cbits[1], prog_while)

    # QWhile is inserted into the quantum program
    prog << qwhile

    # Run and print the measurement results
    result = directly_run(prog)
    print(result)
    print(cbits[0].get_val())
    finalize()
```

Running results:

```
2
{'c1': False}
```



## 2.4 2.4 QIf

QIf refers to the conditional judgment of quantum program; the input parameters are considered as the conditional judgment expression, with the function to execute the conditional judgment.

### 2.4.1 2.4.1 Interface introduction

In pyQPanda, QIfProg is used to execute the conditional judgment of quantum program, and also regarded as a type of QNode; one QIfProg object can be initialized by the following two measures:

```
qif = QIfProg(ClassicalCondition, QNode)
qif = QIfProg(ClassicalCondition, QNode, QNode)
```

Or

```
qif = create_if_prog(ClassicalCondition, QNode)
qif = create_if_prog(ClassicalCondition, QNode, QNode)
```

The mentioned function needs to have two types of parameters, namely quantum expression of ClassicalCondition and QNode. When one QNode parameter passes, QNode is correct branch node. When two QNode parameters pass, the first parameter is a correct branch node and the second parameter is a wrong branch node. Passable QNode includes QProg, QCircuit, QGate, QWhileProg, QIfProg and QMeasure.

### 2.4.2 2.4.2 Example

```
from pyqpanda import *

if __name__ == "__main__":

    init(QMachineType.CPU)
    qubits = qAlloc_many(3)
    cbits = cAlloc_many(3)
    cbits[0].set_val(0)
    cbits[1].set_val(3)

    prog = QProg()
    branch_true = QProg()
    branch_false = QProg()

    # Build the right and wrong branches of QIf
    branch_true << H(qubits[0]) << H(qubits[1]) << H(qubits[2])
    branch_false << H(qubits[0]) << CNOT(qubits[0], qubits[1])\
```

(continues on next page)

(continued from previous page)

```

<< CNOT(qubits[1], qubits[2])

    # Build QIf
    qif = create_if_prog(cbits[0] > cbits[1], branch_true, branch_false)

    # QIf is inserted into the quantum program
    prog << qif

# Probability measurement, and returns the probability measurement
# result of the target qubit, subscript decimal
    result = prob_run_tuple_list(prog, qubits, -1)

    # Print the probability measurement results
    print(result)
    finalize()

```

Running results:

```

[(0, 0.4999999999999999), (7, 0.4999999999999999), (1, 0.0), (2, 0.0), (3, 0.0), (4, 0.0), (5, 0.0), (6, 0.0)]

```

## 2.5 Quantum program

For the compilation and construction of the quantum program, the quantum program is designed. Generally, it can be understood as an operation sequence. As the quantum algorithm also includes classical computing, so the industrial assumption is that the quantum computer in the immediate future is a hybrid structure and composed of two parts: one is the classical computer for classical computing and control; and the other is the quantum equipment for quantum computing. pyQPanda considers the programming procedure of quantum program as a part of classical program running, and the entire peripheral host program must contain the part of quantum program creation.

### 2.5.1 Interface introduction

In pyQPanda, QProg is a container type of quantum programming and the highest unit of one quantum program. It is also a type of QNode; and one QProg object is initialized by the following measures:

```
prog = QProg()
```

Or

```
prog = create_empty_qprog()
```

Quantum programs can also be constructed through the existing QNode, for example:

```
qubit = qAlloc()
gate = H(qubit)
prog = QProg(gate)
```

The quantum programs of QCircuit, QGate, QWhileProg, QIfProg, ClassicalCondition, and QMeasure may be constructed by the similar measure.

You can populate nodes into the end of QProg with the following ways, in which pyqpanda reloads << operators as a way to insert the quantum circuit.

```
prog << node
```

QNode includes QGate, QPorg, QIf, Measure and the like; and QProg supports insertion of all types of QNode.

## 2.5.2 Example

```
from pyqpanda import *

if __name__ == "__main__":

    init(QMachineType.CPU)
    qubits = qAlloc_many(4)
    cbits = cAlloc_many(4)
    prog = QProg()

    # Building quantum programs
    prog << H(qubits[0]) \
        << X(qubits[1]) \
        << iSWAP(qubits[0], qubits[1]) \
        << CNOT(qubits[1], qubits[2]) \
        << H(qubits[3]) \
        << measure_all(qubits, cbits)

    # The quantum program runs 1000 times and returns the measurement
    result = run_with_configuration(prog, cbits, 1000)

    # The number of times the printed quantum state appears in the result
    # of multiple runs of the quantum program
    print(result)
    finalize()
```

Running results:

```
{'0001': 255, '0111': 253, '1001': 258, '1111': 247}
```

## 2.6 Quantum simulator

Before an actual quantum computer is not constituted, the quantum simulators are used to undertake the verification of quantum algorithms and quantum applications. pyQPanda supports full-amplitude quantum simulator, single-amplitude quantum simulator, partial-amplitude quantum simulator and noise inclusive quantum simulator.

### 2.6.1 Full-amplitude quantum simulator

The full-amplitude quantum simulator can simulate all amplitudes of the quantum state at one time, supports the computing ways such as CPU, single-circuit computing and GPU, and performs configurations during the initialization by the same way, but has different computing efficiency.

#### 2.6.1.1 Interface introduction

Type of full-amplitude quantum simulator:

```
class QMachineType(__pybind11_builtins.pybind11_object):
    """
        Members:

        CPU

        GPU

        CPU_SINGLE_THREAD

        NOISE
    """
```

In pyQPanda, the quantum simulator is constructed by the following ways:

```
init(QMachineType.CPU)
# With init, qvm is not returned and a global qvm is generated in the code
auto qvm = init_quantum_machine(QMachineType.CPU)
# Get the quantum machine object through the interface
qvm = CPUQVM()
# Create a new Quantum Machine object
```

---

#### Note

The functions of `init` and `init_quantum_machine` functions are not thread-safe and not suitable for multi-threaded programming, and the maximum number of qubits and the number of classical registers are both 25 by default.

---

The quantum simulator needs to be initialized after configuration:

```
qvm.init_qvm()
```

---

### Note

No initialization is required when `init` and `init_quantum_machine` interfaces are invoked.

---

Now, we need to apply for qubits and classical registers.

Setting of the maximum number of qubits

```
# Set the maximum number of qubits and the maximum number of classical
# registers
qvm.set_configure(30, 30)
```

---

### Note

If this parameter not set, the maximum number of qubits is 29 by default.

---

For example, we apply for 4 qubits:

```
qubits = qvm.qAlloc_many(4)
```

---

This interface can be used when a qubit is applied:

```
qubit = qvm.qAlloc()
```

---

The classical register is also applied by the interface similar to that to apply for the qubit; and the application method is identical to the qubit application method, for example, the method of applying four classical registers:

```
cbits = qvm.cAlloc_many(4)
```

---

Such interface may be used when one classical register is applied:

```
cbit = qvm.cAlloc()
```

---

In a quantum simulator, qubits or classical registers are applied several times, and therefore we want to know how many qubits or classical registers are applied by the following methods:

```
num_qubit = qvm.get_allocate_qubit_num() # Number of qubits applied
num_cbit = qvm.get_allocate_cmem_num() # Number of classical registers
# applied
```

How should we use simulator to execute quantum programs? The following methods can be adopted:

```
prog = QProg()
prog << H(qubits[0]) << CNOT(qubits[0], qubits[1])\
<< Measure(qubits[0], cbits[0])
result = qvm.directly_run(prog) # Execution of quantum program
```

If you want to run a quantum program and obtain the results of each quantum program, we also provide an interface `run_with_configuration`, in addition to the recursive call of `directly_run`; such interface has two reloading methods as follows:

```
result = qvm.run_with_configuration(prog, cbits, shots)
```

In a method, `prog` is a quantum program; `cbits` is `ClassicalCondition` list; `shots` as reshaped data are the running frequency of the quantum program.

```
result = qvm.run_with_configuration(prog, cbits, config)
```

In another method, `prog` is a quantum program; `cbits` is `ClassicalCondition` list; `config` is dictionary data as follows:

```
config = {'shots': 1000}
```

If you want to obtain the amplitudes of various quantum states after the quantum program runs, `get_qstate` function can be invoked:

```
stat = qvm.get_qstate()
```

The measurement and probability usage of the quantum simulator is the same as those introduced in quantum measurement and probability measurement, which will not be described here.

### 2.6.1.2 2.6.1.2 Example 1

```
from pyqpanda import *

if __name__ == "__main__":
    qvm = CPUQVM()
    qvm.init_qvm()

    qvm.set_configure(29, 29)
    qubits = qvm.qAlloc_many(4)
```

(continues on next page)

(continued from previous page)

```

cbits = qvm.cAlloc_many(4)

# Building a quantum program
prog = QProg()
prog << H(qubits[0]) << CNOT(qubits[0], qubits[1])\
<< Measure(qubits[0], cbits[0])

# The quantum program runs 1000 times and returns the measurement
result = qvm.run_with_configuration(prog, cbits, 1000)

# Print the number of times the quantum state appears in the results of
# multiple runs of the quantum program
print(result)
qvm.finalize()

```

Running results:

```
{'0000': 481, '0001': 519}
```

## Note

The computational results of the quantum program are indefinite, but the corresponding values of 0000 and 0001 should both be approximately 500.

For the ease of use, pyqpanda also encapsulates some process-oriented interfaces, which have basically the same names and using methods as those described above. We modify the above example into a process-oriented interface as follows:

### 2.6.1.3 Example 2

```

from pyqpanda import *

if __name__ == "__main__":
    init(QMachineType.CPU)
    qubits = qAlloc_many(4)
    cbits = cAlloc_many(4)

    # Building a quantum program
    prog = QProg()
    prog << H(qubits[0]) << CNOT(qubits[0], qubits[1])\
    << Measure(qubits[0], cbits[0])

```

(continues on next page)

(continued from previous page)

```
# The quantum program runs 1000 times and returns the measurement
result = run_with_configuration(prog, cbits, 1000)

# Print the number of times the quantum state appears in the results of
# multiple runs of the quantum program

print(result)
finalize()
```

Running results:

```
{'0000': 484, '0001': 516}
```

## 2.6.2 2.6.2 Noise inclusive quantum simulator

Restricted by the physical characteristics of the qubits, the actual quantum computer usually has the inevitable computing error. To better simulate such error in the quantum simulator, pyQPanda provides a noise inclusive quantum simulator on the basis of the quantum simulator. The noise inclusive quantum simulator is closer to the actual quantum computer in terms of simulation. We can customize the types of logic gates supported and the noise model supported by logic gates. Through the custom forms, the quantum programs developed by pyQPanda will be widely applied in practice.

### 2.6.2.1 2.6.2.1 Introduction to noise models

#### 2.6.2.1.1 (1) DAMPING\_KRAUS\_OPERATOR

DAMPING\_KRAUS\_OPERATOR is the relaxation process noise model of qubit. Its kraus operator and representation are as shown below:

$$K_1 = \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1-p} \end{bmatrix}, K_2 = \begin{bmatrix} 0 & \sqrt{p} \\ 0 & 0 \end{bmatrix}$$

A noise parameter is required.

#### 2.6.2.1.2 (2) DEPHASING\_KRAUS\_OPERATOR

DEPHASING\_KRAUS\_OPERATOR is the dephasing process noise model of qubit. Its kraus operator and representation are as shown below:

$$K_1 = \begin{bmatrix} \sqrt{1-p} & 0 \\ 0 & \sqrt{1-p} \end{bmatrix}, K_2 = \begin{bmatrix} \sqrt{p} & 0 \\ 0 & -\sqrt{p} \end{bmatrix}$$

A noise parameter is required.



### 2.6.2.1.3 (3) DECOHERENCE\_KRAUS\_OPERATOR

DECOHERENCE\_KRAUS\_OPERATOR is the decoherence noise model and a combination of the above two noise models, and their relation is as shown below:

$$P_{\text{damping}} = 1 - e^{-\frac{t_{\text{gate}}}{T_1}}, P_{\text{dephasing}} = 0.5 \times \left(1 - e^{-\left(\frac{t_{\text{gate}}}{T_2} - \frac{t_{\text{gate}}}{2T_1}\right)}\right) \quad K_1 = K_1 \text{ damping } K_1 \text{ dephasing } K_2 = K_1 \text{ damping } K_2 \text{ dephasing } K_3 = K_2 \text{ dampin } K_1 \text{ dephasing } K_4 = K_2 \text{ dampin } K_2 \text{ dephasing}$$

Three noise parameters are required.

### 2.6.2.1.4 (4) DEPOLARIZING\_KRAUS\_OPERATOR

DEPOLARIZING\_KRAUS\_OPERATOR depolarization noise model means that single qubit is replaced by a completely mixed state I/2 under specific probability. Its kraus operator and representation are as shown below:

$$K_1 = \sqrt{1 - 3p/4} \times I, K_2 = \sqrt{p}/2 \times X \\ K_3 = \sqrt{p}/2 \times Y, K_4 = \sqrt{p}/2 \times Z$$

I, X, Y and Z respectively indicate that the matrix corresponding to their quantum logic gates

A noise parameter is required.

### 2.6.2.1.5 (5) BITFLIP\_KRAUS\_OPERATOR

BITFLIP\_KRAUS\_OPERATOR is the qubit inversion noise model. Its kraus operator and representation are as shown below:

$$K_1 = \begin{bmatrix} \sqrt{1-p} & 0 \\ 0 & \sqrt{1-p} \end{bmatrix}, K_2 = \begin{bmatrix} 0 & \sqrt{p} \\ \sqrt{p} & 0 \end{bmatrix}$$

A noise parameter is required.

### 2.6.2.1.6 (6) BIT\_PHASE\_FLIP\_OPERATOR

BIT\_PHASE\_FLIP\_OPERATOR is the qubit-phase inversion noise model. Its kraus operator and representation are as shown below:

$$K_1 = \begin{bmatrix} \sqrt{1-p} & 0 \\ 0 & \sqrt{1-p} \end{bmatrix}, K_2 = \begin{bmatrix} 0 & -i \times \sqrt{p} \\ i \times \sqrt{p} & 0 \end{bmatrix}$$

### 2.6.2.1.7 (7) PHASE\_DAMPING\_OPERATOR

PHASE\_DAMPING\_OPERATOR is the phase damping noise model. Its kraus operator and representation are as shown below:

$$K_1 = \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1-p} \end{bmatrix}, K_2 = \begin{bmatrix} 0 & 0 \\ 0 & \sqrt{p} \end{bmatrix}$$

A noise parameter is required.

### 2.6.2.1.8 (8) DOUBLE-GATE NOISE MODEL

Similarly, the double-gate noise model is also divided into the above-mentioned types: DAMPING\_KRAUS\_OPERATOR,

DEPHASING\_KRAUS\_OPERATOR,

DECOHERENCE\_KRAUS\_OPERATOR,

DEPOLARIZING\_KRAUS\_OPERATOR,

BITFLIP\_KRAUS\_OPERATOR,

BIT\_PHASE\_FLIP\_OPERATOR,

and PHASE\_DAMPING\_OPERATOR.

They have the same input parameters as the single-gate noise model; their kraus operator and representation correspond to those of the single-gate noise model: if the single-gate noise model is  $\{K_1, K_2\}$ , the corresponding double-gate noise model is  $\{K_1 \otimes K_1, K_1 \otimes K_2, K_2 \otimes K_1, K_2 \otimes K_2\}$ .

## 2.6.2.2 Interface introduction

Noise model supported currently by pyqpanda

```
class NoiseModel(__pybind11_builtins.pybind11_object):
    """
    Members:

    DAMPING_KRAUS_OPERATOR

    DECOHERENCE_KRAUS_OPERATOR

    DEPHASING_KRAUS_OPERATOR

    PAULI_KRAUS_MAP
```

(continues on next page)

(continued from previous page)

```

DECOHERENCE_KRAUS_OPERATOR_P1_P2

BITFLIP_KRAUS_OPERATOR

DEPOLARIZING_KRAUS_OPERATOR

BIT_PHASE_FLIP_OPERATOR

PHASE_DAMPING_OPERATOR

```

A noise parameter is set by the following method:

```

from pyqpanda import *
import numpy as np

qvm = NoiseQVM()
qvm.init_qvm()
q = qvm.qAlloc_many(4)
c = qvm.cAlloc_many(4)

# If no function qubit is specified, all qubits are valid
qvm.set_noise_model(NoiseModel.BITFLIP_KRAUS_OPERATOR, GateType.PAULI_X_GATE, 0.1)
# When specifying a qubit, only the specified qubit takes effect
qvm.set_noise_model(NoiseModel.BITFLIP_KRAUS_OPERATOR, GateType.RY_GATE, 0.1, \ [q[0], \
→q[1]])
# When you specify a qubit for a double gate, you need to specify two qubits at
# the same time and are sensitive to the sequence of the qubits
qvm.set_noise_model(NoiseModel.DAMPING_KRAUS_OPERATOR, GateType.CNOT_GATE, 0.1, \
→[[q[0], q[1]], [q[1], q[2]]])
# Noise can be added to all types in the circuit
qvm.set_noise_model(NoiseModel.BITFLIP_KRAUS_OPERATOR, types, 0.1)
qvm.set_noise_model(NoiseModel.DECOHERENCE_KRAUS_OPERATOR, types, 0.1, 0.2, 0.3)
qvm.set_noise_model(NoiseModel.DAMPING_KRAUS_OPERATOR, GateType.CNOT_GATE, 0.1, q)

```

The first parameter is the type of the noise model; the second parameter is the type of the quantum logic gate; and the third parameter is required for the noise model.

Three noise parameters are set by the following method:

```

# If no function qubit is specified, all qubits are valid
qvm.set_noise_model(NoiseModel.DECOHERENCE_KRAUS_OPERATOR, GateType.PAULI_Y_GATE, \
5, 2, 0.01)
# When specifying a qubit, only the specified qubit takes effect
qvm.set_noise_model(NoiseModel.DECOHERENCE_KRAUS_OPERATOR, GateType.Y_HALF_PI, \

```

(continues on next page)

(continued from previous page)

```

5, 2, 0.01, [q[0], q[1]])
# When you specify a qubit for a double gate, you need to specify two qubits at the_
→ same
# time and are sensitive to the sequence of the qubits
qvm.set_noise_model(NoiseModel.DECOHERENCE_KRAUS_OPERATOR, GateType.CZ_GATE, \
5, 2, 0.01, [[q[0], q[1]], [q[1], q[0]]])
# Noise can be added to all Gatetype in the circuit
qvm.set_noise_model(NoiseModel.BITFLIP_KRAUS_OPERATOR, types, 0.1)
qvm.set_noise_model(NoiseModel.DECOHERENCE_KRAUS_OPERATOR, types, 0.1, 0.2, 0.3)
qvm.set_noise_model(NoiseModel.DAMPING_KRAUS_OPERATOR, GateType.CNOT_GATE, 0.1, q)

```

The noise inclusive simulator also supports the error of the rotation angle of the quantum logic gate with an angle, and its interface usage is as shown below:

```
qvm.set_rotation_error(0.05)
```

Namely, the error of the rotation angle is set to 0.05.

The measurement error is set by the method similar to the above setting method, except that the type of the quantum logic gate is not designated.

```
qvm.set_measure_error(NoiseModel.DEPOLARIZING_KRAUS_OPERATOR, 0.1)
```

Setting of reset noise:

```

p0 = 0.9
p1 = 0.05
qvm.set_reset_error(p0, p1)

```

$p_0$  is the probability to reset noise to  $|0\rangle$ ;  $p_1$  is the probability to reset noise to  $|1\rangle$ ; and the probability not to reset noise is  $1-p_0-p_1$ .

Setting of read error:

```

f0 = 0.9
f1 = 0.85
qvm.set_readout_error([[f0, 1 - f0], [1 - f1, f1]])

```

When reading  $q_0$ , the probability to read 0 as 0 is 0.9, the probability to read 0 as 1 is  $1 - f_0$ , the probability to read 1 as 1 is 0.85, and the probability to read 0 as 1 is  $1 - f_1$ .

### 2.6.2.3 Example

```

from pyqpanda import *
import numpy as np

if __name__ == "__main__":
    qvm = NoiseQVM()
    qvm.init_qvm()
    q = qvm.qAlloc_many(4)
    c = qvm.cAlloc_many(4)

    qvm.set_noise_model(NoiseModel.BITFLIP_KRAUS_OPERATOR, \ GateType.PAULI_X_GATE, 0.
↪1)
    qv0 = [q[0], q[1]]
    qvm.set_noise_model(NoiseModel.DEPHASING_KRAUS_OPERATOR, \
GateType.HADAMARD_GATE, 0.1, qv0)
    qves = [[q[0], q[1]], [q[1], q[2]]]
    qvm.set_noise_model(NoiseModel.DAMPING_KRAUS_OPERATOR, \ GateType.CNOT_GATE, 0.1, ↪
↪qves)

    f0 = 0.9
    f1 = 0.85
    qvm.set_readout_error([[f0, 1 - f0], [1 - f1, f1]])
    qvm.set_rotation_error(0.05)

    prog = QProg()
    prog << X(q[0]) << H(q[0]) \
        << CNOT(q[0], q[1]) \
        << CNOT(q[1], q[2]) \
        << CNOT(q[2], q[3]) \
        << measure_all(q, c)

    result = qvm.run_with_configuration(prog, c, 1000)
    print(result)

```

Running results:

```

{'0000': 347, '0001': 55, '0010': 50, '0011': 43, '0100': 41, '0101': 18, '0110': 16,
↪ '0111': 34, '1000': 50, '1001': 18, '1010': 18, '1011': 37, '1100': 15, '1101': 49,
↪ '1110': 42, '1111': 167}

```

### 2.6.3 Single-amplitude quantum simulator

At present, we can use the classical computer to simulate quantum simulator based on relevant quantum computing theories. In terms of simulation, the quantum simulator is mainly divided into full amplitude and single amplitude. Their difference mainly lies in that: all amplitudes of the quantum state can be worked out through one-time full-amplitude simulation computing, and one of  $2^n$  amplitudes only can be worked out through one single-amplitude simulation computing.

However, the full-amplitude simulation quantum computing requires relatively long time, and therefore the computing quantity increases along the number of qubits; under the existing hardware, when the number of qubits is beyond 49, no simulation is allowed. When the number of qubits is beyond 49, the simulation can be achieved by the single-amplitude quantum simulator; the simulation speed is greatly improved, and the computing quantity of the algorithm does not increase along with the number of qubits.

#### 2.6.3.1 Instructions

Its usage is very similar to the usage of the quantum simulator module described above. The main interfaces are as follows:

**run:** input parameters include executed quantum programs, applied qubits, maximum RANK, and maximum running time to optimize quickBB

**pmeasure\_bin\_index:** input parameters are binary index strings, and output parameters are the quantum state under the index. Before use, run interface requires to be invoked, such as `pmeasure_bin_index( "0000000000" )`; meanwhile, it is ensured that the string length is the same as the number of qubits measured.

**pmeasure\_dec\_index:** input parameters are decimal index strings, and output parameters are the quantum state under the index. Before use, run interface requires to be invoked, such as `pmeasure_dec_index( "1" )`; meanwhile, it is ensured that the index is not beyond  $2^n$  ( $n$  is the number of qubits).

**get\_prob\_dict:** input parameters include quantum program to be executed and qubits to be measured. Output parameters are all state results of the corresponding qubits. Before use, run interface requires to be invoked. It should be noted that the interface can be used when the number of qubits is within 30.

**prob\_run\_dict:** input parameters include quantum program to be executed and qubits to be measured. Output parameters are all state results of the corresponding qubits. It should be noted that the interface can be used when the number of qubits is within 30.

Firstly, initialize a single-amplitude quantum simulator object through `SingleAmpQVM` in order to manage a series of subsequent behaviors.

```
from pyqpanda import *
from numpy import pi

qvm = SingleAmpQVM()
```

Secondly, initialize, construct and carry the quantum programs:

```

qvm.init_qvm()

qv = qvm.qAlloc_many(10)
cv = qvm.cAlloc_many(10)

prog = QProg()

# Building quantum programs
prog << CZ(qv[1], qv[5])\
    << CZ(qv[3], qv[5])\
    << CZ(qv[2], qv[4])\
    << CZ(qv[3], qv[7])\
    << CZ(qv[0], qv[4])\
    << RY(qv[7], pi / 2)\
    << RX(qv[8], pi / 2)\
    << RX(qv[9], pi / 2)\
    << CR(qv[0], qv[1], pi)\
    << CR(qv[2], qv[3], pi)\
    << RY(qv[4], pi / 2)\
    << RZ(qv[5], pi / 4)\
    << RX(qv[6], pi / 2)\
    << RZ(qv[7], pi / 4)\
    << CR(qv[8], qv[9], pi)\
    << CR(qv[1], qv[2], pi)\
    << RY(qv[3], pi / 2)\
    << RX(qv[4], pi / 2)\
    << RX(qv[5], pi / 2)\
    << CR(qv[9], qv[1], pi)\
    << RY(qv[1], pi / 2)\
    << RY(qv[2], pi / 2)\
    << RZ(qv[3], pi / 4)\
    << CR(qv[7], qv[8], pi)

```

The interface is used as follows:

pmeasure\_bin\_index is combined with run method in use. Example:

```

qvm.run(prog, qv)
dec_result = qvm.pmeasure_bin_index("0001000000")
print("0001000000 : ",dec_result)

```

Output results are as follows:

```
0001000000 : 0.001953123603016138
```

pmeasure\_dec\_index is combined with run method in use. Example:

```
qvm.run(prog, qv)
dec_result = qvm.pmeasure_dec_index("2")
print("2 : ",dec_result)
```

Output results are as follows:

```
2 : 0.001953123603016138
```

get\_prob\_dict is combined with run method in use. Example:

```
qvm.run(prog, qv)
res = qvm.get_prob_dict(qv)
```

prob\_run\_dict interface is the encapsulation of get\_prob\_dict and run, with the example as follows:

```
res_1 = qvm.prob_run_dict(prog, qv)
```

## 2.6.4 2.6.4 Partial-amplitude quantum simulator

At present, the classical computer simulates the quantum simulator mainly by full amplitude and single amplitude. Besides, the partial-amplitude quantum simulator can realize higher simulation efficiency under the lower hardware conditions.

### 2.6.4.1 2.6.4.1 Instructions

The usage of the partial-amplitude quantum simulator is very similar to that of the quantum simulator module described above. Firstly, initialize a partial-amplitude quantum simulator object through `PartialAmpQVM` in order to manage a series of subsequent behaviors.

```
from pyqpanda import *
from numpy import pi
machine = PartialAmpQVM()
```

Secondly, initialize, construct and carry the quantum programs, where the quantum programs are demonstrated in the partial-amplitude example program of ref: of pyQPanda.

```
machine.init_qvm()

q = machine.qAlloc_many(10)
c = machine.cAlloc_many(10)

# Building quantum programs
prog = QProg()
prog << hadamard_circuit(q)\
```

(continues on next page)



(continued from previous page)

```

<< CZ(q[1], q[5])\
<< CZ(q[3], q[7])\
<< CZ(q[0], q[4])\
<< RZ(q[7], pi / 4)\
<< RX(q[5], pi / 4)\
<< RX(q[4], pi / 4)\
<< RY(q[3], pi / 4)\
<< CZ(q[2], q[6])\
<< RZ(q[3], pi / 4)\
<< RZ(q[8], pi / 4)\
<< CZ(q[9], q[5])\
<< RY(q[2], pi / 4)\
<< RZ(q[9], pi / 4)\
<< CZ(q[2], q[3])

```

```
machine.run(prog)
```

Partial interfaces are used as follows:

● `pmeasure_bin_index(string)`, example

```

result = machine.pmeasure_bin_index("0000000000")
print(result)

```

Output results are as follows:

```
(-0.00647208746522665-0.006472080945968628j)
```

● `pmeasure_dec_index(string)`, example

```

result = machine.pmeasure_dec_index("1")
print(result)

```

Output results are as follows:

```
(-6.068964220062867e-10-0.009152906015515327j)
```

● `pmeasure_subset(state_index)`, example

```

state_index = ["0", "1", "2"]
result = machine.pmeasure_subset(state_index)
print(result)

```

Output results are as follows:

```
{'0': (-0.00647208746522665-0.006472080945968628j),
'1': (-6.068964220062867e-10-0.009152906015515327j),
'2': (-6.984919309616089e-10-0.009152908809483051j)}
```

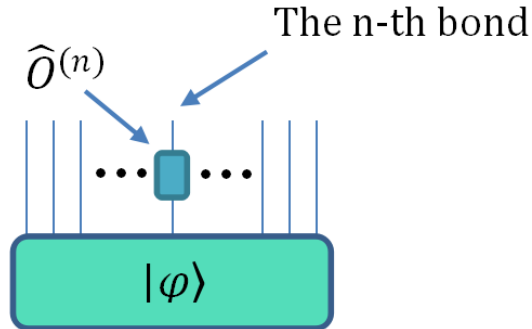
**Caution:** Partial old interfaces, including `get_qstate()`, `pmeasure(string)`, `pmeasure(string)` and `get_prob_dict(qvec, string)`, have been obsoleted.

## 2.6.5 Tensor network quantum simulator

For a spin system with  $N$  qubits, its dimension of Hilbert space is  $2^N$

With respect to the state evolution of the complex system, the traditional full-amplitude simulator considers it as a one-dimensional vector with  $2^N$  elements.

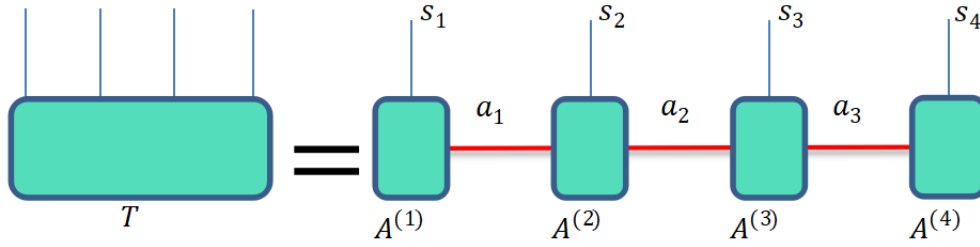
Viewed from the tensor network, the coefficient of the entire system quantum state corresponds to  $2^N$  dimension tensor (namely,  $N$ -order tensor with  $N$  indicators respectively taking 2 as dimension); the coefficient of the quantum operator is  $2^{2N}$  dimension tensor (namely,  $2N$ -order tensor with  $2N$  indicators respectively taking 2 as dimension); the quantum state can be denoted by the following graphs:



As the number of spins of the quantum system increases, the number of quantum state coefficients increases exponentially, which is considered as the exponential wall. The exponential wall limits the maximum number of simulation spins and performance of the traditional full-amplitude simulator.

However, the exponential wall can be addressed by the tensor network, so as to avoid its influence. In the tensor network, our simulation of quantum system, including operation and measurement of quantum logic gate, can be realized through the contraction and decomposition of tensors. The matrix product state is the most common representation in the tensor network and taken as TT (tensor-train) in the multi-linear algebra, as illustrated below.

The quantum state is decomposed as the representation form (namely, the right side of the equation); for some quantum logic gates in the quantum circuit, the global problems can be transformed into a local tensor problem, thereby effectively reducing the time complexity and space complexity.



### 2.6.5.1 Applications

In the simulation method of quantum circuit, it is very important to select the appropriate simulation backend. Different quantum circuit simulators have the application places as follows:

**Full-amplitude quantum simulator:** the full-amplitude quantum simulator can simultaneously simulate and store all the amplitudes of the quantum state. However, due to the restrictions of the machine memory, the limit of qubit is 50 bits and suitable for quantum circuits with low qubits and high depth, such as Google random quantum circuits with low bits and scenarios required for the acquisition of all simulation results.

**Partial-amplitude quantum simulator:** the partial-amplitude quantum simulator can simulate a higher number of qubits depending upon the low-bit quantum circuit amplitude simulation results provided by other simulators, but its simulation depth is reduced. Therefore, such simulator is usually used to obtain partial subset simulation results of quantum state amplitude.

**Single-amplitude quantum simulator:** the single-amplitude quantum simulator can simulate a higher qubit circuit diagram, have higher simulation performance, but will not increase exponentially as the number of qubits. With the increasing circuit depth, simulation performance decreases sharply; at the same time, the difficulty in simulation of more control gates also becomes its drawback, such simulator is suitable for the simulation of the high-bit low-depth quantum circuit, and usually suitable for the quick simulation to obtain single quantum state amplitude.

**Tensor network quantum simulator:** The tensor network simulator is similar to the single-amplitude quantum simulator; compared with the single-amplitude quantum simulator, it can simulate more control gates and have the performance advantage in higher-depth circuit simulation.

**Quantum cloud simulator:** the quantum cloud simulator can submit tasks to remote high-performance computing clusters for running, thereby breaking through the restrictions of the local hardware performance and supporting the quantum algorithm on the real quantum chip.

### 2.6.5.2 2.6.5.2 Instructions

In pyqpanda, the quantum circuit can be simulated through the tensor network through MPSQVM. Like the application methods of many other simulators, the tensor network simulator is provided with the same quantum simulator interface, for example, the simple example code is given below:

```
from numpy import pi
from pyqpanda import *

# Build a quantum virtual machine
qvm = MPSQVM()

# Initialization operation
qvm.set_configure(64, 64)
qvm.init_qvm()

q = qvm.qAlloc_many(10)
c = qvm.cAlloc_many(10)

# Building quantum programs
prog = QProg()
prog << hadamard_circuit(q)\
    << CZ(q[2], q[4])\
    << CZ(q[3], q[7])\
    << CNOT(q[0], q[1])\
    << Measure(q[0], c[0])\
    << Measure(q[1], c[1])\
    << Measure(q[2], c[2])\
    << Measure(q[3], c[3])

# The quantum program runs 100 times and returns the measurement
result = qvm.run_with_configuration(prog, c, 100)

# The number of times the printed quantum state appears in the result of
# multiple runs of the quantum program
print(result)

qvm.finalize()
```

### 2.6.5.3 Complete example code

The following example shows how to use the computing interfaces of the tensor network simulator.

```

from numpy import pi
from pyqpanda import *

qvm = MPSQVM()
qvm.set_configuration(64, 64)
qvm.init_qvm()

q = qvm.qAlloc_many(10)
c = qvm.cAlloc_many(10)

prog = QProg()
prog << hadamard_circuit(q)\
    << CZ(q[2], q[4])\
    << CZ(q[3], q[7])\
    << CNOT(q[0], q[1])\
    << CZ(q[3], q[7])\
    << CZ(q[0], q[4])\
    << RY(q[7], pi / 2)\
    << RX(q[8], pi / 2)\
    << RX(q[9], pi / 2)\
    << CR(q[0], q[1], pi)\
    << CR(q[2], q[3], pi)\
    << RY(q[4], pi / 2)\
    << RZ(q[5], pi / 4)\
    << Measure(q[0], c[0])\
    << Measure(q[1], c[1])\
    << Measure(q[2], c[2])

# Monte Carlo sampling simulation interface
result0 = qvm.run_with_configuration(prog, c, 100)

# Probability measurement interface
result1 = qvm.prob_run_dict(prog, [q[0], q[1], q[2]], -1)

print(result0)
print(result1)

qvm.finalize()

```

In the above code, “run\_with\_configuration” and prob\_run\_dict interfaces are respectively used for the sampling simulation and probability measurement of Monte Carlo, and output the simulation sampling results and the probability

corresponding to amplitude, and the above program has the following computing results:

```
# Monte Carlo sampling simulation results
{'0000000000': 7,
 '0000000001': 12,
 '0000000010': 13,
 '0000000011': 10,
 '0000000100': 16,
 '0000000101': 14,
 '0000000110': 12,
 '0000000111': 16}

# Probability measurement results
{'000': 0.124999999999999194,
 '001': 0.124999999999999185,
 '010': 0.124999999999999194,
 '011': 0.1249999999999992,
 '100': 0.124999999999999198,
 '101': 0.124999999999999194,
 '110': 0.124999999999999198,
 '111': 0.124999999999999208}
```

## 2.7 2.7 Qubit pool

### 2.7.1 2.7.1 Introduction

In previous QPanda, the application, management and control of qubits and classical registers are done by the simulator. A method independent to the simulator is provided, that is, qubits and classical registers are not managed by the simulator, and can be directly applied and released by the qubit pool provided. In order to better use qubits and classical registers, we further support physical addresses in substitution of the corresponding qubits.

### 2.7.2 2.7.2 Interface description

Qubit pool:

OriginQubitPool is to get single qubit pool, and the qubits are released by applying the pool object

get\_capacity is to get the maximum capacity

set\_capacity sets the capacity

get\_qubit\_by\_addr is to get the qubits through the physical address

Classical register pool:

OriginCMem is to get single classical register and the classical register is released by applying the pool object

get\_capacity is to get the maximum capacity

set\_capacity sets the capacity

get\_cbit\_by\_addr is to get the qubit through the physical address

The qubit pool has the same application and release methods as the simulator. A detailed description is given in [quantum simulator](#). Meanwhile, in use of the qubits and the classical registers, the parameters can be directly transmitted through the address corresponding to qubit.

For example:  $H(1)$  can be understood that H gate performs its function on the qubit taking 1 as the physical address.  $Measure(1, 1)$  can be understood that measurement is exerted to the qubit taking 1 as the physical address, and the results are stored into the classical register taking 1 as the address.

### 2.7.3 Example

```
from pyqpanda import *
from numpy import pi
if __name__ == "__main__":
    # The qubit can be separated from the virtual machine and obtain the singleton of the
    # corresponding pool. Different from QPanda, the object constructed here is the
    ↪ singleton
    # pool
    qpool = OriginQubitPool()
    qpool_1 = OriginQubitPool()
    cmem = OriginCMem()
    # Get the qubit pool capacity
    print("get_capacity : ", qpool.get_capacity())
    # Set the qubit pool capacity
    qpool.set_capacity(20)
    print("qpool get_capacity : ", qpool.get_capacity())
    # Since the qubit pool is a singleton object, set the capacity to 20 above, where qool_
    ↪ 1
    # will also get the capacity to 20
    print("qpool_1 get_capacity : ", qpool_1.get_capacity())
    # Apply for qubits from a qubit pool. In the singleton mode, ensure that the number of
    # applied qubits does not exceed the maximum capacity
    qv = qpool.qAlloc_many(6)
    cv = cmem.cAlloc_many(6)
    # Creating a simulator
    qvm = CPUQVM()
    qvm.init_qvm()
    prog = QProg()
    # Use the physical address directly as the qubit information input parameter
```

(continues on next page)

(continued from previous page)

```

prog << H(0)\
    << H(1)\
    << H(2)\
    << H(4)\
    << X(5)\
    << X1(2)\
    << CZ(2, 3)\
    << RX(3, pi / 4)\
    << CR(4, 5, pi / 2)\
    << SWAP(3, 5)\
    << CU(1, 3, pi / 2, pi / 3, pi / 4, pi / 5)\
    << U4(4, 2.1, 2.2, 2.3, 2.4)\
    << BARRIER([0, 1,2,3,4,5])\
    << BARRIER(0)

#print(prog)
# Measurement methods can also use bit physical addresses
res_0 = qvm.probab_run_dict(prog, [ 0,1,2,3,4,5 ])
#res_1 = qvm.probab_run_dict(prog, qv) # 同等上述方法
#print(res_0)
# The same classical bit address can also be used as a classical bit information.
→input
prog << Measure(0, 0)\
    << Measure(1, 1)\
    << Measure(2, 2)\
    << Measure(3, 3)\
    << Measure(4, 4)\
    << Measure(5, 5)

# Use the classical bit address to enter the parameter
res_2 = qvm.run_with_configuration(prog, [ 0,1,2,3,4,5 ], 5000)
# res_3 = qvm.run_with_configuration(prog, cv, 5000) # Same as above
#print(res_2)
qvm.finalize()

# At the same time, we can also use the QV applied here again to avoid the problem of
# applying bits for multiple times using virtual machines
qvm_noise = NoiseQVM()
qvm_noise.init_qvm()
res_4 = qvm_noise.run_with_configuration(prog, [ 0,1,2,3,4,5 ], 5000)
qvm_noise.finalize()

```

Output results are as follows:

```

get_capacity : 29
qpool get_capacity : 20
qpool_1 get_capacity : 20

```



## 2.8 Quantum measurement

Quantum measurement is to obtain necessary information by externally interfering with the quantum system, and the measurement gate is measured by the Monte Carlo method. It is represented by the following icons in the quantum circuit:



### 2.8.1 Interface introduction

The Chapter mainly introduces obtained quantum measurement objects, quantum programs containing quantum measurement according to run configuration, as well as quick measure.

In the quantum program, we need to measure a specific qubit and store the measurement results in the classical register. Besides, a measurement object can be obtained by the following way:

```
measure = Measure(qubit, cbit);
```

It can be seen that two parameters access to Measure, where the first parameter is the measurement qubit and the second parameter is the classical register.

If you want to measure all the qubits and store them in the corresponding classical register, the following operations can be conducted:

```
measureprog = measure_all(qubits, cbits);
```

qubits is classified as QVec; and cbits is classified as ClassicalCondition list.

---

#### Note

The `measure_all` returned values are classified as `QProg`.

---

After getting the program containing quantum measurement, we can invoke `directly_run` or `run_with_configuration` to get the measurement results of the quantum program.

`directly_run` function is to run the quantum programs and return the run results, and its usage is as follows:

```
prog = QProg()
prog << H(qubits[0])\
    << CNOT(qubits[0], qubits[1])\
    << CNOT(qubits[1], qubits[2])\
    << CNOT(qubits[2], qubits[3])\
    << Measure(qubits[0], cbits[0])

result = directly_run(prog)
```

`run_with_configuration` function is to count the multi-run measurement results of the quantum program, and its usage is as follows:

```
prog = QProg()
prog << H(qubits[0])\
    << H(qubits[0])\
    << H(qubits[1])\
    << H(qubits[2])\
    << measure_all(qubits, cbits)

result = run_with_configuration(prog, cbits, 1000)
```

The first parameter is the quantum program, the second parameter is ClassicalCondition list, and the third parameter is the running frequency.

## 2.8.2 Example

```
from pyqpanda import *

if __name__ == "__main__":
    init(QMachineType.CPU)
    qubits = qAlloc_many(4)
    cbits = cAlloc_many(4)

    # Building quantum programs
    prog = QProg()
    prog << H(qubits[0])\
        << H(qubits[1])\
```

(continues on next page)

(continued from previous page)

```

    << H(qubits[2])\
    << H(qubits[3])\
    << measure_all(qubits, cbits)

    # The quantum program runs 1000 times and returns the measurement
    result = run_with_configuration(prog, cbits, 1000)

    # Print measurement results
    print(result)
    finalize()

```

Output results are as follows:

```

{'0000': 59, '0001': 69, '0010': 52, '0011': 62,
'0100': 63, '0101': 67, '0110': 79, '0111': 47,
'1000': 73, '1001': 59, '1010': 72, '1011': 60,
'1100': 61, '1101': 71, '1110': 50, '1111': 56}

```

## 2.9 2.9 Probability measurement

Probability measurement is to obtain the amplitude of target qubit, where the target qubit can be single qubit or a set of qubits. In pyQPanda, probability measurement is also named as PMeasure. Probability measurement and [quantum measurement](#) are completely different; Measure is to perform a measurement, return a definite 0/1 result, and change the quantum state.

### 2.9.1 2.9.1 Interface introduction

PyQPanda provides three ways to obtain PMeasure results, including `prob_run_list`, `prob_run_tuple_list`, and `prob_run_dict`.

- `prob_run_list` is to get the list of probability measurement results of the target qubits.
- `prob_run_tuple_list` is to get the probability measurement results of the target qubit and belongs to a dictionary; its corresponding index is of the decimal system.
- `prob_run_dict` is to get the probability measurement results of the target qubit and belongs to a dictionary; its corresponding index is of the binary system.

The above-mentioned three functions have the same usage; and the following describes `prob_run_dict` as an example, and its usage is given below:

```

prog = QProg()
prog << H(qubits[0])\

```

(continues on next page)

(continued from previous page)

```
<< CNOT(qubits[0], qubits[1])\  
<< CNOT(qubits[1], qubits[2])\  
<< CNOT(qubits[2], qubits[3])  
  
result = prob_run_dict(prog, qubits, 3)
```

The first parameter is the quantum program; and the second parameter is `QVec` and defines the qubits concerned by us. When the third parameter is -1, all probability measurement results are obtained; when it is more than zero, the largest top several numbers are obtained.

## 2.9.2 Example

```
from pyqpanda import *  
  
if __name__ == "__main__":  
    init(QMachineType.CPU)  
    qubits = qAlloc_many(2)  
    cbits = cAlloc_many(2)  
  
    prog = QProg()  
    prog << H(qubits[0])\  
        << CNOT(qubits[0], qubits[1])  
  
    print("prob_run_dict: ")  
    result1 = prob_run_dict(prog, qubits, -1)  
    print(result1)  
  
    print("prob_run_tuple_list: ")  
    result2 = prob_run_tuple_list(prog, qubits, -1)  
    print(result2)  
  
    print("prob_run_list: ")  
    result3 = prob_run_list(prog, qubits, -1)  
    print(result3)  
  
    finalize()
```

Output results are as follows:

```
prob_run_dict:  
{'00': 0.4999999999999999, '01': 0.0, '10': 0.0, '11': 0.4999999999999999}  
prob_run_tuple_list:
```

(continues on next page)

(continued from previous page)

```
[ (0, 0.4999999999999999), (3, 0.4999999999999999), (1, 0.0), (2, 0.0) ]
prob_run_list:
[0.4999999999999999, 0.0, 0.0, 0.4999999999999999]
```

**Note**

Probability measurement is not applicable to the noise simulator.

## 2.10 2.10 OriginQ Cloud service

In complex quantum circuit simulation, there is a need to use the high-performance computer cluster or the real quantum computer to replace local computing with cloud computing, which will result in reducing the user's computing cost to a certain extent and obtaining the better computing experience.

Origin quantum cloud platform submits tasks to the remote quantum computer or computing cluster through the scheduling server and receives returned results, as shown in Figure below.



As pyqpanda encapsulates the quantum cloud simulator, it can send computing instructions to the computing server cluster of the origin quantum or the real quantum chip, and obtain computing results. Before using the simulators described below, you need to ensure that the corresponding simulators have been launched.

### 2.10.1 2.10.1 Real chip computing service

#### 2.10.1.1 2.10.1.1 Origin Wuyuan superconducting chip

Origin Wuyuan is the superconducting quantum computer developed independently by Origin Quantum on September 12, 2020 (which carries 6-qubit superconducting quantum processor KF C6-130). Benefited from the Origin superconducting quantum computing cloud platform, the quantum computer can escape from the laboratory, provide many potential industries with basic conditions to explore quantum computing, and promote the implementation and engineering development of quantum computing industry, thereby truly serving the human society.



**部分振幅量子虚拟机**  
计算全振幅中一个任意指定子集信息

立即使用



**全振幅量子虚拟机**  
一次计算出量子态的所有振幅

立即使用

**单振幅量子虚拟机**  
一次计算出 $2^n$ 个振幅中的一个

立即使用


```
#include "QPanda.h"
using namespace QPanda;
using namespace QPanda;

int main()
{
    auto qm = InitQuantumMachine(2, MachineType::CPU);
    auto q = qm.getMany();
    auto c = qm.getMany();

    QProg prog;
    QCMul c[0], c[1];

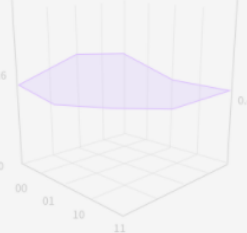
    auto gate = S1q[0];

    prog.addGate(c[0]);
    c[0] == H[0] == S[0] == C[0][0], c[1] == C[0][1], c[2] == gate;
    c[1].setTrigger(c[0]);
    c[2].setTrigger(c[1]);
}
```



**含噪声量子虚拟机**  
更贴近真实的量子计算机计算

立即使用



```
#include "QPanda.h"
using namespace QPanda;
using namespace QPanda;

int main()
{
    auto qm = InitQuantumMachine(2, MachineType::CPU);
    auto q = qm.getMany();
    auto c = qm.getMany();

    QProg prog;
    QCMul c[0], c[1];
}
```

As a bridge to link the user with the quantum computing system, the superconducting quantum computing cloud platform plays a crucial coordination and transfer role in the process in which the user initiates computing tasks to the quantum system and the quantum system completes the tasks and returns the computing results.

The chip topology structure of Origin Wuyuan is shown below:



The corresponding chip parameters are shown in the figure below:

量子比特编号	Q0	Q1	Q2	Q3	Q4	Q5
工作频率(GHz)	6456.185	4824.132	5319.214	4693.549	5214.995	4579.685
T1(μs)	21	22	22	17	18	15
T2(μs)	9	11	8	14	9	10
读取保真度F1/F0	0.967/0.988	0.945/0.980	0.967/0.995	0.949/0.986	0.933/0.974	0.896/0.944
平均单比特门保真度	0.9984	0.9988	0.9989	0.9985	0.9984	0.9994
两比特CZ	CZ01	CZ12	CZ23	CZ34	CZ45	
保真度	0.9653	0.9843	0.9807	0.9942	0.99	

The interface is described as follows:

- 1. Monte Carlo measurement interface: `real_chip_measure`, with the example as follows:

```

from pyqpanda import *
PI = 3.1416
# Create a quantum cloud simulator machine through QCloud()
QCM = QCloud()
# Initialization by passing in the current user's token
QCM.init_qvm("E02BB115D5294012AA88D4BE82603984", True)
q = QCM.qAlloc_many(6)
c = QCM.cAlloc_many(6)
# Building quantum programs
prog = QProg()
prog << hadamard_circuit(q)\
    << RX(q[1], PI / 4)\
    << RX(q[2], PI / 4)\
    << RX(q[1], PI / 4)\
    << CZ(q[0], q[1])\

```

(continues on next page)

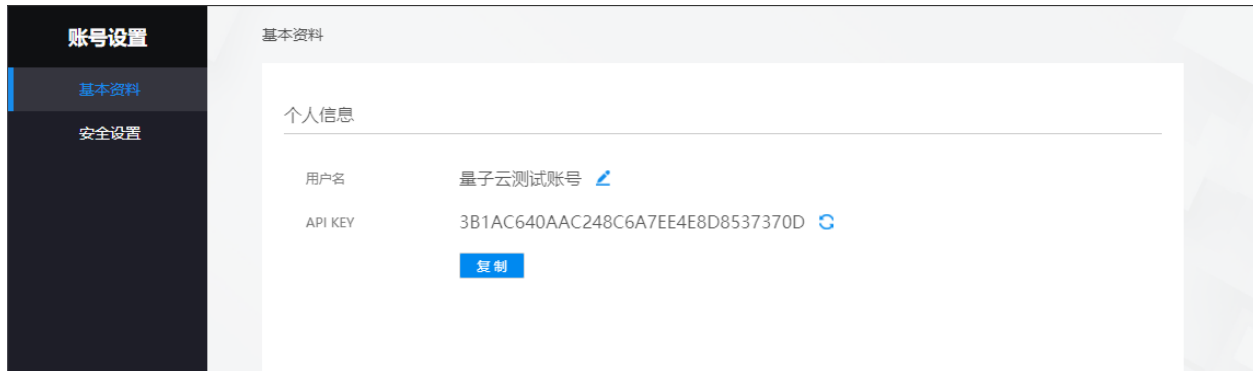
(continued from previous page)

```

    << CZ(q[1], q[2])\
    << Measure(q[0], c[0])\
    << Measure(q[1], c[1])
# To invoke the real chip computing interface, at least two parameters,
# quantum program and measurement times, are required. The next three
# default parameters are chip type and whether line mapping and line
# optimization functions are enabled.
result = QCM.real_chip_measure(prog, 1000, real_chip_type.origin_wuyuan_d4)
print(result)
QCM.finalize()

```

It should be noted in the above process that `init` requires a subscriber to import the subscriber validation token of the quantum cloud platform, and obtains it from the personal information of the OriginQ Cloud platform, and its details are shown in the following screenshot.



The output results are given below: the binary representation of the quantum state on the left and the corresponding probability of the number of measurements on the right:

- 2. Get quantum state (qst) tomography result interface: `get_state_tomography_density`, with the example shown below:

```

from pyqpanda import *

PI = 3.1416

# Create a quantum cloud simulator machine through QCloud()
QCM = QCloud()

# Initialization by passing in the current user's token
QCM.init_qvm("E02BB115D5294012AA88D4BE82603984", True)

q = QCM.qAlloc_many(6)
c = QCM.cAlloc_many(6)

```

(continues on next page)



(continued from previous page)

```

# Building quantum programs
prog = QProg()
prog << hadamard_circuit(q)\
    << RX(q[1], PI / 4)\
    << RX(q[2], PI / 4)\
    << RX(q[1], PI / 4)\
    << CZ(q[0], q[1])\
    << CZ(q[1], q[2])\
    << Measure(q[0], c[0])\
    << Measure(q[1], c[1])

# To invoke the real chip computing interface, at least two parameters,
# quantum program and measurement times, are required. The next three
# default parameters are chip type and whether line mapping and line
# optimization functions are enabled.
result = QCM.get_state_tomography_density( prog, 1000, real_chip_type.origin_wuyuan_
↪d4)
print(result)

QCM.finalize()

```

Output results are as follows:

```

[[ (0.26001013684744045+0j), (0.23492143943233657+0.000760263558033436j), (0.
↪01267105930055755+0.002280790674100364j), (-0.003547896604156095-0.
↪003294475418144968j)],
[ (0.23492143943233657-0.000760263558033436j), (0.250886974151039+0j), (0.
↪00937658388241254+0.003547896604156081j), (0.009883426254434847-0.
↪0025342118601114905j)],
[ (0.01267105930055755-0.002280790674100364j), (0.00937658388241254-0.
↪003547896604156081j), (0.2412569690826153+0j), (-0.2240243284338571-0.
↪009123162696401413j)],
[ (-0.003547896604156095+0.003294475418144968j), (0.009883426254434847+0.
↪0025342118601114905j), (-0.2240243284338571+0.009123162696401413j), (0.
↪24784591991890528+0j)]]

● 3. Get quantum state fidelity interface: ``get\_state\_fidelity``, with the example
↪shown below:

```

```

from pyqpanda import *

PI = 3.1416

```

(continues on next page)

(continued from previous page)

```

# Create a quantum cloud simulator machine through QCloud()
QCM = QCloud()

# Initialization by passing in the current user's token
QCM.init_qvm("E02BB115D5294012AA88D4BE82603984", True)

q = QCM.qAlloc_many(6)
c = QCM.cAlloc_many(6)

# Building quantum programs
prog = QProg()
prog << hadamard_circuit(q)\
    << RX(q[1], PI / 4)\
    << RX(q[2], PI / 4)\
    << RX(q[1], PI / 4)\
    << CZ(q[0], q[1])\
    << CZ(q[1], q[2])\
    << Measure(q[0], c[0])\
    << Measure(q[1], c[1])

# To invoke the real chip computing interface, at least two parameters, quantum
# program and measurement times, are required. The next three default
# parameters are chip type and whether line mapping and line optimization
# functions are enabled.
result = QCM.get_state_fidelity(prog, 1000, real_chip_type.origin_wuyuan_d4)
print(result)

QCM.finalize()

```

Output results are as follows:

```
0.942748
```

When applying the Origin Wuyuan real chip for measurement, you often encounter various errors; the following introduces some error information, you can find out them according to given error exception information.

- server connection failed refers to server downtime or server

connection failed

- api key error indicates that the API-Key parameter of the subscriber

is abnormal, please confirm the personnel information on the official website.

- un-activate products or lack of computing power indicates that the

subscriber does not activate products or is lack of computing power.

- build system error refers to the run error of the compiling system
- exceeding maximum timing sequence refers to the relatively long quantum program timing
- unknown task status refers to the abnormalities of other task states

---

## Note

- Before using the corresponding computing interface, there is a need to ensure that the current subscriber has enabled the product. Otherwise, it is possible not to successfully submit computing tasks.
  - In the noise simulation, there are respective three single-gate and double-gate parameters of decoherence, which are different from other noise.
  - The measurement frequency supported by Origin Wuyuan measurement is between 1000 and 10000, and currently only supports the simulation of the quantum circuit of 6 or below qubits. Other quantum chips will be added in the future, please look forward to it.
  - If you experience any problems in use, please give user feedback to us. We will solve your problems as soon as possible.
- 

## 2.10.2 2.10.2 Origin high-performance computing cluster cloud service

The high-performance computing cluster of Origin Quantum provides various simulator computing backends with powerful functions, and is applicable to the simulation requirements for quantum circuits in different conditions, and the following introduces the complete example program:

```
from pyqpanda import *
import numpy as np

# Create a quantum cloud simulator machine through QCloud()
QCM = QCloud()

# Initialization by passing in the current user's token
QCM.init_qvm("3B1AC640AAC248C6A7EE4E8D8537370D")

qlist = QCM.qAlloc_many(6)
clist = QCM.cAlloc_many(6)

# Building quantum program. It can be entered manually, from OriginIR or QASM syntax
# files, etc.
measure_prog = QProg()
measure_prog << hadamard_circuit(qlist)\
    << CZ(qlist[1], qlist[5])\
    << Measure(qlist[0], clist[0])\
    << Measure(qlist[1], clist[1])
```

(continues on next page)

(continued from previous page)

```

pmeasure_prog = QProg()
pmeasure_prog << hadamard_circuit(qlist)\
    << CZ(qlist[1], qlist[5])\
    << RX(qlist[2], np.pi / 4)\
    << RX(qlist[1], np.pi / 4)\

# To call the calculation interface of full-amplitude Monte Carlo measurement
# operation, two parameters are required: quantum program and measurement times
result = QCM.full_amplitude_measure(measure_prog, 1000)
print(result)

```

### 2.10.2.1 Full-amplitude simulation cloud computing

The interface is described as follows:

- `full_amplitude_measure` (full-amplitude Monte Carlo measurement):

```

result0 = QCM.full_amplitude_measure(measure_prog, 100)
print(result0)

```

The second parameter required **for input is** the number of measurements,

and the output results are given below: the binary representation of the quantum state on the left and the corresponding probability of the number of measurements on the right:

```

{'00': 0.25,
 '01': 0.28,
 '10': 0.22,
 '11': 0.25}

```

- `full_amplitude_pmeasure` (full-amplitude probability measurement):

```

result1 = QCM.full_amplitude_pmeasure(pmeasure_prog, [0, 1, 2])
print(result1)

```

The second parameter required for input is the measurement qubit, and the output results are given below: the binary representation of the quantum state on the left and the corresponding probability of the number of measurements on the right:

```

{'000': 0.125,
 '001': 0.125,
 '010': 0.125,

```

(continues on next page)

(continued from previous page)

```
'011': 0.125,
'100': 0.125,
'110': 0.125,
'111': 0.125}
```

### 2.10.2.2 2.10.2.2 Partial-amplitude simulation cloud computing

● `partial_amplitude_pmeasure` (partial-amplitude probability measurement):

```
result2 = QCM.partial_amplitude_pmeasure(pmeasure_prog, ["0", "1", "2"])
print(result2)
```

The second parameter required for input is the decimal representation of the measured quantum state amplitude, and the output results are given below: the decimal representation of the quantum state on the left and the plural amplitudes on the right:

```
{'0': (0.08838832192122936-0.08838833495974541j),
'1': (0.08838832192122936-0.08838833495974541j),
'2': (0.08838832192122936-0.08838833495974541j) }
```

### 2.10.2.3 2.10.2.3 Single-amplitude cloud computing

● `single_amplitude_pmeasure` (single-amplitude probability measurement):

```
result3 = QCM.single_amplitude_pmeasure(pmeasure_prog, "0")
print(result3)
```

The second parameter required for input is the measured amplitude (decimal representation), and the output results are given below: only the plural amplitude corresponding to one quantum state is output:

```
(0.08838833056846361-0.08838833850593952j)
```

### 2.10.2.4 2.10.2.4 Noise simulation cloud computing

● `noise_measure` (noise simulator measurement):

```
QCM.set_noise_model(NoiseModel.BIT_PHASE_FLIP_OPERATOR, [0.01], [0.02])
result4 = QCM.noise_measure(measure_prog, 100)
print(result4)
```

The noise parameters are set through `set_noise_model`; the first parameter is the noise model, and the subsequent parameters are respectively the single-gate noise parameter and the double-gate noise parameter, where the noise model is defined as follows:

```
enum NOISE_MODEL
{
    DAMPING_KRAUS_OPERATOR,
    DEPHASING_KRAUS_OPERATOR,
    DECOHERENCE_KRAUS_OPERATOR_P1_P2,
    BITFLIP_KRAUS_OPERATOR,
    DEPOLARIZING_KRAUS_OPERATOR,
    BIT_PHASE_FLIP_OPERATOR,
    PHASE_DAMPING_OPERATOR,
    DECOHERENCE_KRAUS_OPERATOR,
    PAULI_KRAUS_MAP,
    KRAUS_MATRIX_OPERATOR,
    MIXED_UNITARY_OPERATOR,
};
```

It can be obtained through pyqpanda enumerated `NoiseModel`; the interface output results are given below: the binary representation of the quantum state on the left and the corresponding probability of the number of measurements on the right:

```
{'00': 0.26,
 '01': 0.21,
 '10': 0.29,
 '11': 0.24}
```

### 2.10.2.5 2.10.2.5 Get the quantum state tomography results

```
from pyqpanda import *

qm = QCloud()

qm.init_qvm("E02BB115D5294012AA88D4BE82603984")

qlist = qm.qAlloc_many(6)
clist = qm.cAlloc_many(6)

prog = QProg()
prog << hadamard_circuit(qlist)\
    << CZ(qlist[1], qlist[5])\
    << Measure(qlist[0], clist[0])\
```

(continues on next page)

(continued from previous page)

```

    << Measure(qlist[1], clist[1])

result = qm.get_state_tomography_density(prog, 1000)
print(result)
qm.finalize()

```

Apply the way similar to Monte Carlo measurement, with the output results as follows:

```

[[ (0.2587544156749868-8.004934191929294e-19j), (0.251211804846972+0.
↪ 001414451655940455j), (-0.008943457002333129+0.0014876032160007612j), (-0.
↪ 0040247742512866495+0.007632530135083866j)],
[ (0.2512118048469719-0.001414451655940456j), (0.25003193002089275-6.776263578034404e-
↪ 19j), (0.0026098957997104447-0.0145657172180014j), (0.001739623577306608+0.
↪ 003430686695967179j)],
[ (-0.008943457002333132-0.001487603216000763j), (0.002609895799710438+0.
↪ 0145657172180014j), (0.24548904782784528+2.1684043449710093e-19j), (0.
↪ 2290859282493824+0.000791060320984212j)],
[ (-0.0040247742512866495-0.007632530135083866j), (0.001739623577306601-0.
↪ 0034306866959671776j), (0.2290859282493824-0.0007910603209842113j), (0.
↪ 2457246064762752-2.710505431213761e-20j)]]

```

## Note

- Before using the corresponding computing interface, there is a need to ensure that the current subscriber has enabled the product. Otherwise, it is possible not to successfully submit computing tasks.
- In the noise simulation, there are respective three single-gate and double-gate parameters of decoherence, which are different from other noise.
- The measurement frequency supported by Origin Wuyuan measurement is between 1000 and 10000, and currently only supports the simulation of the quantum circuit of 6 or below qubits. Other quantum chips will be added in the future, please look forward to it.
- If you experience any problems in use, please give user feedback to us. We will solve your problems as soon as possible.





## 3 QUANTUM PROGRAM INFORMATION

### 3.1 3.1 NodeIter

NodeIter is QProg or QCircuit iterator externally provided by pyQPanda. We can control our quantum programs conveniently through NodeIter.

#### 3.1.1 3.1.1 Interface introduction

At present, NodeIter has the main operations below:

obtaining the next node

```
iter = iter.get_next()
```

obtaining the previous node

```
iter = iter.get_pre()
```

obtaining the node type

```
type = iter.get_node_type()
```

configuring QProg through iterator

```
type = iter.get_node_type()
if pq.NodeType.PROG_NODE == type:
    prog = pq.QProg(iter)
```

configuring the quantum circuit QCircuit through iterator

```
type = iter.get_node_type()
if pq.NodeType.CIRCUIT_NODE == type:
    cir = pq.QCircuit(iter)
```

configuring QGate through iterator

```
type = iter.get_node_type()
if pq.NodeType.GATE_NODE == type:
    gate = pq.QGate(iter)
```

configuring QIfProg through iterator

```
type = iter.get_node_type()
if pq.NodeType.QIF_START_NODE == type:
    if_prog = pq.QIfProg(iter)
```

configuring QWhileProg through iterator

```
type = iter.get_node_type()
if pq.NodeType.WHILE_START_NODE == type:
    while_prog = pq.QWhileProg(iter)
```

configuring QMeasure through iterator

```
type = iter.get_node_type()
if pq.NodeType.MEASURE_GATE == type:
    measure_gate = pq.QMeasure(iter)
```

### 3.1.2 Example

The following example program has the function to browse one QProg through NodeIter and output the types of node logic gates:

```
import pyqpanda.pyQPanda as pq
import math

machine = pq.init_quantum_machine(pq.QMachineType.CPU)
q = machine.qAlloc_many(8)
c = machine.cAlloc_many(8)
prog = pq.QProg()

prog << pq.H(q[0]) << pq.S(q[2]) << pq.CNOT(q[0], q[1]) \
    << pq.CZ(q[1], q[2]) << pq.CR(q[1], q[2], math.pi/2)
iter = prog.begin()
iter_end = prog.end()
while iter != iter_end:
    if pq.NodeType.GATE_NODE == iter.get_node_type():
        gate = pq.QGate(iter)
```

(continues on next page)

(continued from previous page)

```

        print(gate.gate_type())
        iter = iter.get_next()
    else:
        print('Traversal End.\n')

pq.destroy_quantum_machine(machine)

```

Reverse browsing:

```

import pyqpanda.pyQPanda as pq
import math

machine = pq.init_quantum_machine(pq.QMachineType.CPU)
q = machine.qAlloc_many(8)
c = machine.cAlloc_many(8)
prog = pq.QProg()

prog << pq.H(q[0]) << pq.S(q[2]) << pq.CNOT(q[0], q[1]) \
    << pq.CZ(q[1], q[2]) << pq.CR(q[1], q[2], math.pi/2)
iter_head = prog.head()
iter = prog.last()
while iter != iter_head:
    if pq.NodeType.GATE_NODE == iter.get_node_type():
        gate = pq.QGate(iter)
        print(gate.gate_type())
        iter = iter.get_pre()
    else:
        print('Traversal End.\n')

```

## 3.2 3.2 Logic gate statistics

### 3.2.1 3.2.1 Introduction

Logic gate statistics refers to a method for counting the number of the quantum logic gates (including measurement gates) in quantum programs, quantum circuits, quantum cycle control or quantum condition control.

### 3.2.2 3.2.2 Interface introduction

We create one quantum program firstly through pyqpanda:

```
prog = QProg()
prog << X(qubits[0]) << Y(qubits[1])\
    << H(qubits[0]) << RX(qubits[0], 3.14)\
    << Measure(qubits[0], cbits[0])
```

invoke `get_qgate_num` to count the number of quantum logic gates;

```
number = get_qgate_num(prog)
```

---

#### Note

The method of counting the number of quantum logic gates in `QCircuit`, `QWhileProg` and `QIfProg` is similar to the method of counting `QProg`.

---

### 3.2.3 3.2.3 Example

```
from pyqpanda import *

if __name__ == "__main__":
    qvm = init_quantum_machine(QMachineType.CPU)
    qubits = qvm.qAlloc_many(2)
    cbits = qvm.cAlloc_many(2)

    prog = QProg()

    # Building quantum programs
    prog << X(qubits[0]) << Y(qubits[1])\
        << H(qubits[0]) << RX(qubits[0], 3.14)\
        << Measure(qubits[0], cbits[0])

    # Statistics the number of logical gates
    number = get_qgate_num(prog)
    print("QGate number: " + str(number))

    qvm.finalize()
```

Output results are as follows:

```
QGate number: 4
```

**Warning:** The interface name in the new version is changed, and the old interface `count_gate` is replaced by `get_qgate_num`. Please note that `count_gate` will be removed in the next version.

## 3.3 Counting quantum program clock cycle

### 3.3.1 Introduction

Estimate the running time of a quantum program under the known running time of the quantum logic gates. The time of the quantum logic gates is set to the item metadata configuration file `QPandaConfig.xml`; a default is given if such time is not set, where the time of single quantum gate is defaulted as 2, and that of the double quantum gates is defaulted as 5.

The configuration file can be set according to the following example:

```
"QGate": {
  "SingleGate":{
    "U3":{"time":1}
  },
  "DoubleGate":{
    "CNOT":{"time":2},
    "CZ":{"time":2}
  }
}
```

### 3.3.2 Interface introduction

We create one quantum program firstly through `pyqpanda`:

```
prog = QProg()
prog << H(qubits[0]) << CNOT(qubits[0], qubits[1])\
    << iSWAP(qubits[1], qubits[2]) << RX(qubits[3], np.pi / 4)
```

invoke `get_qprog_clock_cycle` interface to get the clock cycle of the quantum program

```
clock_cycle = get_qprog_clock_cycle(qvm, prog)
```

### 3.3.3 Example

```
from pyqpanda import *
import numpy as np

if __name__ == "__main__":
    qvm = init_quantum_machine(QMachineType.CPU)
    qubits = qvm.qAlloc_many(4)
    cbits = qvm.cAlloc_many(4)

    # Building quantum programs
    prog = QProg()
    prog << H(qubits[0]) << CNOT(qubits[0], qubits[1])\
        << iSWAP(qubits[1], qubits[2]) << RX(qubits[3], np.pi / 4)

    # Statistical quantum program clock cycle
    clock_cycle = get_qprog_clock_cycle(prog, qvm)

    print(clock_cycle)
    destroy_quantum_machine(qvm)
```

Output results are as follows:

```
5
```

**Warning:** The interface name in the new version is changed, and the old interface `get_clock_cycle` will be replaced by `get_qprog_clock_cycle`. Please note that `get_clock_cycle` will be removed in the next version.

## 3.4 Get the corresponding matrix of the quantum circuit

The `get_matrix` interface can get the corresponding matrix of the input circuit and has three output parameters, where one parameter is quantum circuit `QCircuit` (or `QProg`), and other two parameters are optional parameters: the starting and ending positions of the iterator are used for designating one circuit interval to get corresponding matrix information; if the parameters are null, the matrix information of the entire quantum circuit is obtained.

---

### Note

It should be noted in use of `get_matrix` that the quantum circuit does not involve measurement.

---

### 3.4.1 3.4.1 Example

```
import pyqpanda.pyQPanda as pq
import math

class InitQMachine:
def __init__(self, quBitCnt, cBitCnt,\
machineType = pq.QMachineType.CPU):
    self.m_machine = pq.init_quantum_machine(machineType)
    self.m_qlist = self.m_machine.qAlloc_many(quBitCnt)
    self.m_clist = self.m_machine.cAlloc_many(cBitCnt)
    self.m_prog = pq.QProg()

    def __del__(self):
        pq.destroy_quantum_machine(self.m_machine)

def test_get_matrix(q, c):
    prog = pq.QProg()

    # Building quantum programs
    prog << pq.H(q[0]) \
        << pq.S(q[2]) \
        << pq.CNOT(q[0], q[1]) \
        << pq.CZ(q[1], q[2]) \
        << pq.CR(q[1], q[2], math.pi/2)

    # Obtain the line corresponding matrix
    result_mat = pq.get_matrix(prog)

    # Print matrix information
    pq.print_matrix(result_mat)

if __name__=="__main__":
    init_machine = InitQMachine(16, 16)
    qlist = init_machine.m_qlist
    clist = init_machine.m_clist
    machine = init_machine.m_machine

    test_get_matrix(qlist, clist)
    print("Test over.")
```

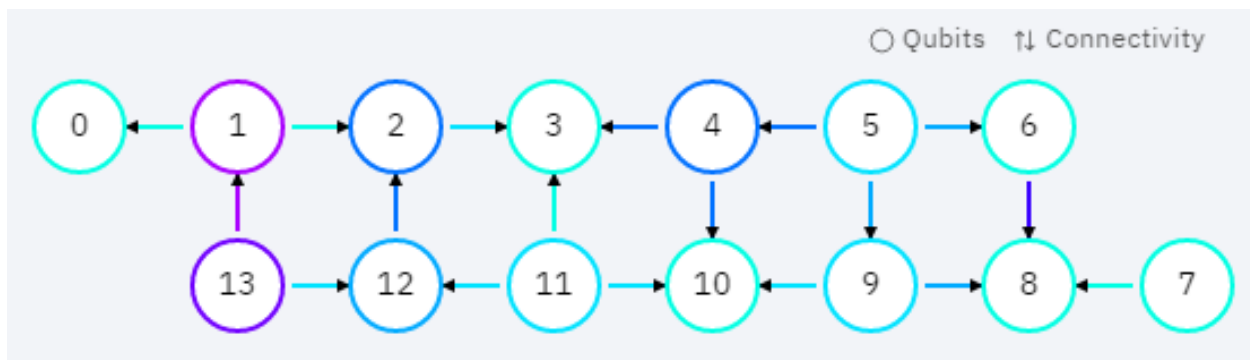
The specific steps are as follows:

1. Initialize a quantum simulator object with `init_quantum_machine` (`pq.QMachineType.CPU`) in order to manage a series of subsequent behaviors

2. Initialize the number of qubits and classical registers with `machine.qAlloc_many()` and `machine.cAlloc_many()`, where respective 8 qubits and classical register are applied.
3. Create prog
4. Determine the circuit interval of computing matrix information, namely setting of `iter_start` and `iter_end`
5. Invoke `get_matrix` interface to output the corresponding matrix of the quantum circuit, and print obtained matrix information through `print_mat` interface.

## 3.5 Judge whether quantum logic gate matches with the quantum topology

Each quantum chip has its own particular qubit topology, for example, `ibmq_16_melbourne` provided by IBMQ:



It can be seen from the Figure that the qubits in the quantum chip are not connected in pairs, and those not interconnected cannot directly execute multi-gate operations. Therefore, it is necessary to judge whether the double-gate (multi-gate) operation in the quantum program is in conformity with the qubit topology prior to the execution of the quantum program.

### 3.5.1 Interface introduction

The `is_match_topology` is to judge whether the quantum logic gate is in conformity with the qubit topology. The first input parameter is the target quantum logic gate `QGate`, the second input parameter is the qubit topology, and the returned value is Boolean value. This indicates whether the target quantum logic gate is in conformity with the qubit topology. True means that the target quantum logic gate is in conformity with the qubit topology. False means that the target quantum logic gate is not in conformity with the qubit topology.

```
import pyqpanda.pyQPanda as pq
import math

class InitQMachine:
def __init__(self, quBitCnt, cBitCnt,\
machineType = pq.QMachineType.CPU):
```

(continues on next page)



(continued from previous page)

```

        self.m_machine = pq.init_quantum_machine(machineType)
        self.m_qlist = self.m_machine.qAlloc_many(quBitCnt)
        self.m_clist = self.m_machine.cAlloc_many(cBitCnt)
        self.m_prog = pq.QProg()

    def __del__(self):
        pq.destroy_quantum_machine(self.m_machine)

def test_is_match_topology(q, c):
    cx = pq.CNOT(q[1], q[3])

    # Building a Topology
    qubits_topology = \
[[0,1,0,0,0],[1,0,1,1,0],[0,1,0,0,0],[0,1,0,0,1],[0,0,0,1,0]]

    # Determine whether the logic gate conforms to the quantum topology
    if (pq.is_match_topology(cx,qubits_topology)) == True:
        print('Match !\n')
    else:
        print('Not match.')

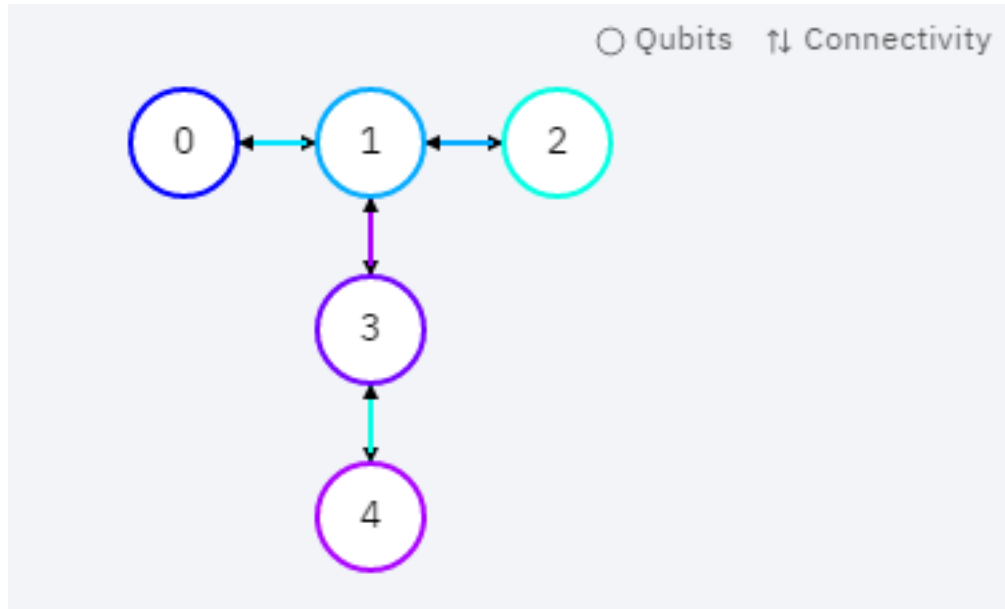
if __name__=="__main__":
    init_machine = InitQMachine(16, 16)
    qlist = init_machine.m_qlist
    clist = init_machine.m_clist
    machine = init_machine.m_machine
    test_is_match_topology(qlist, clist)
    print("Test over.")

```

Prior to the use of `is_match_topology`, the adjacency matrix `qubits_topology` of the qubit topology of the designated quantum chip should be created.

It can be seen from the above example, `qubits_topology` includes 5 qubits. The qubit topological diagram is shown below:

CNOT logic gate operates qubits 1# and 3#. However, it can be seen from the topological diagram that qubits 1# and 3# are interconnected, and therefore the result should be true.



## 3.6 3.6 Get adjacent quantum logic gates at the designated position.

The `get_adjacent_qgate_type` interface can get designated adjacent quantum logic gates in the quantum program. The first input parameter is the target quantum program `QProg`, the second input parameter is the iterator of the target quantum logic gate in the quantum program, and the returned result is a set of iterators of adjacent target quantum logic gates.

### 3.6.1 3.6.1 Example

```

import pyqpanda.pyQPanda as pq
import math

class InitQMachine:
    def __init__(self, quBitCnt, cBitCnt, machineType = pq.QMachineType.CPU):
        self.m_machine = pq.init_quantum_machine(machineType)
        self.m_qlist = self.m_machine.qAlloc_many(quBitCnt)
        self.m_clist = self.m_machine.cAlloc_many(cBitCnt)
        self.m_prog = pq.QProg()

    def __del__(self):
        pq.destroy_quantum_machine(self.m_machine)

def test_get_adjacent_qgate_type(qlist, clist):
    prog = pq.QProg()

```

(continues on next page)

(continued from previous page)

```

# Building quantum programs
prog << pq.T(qlist[0]) \
    << pq.CNOT(qlist[1], qlist[2]) \
    << pq.Reset(qlist[1]) \
    << pq.H(qlist[3]) \
    << pq.H(qlist[4])

iter = prog.begin()
iter = iter.get_next()
type = iter.get_node_type()
if pq.NodeType.GATE_NODE == type:
    gate = pq.QGate(iter)
    print(gate.gate_type())

# Gets the types of logical gates before and after the specified location
list = pq.get_adjacent_qgate_type(prog, iter)
print(len(list))
print(len(list[0].m_target_qubits))
print(list[1].m_is_dagger)

node_type = list[0].m_node_type
print(node_type)
if node_type == pq.NodeType.GATE_NODE:
    gateFront = pq.QGate(list[0].m_iter)
    print(gateFront.gate_type())

node_type = list[1].m_node_type
print(node_type)
if node_type == pq.NodeType.GATE_NODE:
    gateBack = pq.QGate(list[1].m_iter)
    print(gateBack.gate_type())

if __name__ == "__main__":
    init_machine = InitQMachine(16, 16)
    qlist = init_machine.m_qlist
    clist = init_machine.m_clist
    machine = init_machine.m_machine
    test_get_adjacent_qgate_type(qlist, clist)
    print("Test over.")

```

The above example shows how to use `get_adjacent_qgate_type` interface:

1. Create one quantum program prog;
2. Designate position information, namely setting of iter

3. Invoke `get_adjacent_qgate_type` interface to get a set of iterators of adjacent iter logic gates. The types of obtained logic gates are respectively printed in the last four rows of the example code

When using `get_adjacent_qgate_type` interface, we need to note the following points:

1. A set of the iterators of adjacent target quantum logic gates always includes two elements; The first element is an iterator of the previous quantum logic gate, and the second element is an iterator of the next logic gate.
2. If the target quantum logic gate is the first node of the quantum program, only the iterator of the following target quantum logic gate can be obtained in a set of iterators of adjacent target quantum logic gates (as the output parameter), and the first element in the set is the null iterator.
3. If the target quantum logic gate is the last quantum logic gate in the quantum program, only the iterator of the previous target quantum logic gate can be obtained in a set of iterators of adjacent target quantum logic gates (as the output parameter), and the second element in the set is the null iterator.
4. If the previous node of the target quantum logic gate is QIf or QWhile, only the iterator of the following target quantum logic gate can be obtained in a set of iterators of adjacent target quantum logic gates (as the output parameter), and the first element in the set is the null iterator.
5. If the following node of the target quantum logic gate is QIf or QWhile, only the iterator of the previous target quantum logic gate can be obtained in a set of iterators of adjacent target quantum logic gates (as the output parameter), and the second element in the set is the null iterator.
6. If the target quantum logic gate is the first quantum logic gate of QWhile, only the iterator of the following target quantum logic gate can be obtained in a set of iterators of adjacent target quantum logic gates (as the output parameter), and the first element in the set is the null iterator.
7. If the target quantum logic gate is the last quantum logic gate of QWhile, only the iterator of the previous target quantum logic gate can be obtained in a set of iterators of adjacent target quantum logic gates (as the output parameter), and the second element in the set is the null iterator.

## 3.7 Judge whether two quantum logic gates are interchanged for their positions

The `is_swappable` interface can judge whether two designated quantum logic gates in the quantum program are interchanged for their positions. The first input parameter is the quantum program QProg; the second and third input parameters are the iterators of two quantum logic gates to be judged. The returned value is Boolean value; True represents the interchangeable position; and False represents the non-interchangeable position.

### 3.7.1 Example

The following example demonstrates how to use `is_swappable` interface.

1. Create a quantum program prog. Here, a slightly complex quantum program with nested nodes is enumerated;
2. Get iterators at two designated positions of the nested node cir: `iter_first` and `iter_second`;
3. Invoke `is_swappable` interface to judge whether two designated logic gates are interchanged for their positions, and output changeable judgement results on the console.

```
import math

class InitQMachine:
def __init__(self, quBitCnt, cBitCnt,\
machineType = pq.QMachineType.CPU):
    self.m_machine = pq.init_quantum_machine(machineType)
    self.m_qlist = self.m_machine.qAlloc_many(quBitCnt)
    self.m_clist = self.m_machine.cAlloc_many(cBitCnt)
    self.m_prog = pq.QProg()

    def __del__(self):
        pq.destroy_quantum_machine(self.m_machine)

# Test interface: judge whether two designated logic gates are interchanged
# for their positions
def test_is_swappable(q, c):
    prog = pq.QProg()
    cir = pq.QCircuit()
    cir2 = pq.QCircuit()
    cir2 << pq.H(q[3]) << pq.RX(q[1], math.pi/2) << pq.T(q[2])\
    << pq.RY(q[3], math.pi/2) << pq.RZ(q[2], math.pi/2)
    cir2.set_dagger(True)
    cir << pq.H(q[1]) << cir2 << pq.CR(q[1], q[2], math.pi/2)
    prog << pq.H(q[0]) << pq.S(q[2]) \
    << cir\
    << pq.CNOT(q[0], q[1]) << pq.CZ(q[1], q[2]) << pq.measure_all(q,c)

    iter_first = cir.begin()

    iter_second = cir2.begin()
    #iter_second = iter_second.get_next()
    #iter_second = iter_second.get_next()
    #iter_second = iter_second.get_next()

    type = iter_first.get_node_type()
```

(continues on next page)

(continued from previous page)

```

    if pq.NodeType.GATE_NODE == type:
        gate = pq.QGate(iter_first)
        print(gate.gate_type())

    type = iter_second.get_node_type()
    if pq.NodeType.GATE_NODE == type:
        gate = pq.QGate(iter_second)
        print(gate.gate_type())

    if (pq.is_swappable(prog, iter_first, iter_second)) == True:
        print('Could be swapped !\n')
    else:
        print('Could NOT be swapped.')

if __name__=="__main__":
    init_machine = InitQMachine(16, 16)
    qlist = init_machine.m_qlist
    clist = init_machine.m_clist
    machine = init_machine.m_machine

    test_is_swappable(qlist, clist)
    print("Test over.")

```

## 3.8 Judge whether the logic gates are of a set of quantum logic gates supported by the quantum chip

A set of quantum logic gates supported by the quantum chip can be configured in the metadata configuration file QPandaConfig.xml. If we do not set the configuration files, QPanda sets a default set of quantum logic gates.

The default collection is as follows:

```

single_gates.push_back("RX");
single_gates.push_back("RY");
single_gates.push_back("RZ");
single_gates.push_back("X1");
single_gates.push_back("H");
single_gates.push_back("S");

double_gates.push_back("CNOT");
double_gates.push_back("CZ");
double_gates.push_back("ISWAP");

```

The configuration file can be set according to the following example:

```
"QGate": {
    "SingleGate":{
        "U3":{"time":1}
    },
    "DoubleGate":{
        "CNOT":{"time":2},
        "CZ":{"time":2}
    }
}
```

It can be seen from the above examples that the quantum chip supports RX, RY, RZ, S, H, X1, CNOT, CZ and ISWAP gates. We may call the interface `is_supported_qgate_type` upon the configuration of the configuration file, to determine whether the logic gate falls into the quantum logic gate set that the quantum chip supports. The interface `is_supported_qgate_type` only has one parameter: target quantum logic gate.

```
import pyqpanda.pyQPanda as pq
import math

class InitQMachine:
def __init__(self, quBitCnt, cBitCnt,\
machineType = pq.QMachineType.CPU):
    self.m_machine = pq.init_quantum_machine(machineType)
    self.m_qlist = self.m_machine.qAlloc_many(quBitCnt)
    self.m_clist = self.m_machine.cAlloc_many(cBitCnt)
    self.m_prog = pq.QProg()

    def __del__(self):
        pq.destroy_quantum_machine(self.m_machine)

def test_support_qgate_type():
    machine = pq.init_quantum_machine(pq.QMachineType.CPU)
    q = machine.qAlloc_many(8)
    c = machine.cAlloc_many(8)

    prog = pq.QProg()
    prog << pq.H(q[1])
    result = pq.is_supported_qgate_type(prog.begin())
    if result == True:
        print('Support !\n')
    else:
        print('Unsupport !')

if __name__=="__main__":
```

(continues on next page)

(continued from previous page)

```
init_machine = InitQMachine(16, 16)
qlist = init_machine.m_qlist
clist = init_machine.m_clist
machine = init_machine.m_machine

test_support_qgate_type()
print("Test over.")
```

---

## Note

The user may access the default configuration file QPandaConfig.json through the following link, and place it under the same directory as the executive program, which will automatically parse the file.

---

## 3.9 Character drawing of quantum circuit

At present, PyQPanda provides three visualization methods of quantum circuit. Please refer to the following examples for specific usage methods.

### 3.9.1 Example

```
import pyqpanda.pyQPanda as pq
from pyqpanda.Visualization.circuit_draw import *
import math

class InitQMachine:
    def __init__(self, quBitCnt, cBitCnt, machineType = pq.QMachineType.CPU):
        self.m_machine = pq.init_quantum_machine(machineType)
        self.m_qlist = self.m_machine.qAlloc_many(quBitCnt)
        self.m_clist = self.m_machine.cAlloc_many(cBitCnt)

    def __del__(self):
        pq.destroy_quantum_machine(self.m_machine)

def test_print_qcircuit(q, c):
    # Building quantum programs
    prog = pq.QCircuit()
    prog << pq.CU(1, 2, 3, 4, q[0], q[5]) << pq.H(q[0]) << pq.S(q[2])\
        << pq.CNOT(q[0], q[1]) << pq.CZ(q[1], q[2])\
    << pq.CR(q[2], q[1], math.pi/2)
    prog.set_dagger(True)
```

(continues on next page)



(continued from previous page)

```

    print('draw_qprog:')

# Through print directly outputs the character drawing of quantum
# circuit. This method will output the quantum circuit in the console,
# and the output format is uft8 encoding. Therefore, in the console
# with non-uft8 encoding, the character drawing of output will be
# garbled.

# At the same time, this method will save the current quantum circuit
# character drawing information to the file named QCircuitTextPic.txt
# the file is encoded with uft8 and saved under the face path.
# Therefore, users can also view quantum circuit information through
# this file. Note that the file must be opened in uft8 format,
# otherwise garbled characters will appear.

    print(prog)
# Output quantum circuit character drawing through draw_qprog
# interface. The function of this method is the same as print method,
# but the difference is that this interface can specify the console
# encoding type to ensure that the quantum circuit character drawing
# output on the console can be displayed normally.
# The "console_encode_type" parameter is used to specify the console
# type. Currently, two encoding modes are supported: utf8 and gbk. The
# default is utf8

    draw_qprog(prog, 'text', console_encode_type='gbk')
# The draw_qprog interface can also save the quantum circuit as a
# picture, called as follows. The "filename" parameter is used to
# specify a filename to save.

    draw_qprog(prog, 'pic', filename='D:/test_cir_draw.png')

if __name__=="__main__":
    init_machine = InitQMachine(16, 16)
    qlist = init_machine.m_qlist
    clist = init_machine.m_clist
    machine = init_machine.m_machine

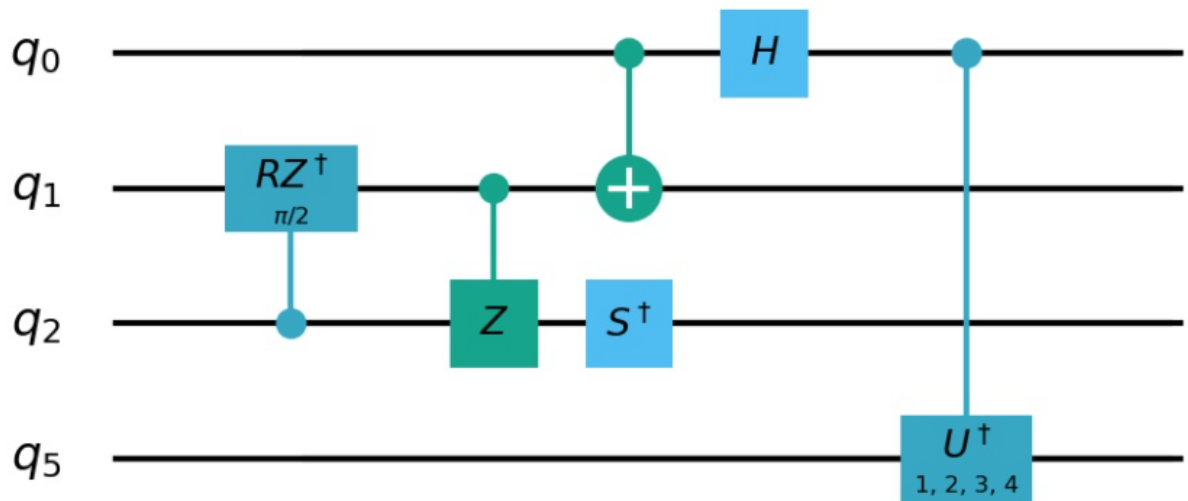
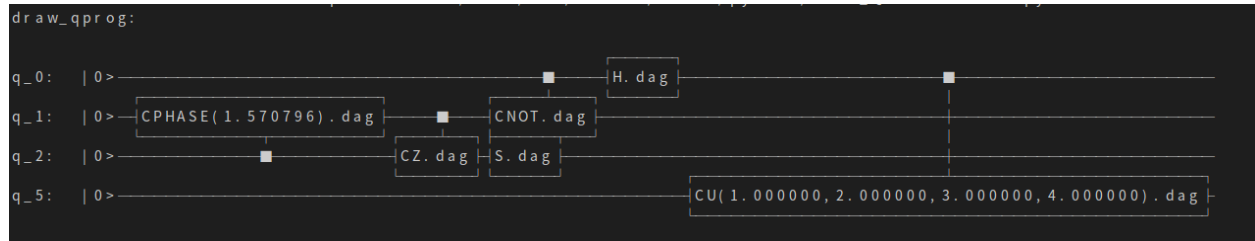
    test_print_qcircuit(qlist, clist)

```

The above example demonstrates the draw\_qprog interface, respectively. The output from the above code is as follows:

The output quantum circuit picture effect is as follows:

The parameters of draw\_qprog() are described as follows:



```

"""Draw a quantum circuit to different formats (set by output parameter):

**text**: ASCII art TextDrawing that can be printed in the console.

**pic**: images with color rendered purely in Python.

Args:
    prog : the quantum circuit to draw
    scale (float): scale of image to draw (shrink if < 1). Only used by the ``pic``
↳outputs.
    filename (str): file path to save image to
    NodeIter_first: circuit printing start position.
    NodeIter_second: circuit printing end position.
    console_encode_type(str): Target console encoding type.
        Mismatching of encoding types may result in character confusion, 'utf8' and
↳'gbk' are supported.
        Only used by the ``pic`` outputs.
    line_length (int): Sets the length of the lines generated by `text` output type.

Returns: no return

"""
def draw_qprog(prog, output=None, scale=0.7, filename=None, line_length=None,
↳NodeIter_first=None, \
NodeIter_second=None, console_encode_type = 'utf8'):

```

As a demonstration, we change the `test_print_qcircuit()` interface implementation from the example code above to the following:

```

prog = pq.QCircuit()
prog << pq.CU(1, 2, 3, 4, q[0], q[5]) << pq.H(q[0]) << pq.S(q[2]) << pq.CNOT(q[0],
↳q[1]) << pq.CZ(q[1], q[2]) << pq.CR(q[2], q[1], math.pi/2)
iter_start = prog.begin()
iter_end = iter_start.get_next()
iter_end = iter_end.get_next()
iter_end = iter_end.get_next()
prog.set_dagger(True)
draw_qprog(prog, 'text', NodeIter_first=iter_start, NodeIter_second=iter_end, console_
↳encode_type='gbk')
draw_qprog(prog, 'pic', NodeIter_first=iter_start, NodeIter_second=iter_end, filename=
↳'D:/test_cir_draw.jpg')

```

The above example code will output only the first 4 logical gate nodes of prog, users can replace the above code to the previous example program, run to view the results, not repeated here.

## 3.10 3.10 Quantum volume

### 3.10.1 3.10.1 Introduction

Quantum volume is a protocol used to evaluate the performance of the quantum computing system. It represents the random circuit with the maximum equal width depth that can be performed on the system. The higher the operating fidelity of the quantum computing system is, the higher the correlation is; the larger the gate operation set calibrated is, the higher the quantum volume is. Quantum volume relates to the overall performance of the system, including the overall error rate of the system, potential physical qubit correlation, and gate operation parallelism. Generally speaking, quantum volume is practical method for making an overall evaluation of the quantum computing system in the near future; the higher the value is, the lower the overall error rate of the system is, the better the performance is.

To measure the quantum volume in a standard way is to perform random circuit operations for the system with specified quantum circuit model, entangle qubits as far as possible, and compare the experimental results with the simulated results. The statistical results are analyzed as required.

Quantum volume is defined in the exponential form:

$$V_Q = 2^n$$

Where n is the maximum logical depth of system operation under the given number of qubits m (m is greater than n) and the completion of computing tasks; if the maximum executive logical depth of the chip n is greater than the number of qubits m, the quantum volume of the system is

$$2^M$$

### 3.10.2 3.10.2 Interface description

The input parameters of `calculate_quantum_volume` are the noise simulator or quantum cloud machine, qubit to be measured, number of random iterations, and number of measurements. The output is `size_t`, which is the quantum volume size.

### 3.10.3 3.10.3 Example

```
from pyqpanda import *

if __name__ == "__main__":
    # Create a noise simulator and set noise parameters
    qvm = NoiseQVM()
    qvm.init_qvm()
    qvm.set_noise_model(NoiseModel.DEPOLARIZING_KRAUS_OPERATOR, GateType.CZ_GATE, 0.
↪ 005)
```

(continues on next page)

(continued from previous page)

```

# You can also apply for cloud computing machines (using real chips), using real chips
# to consider the chip structure
    #qvm = QCloud()
    #qvm.init_qvm("898D47CF515A48CEAA9F2326394B85C6")

# Construct the qubit combination to be measured, where the qubit combination is two
# groups; the qubit 3 and 4 are a group, qubits 2, 3, and 5 are a group.
qubit_lists = [[3,4], [2,3,5]]

    # Set the number of random iterations
    ntrials = 100

    # Set the measurement times, namely real chip or noise simulator shots value
    shots = 2000
    qv_result = calculate_quantum_volume(qvm, qubit_lists, ntrials, shots)
    print("Quantum Volume : ", qv_result)
    qvm.finalize()

```

Running results:

```
Quantum Volume : 8
```

## 3.11 3.11 Random benchmark

### 3.11.1 3.11.1 Introduction

Random benchmark (RB) is a benchmark test for quantum gates with a randomization method. Since the complete process tomography doesn't work for the large system, more attention is paid to the extensible method, to characterize noise that affects the quantum system. An extensible and robust algorithm (a system comprising  $n$  qubits) is put forward in [1], i.e. benchmark test is performed for the whole Clifford gate by a single parameter with the randomizing technique.

### 3.11.2 3.11.2 Interface description

The input parameters of `single_qubit_rb` are the noise simulator or quantum cloud machine, qubit to be measured, a different number of combinations of random circuit clifford gate sets, number of random circuits, number of measurements, verification of basic logic gate (default: none), output being `std::map` data, key value being the number of clifford gate sets, and value corresponding to the expected probability.

The input parameters of `double_qubit_rb` are the noise simulator or quantum cloud machine, qubit 0 to be measured, qubit 1 to be measured, a different number of combinations of random circuit clifford gate sets, number of random circuits,

number of measurements, verification of basic logic gate (default: none), output being `std::map` data, key value being the number of clifford gate sets, and value corresponding to the expected probability.

```
from pyqpanda import *

if __name__=="__main__":
    # Build a noise simulator and adjust the noise to simulate the real chip
    qvm = NoiseQVM()
    qvm.init_qvm()
    qvm.set_noise_model(NoiseModel.DEPOLARIZING_KRAUS_OPERATOR, GateType.CZ_GATE, 0.
↪005)
    qvm.set_noise_model(NoiseModel.DEPOLARIZING_KRAUS_OPERATOR, GateType.PAULI_Y_GATE, \ 0.
↪005)
    qv = qvm.qAlloc_many(4)

    # You can also apply for cloud computing machines (with real chips)
    # qvm = QCloud()
    # qvm.init_qvm("898D47CF515A48CEAA9F2326394B85C6")

    # Set the number of Clifford gates in a random line
    range = [ 5,10,15 ]

    # Measuring a single qubit random reference
    res = single_qubit_rb(qvm, qv[0], range, 10, 1000)

    # It is also possible to measure two-qubit random datum
    #res = double_qubit_rb(qvm, qv[0], qv[1], range, 10, 1000)

    # With the influence of noise, the larger the noise value is, the smaller the result
    # is. And with the increase of Clifford gate set, the smaller the result is.
    print(res)

    qvm.finalize()
```

### 3.11.3 Example

Running results:

```
{5: 0.9996, 10: 0.9999, 15: 0.9993000000000001}
```

## 3.12 3.12 Cross-entropy benchmark

### 3.12.1 3.12.1 Introduction

Cross-entropy benchmark (xeb) is a method used to evaluate the gate performance by applying the random circuits and measuring the cross-entropy between the observed values of bit strings and the expected probability of such bit strings obtained from the simulation.

### 3.12.2 3.12.2 Interface description

The input parameters of `double_gate_xeb` are the noise simulator or quantum cloud machine, qubit 0 to be measured, qubit 1 to be measured, different layers of circuits, number of random circuits, number of measurements, verification of double-gate type (default: CZ gate), output being `std::map` data, key value being the number of circuit layers, and value corresponding to the expected probability.

### 3.12.3 3.12.3 Example

```
from pyqpanda import *

if __name__ == "__main__":

    # Build a noise simulator and adjust the noise to simulate the real chip
    qvm = NoiseQVM()
    qvm.init_qvm()
    qv = qvm.qAlloc_many(4)

    # Setting noise Parameters
    qvm.set_noise_model(NoiseModel.DEPOLARIZING_KRAUS_OPERATOR, GateType.CZ_GATE, 0.1)

    # You can also apply for cloud computing machines (with real chips)
    # qvm = QCloud()
    # qvm.init_qvm("898D47CF515A48CEAA9F2326394B85C6")

    # Set different layer combinations
    range = [2,4,6,8,10]
    # Currently, the main testable dual-gate types are CZ CNOT SWAP ISWAP SQISWAP
    res = double_gate_xeb(qvm, qv[0], qv[1], range, 10, 1000, GateType.CZ_GATE)
    # With the influence of noise, the larger the noise value is, the smaller the result
    # will be; and the number of layer increases, the smaller the result will be.

    print(res)
```

(continues on next page)

(continued from previous page)

```
qvm.finalize()
```

Running results:

```
{2: 0.9922736287117004, 4: 0.9303175806999207, 6: 0.7203856110572815, 8: 0.  
↪ 7342230677604675, 10: 0.7967881560325623}
```



## 4 COMPILING OF QUANTUM PROGRAM

### 4.1 Conversion of QASM by a quantum program

You may parse the quantum program created by QPanda2 with this function module, and extract the qubit information and quantum logic gate operation information contained therein, to obtain the QASM instruction set saved in a fixed form.

#### 4.1.1 QASM introduction

Quantum Assembly Language is put forward by IBM, and is similar to the syntax rules in the QRunes introduction; a QASM code segment is as follows:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[10];
creg c[10];

x q[0];
h q[1];
tdg q[2];
sdg q[2];
cx q[0],q[2];
cx q[1],q[4];
u1(pi) q[0];
u2(pi,pi) q[1];
u3(pi,pi,pi) q[2];
cz q[2],q[5];
ccx q[3],q[4],q[6];
cu3(pi,pi,pi) q[0],q[1];
measure q[2] -> c[2];
measure q[0] -> c[0];
```

It should be noted that the syntax formats of QASM and QRunes are alike but different, with the following differences:

- For quantum logic gates and quantum circuits requiring the transposed conjugate, QRunes needs to place the target between DAGGER and ENDAG-GER sentences; QASM does the conversion directly.
- QRunes supports the control operation of the quantum logic gates and quantum circuits, but QASM doesn't. Before the conversion of QASM by a quantum program, the control operation contained therein will be decomposed.

Refer to [IBM Q Experience quantum cloud platform](#) for more details on the introduction, use, and experience of QASM.

QPanda2 provides the QASM conversion tool interface `convert_qprog_to_qasm`, which is easy to use. Reference can be made to the following example program.

### 4.1.2 Example

The following routine demonstrates the process of converting QASM instruction set by a quantum program through a simple interface call.

```
from pyqpanda import *

if __name__ == "__main__":
    qvm = init_quantum_machine(QMachineType.CPU)
    q = qvm.qAlloc_many(6)
    c = qvm.cAlloc_many(6)
    prog = QProg()
    cir = QCircuit()
    cir << T(q[0]) << S(q[1]) << CNOT(q[1], q[0])
    prog << cir
    prog << X(q[0]) << Y(q[1]) << CU(1.2345, 3, 4, 5, q[5], q[2])\
        << H(q[2]) << RX(q[3], 3.14)\
        << Measure(q[0], c[0])

    qasm = convert_qprog_to_qasm(prog, qvm)
    print(qasm)
    qvm.finalize()
```

The specific steps are as follows:

- Firstly, initialize a quantum simulator object with `init_quantum_machine` in the main program, in order to manage a series of subsequent behaviors
- Then, initialize the number of qubits and classical registers with `qAlloc_many` and `cAlloc_many`.
- Next, call `QProg` to create the quantum program.
- Finally, the interface `convert_qprog_to_qasm` is called to output the QASM instruction set; `finalize()` is used to release system resources

The running results are as follows:

```

OPENQASM 2.0;
include "qelib1.inc";
qreg q[6];
creg c[6];
u3(0,0.78539816339744828,0) q[0];
u3(0,1.5707963267948966,0) q[1];
cx q[1],q[0];
u3(3.1415926535897931,0,3.1415926535897931) q[0];
u3(3.1415926535897931,0,0) q[1];
u3(0,-0.33629632679489674,0) q[5];
u3(0,-0.67259265358979359,0) q[2];
cx q[5],q[2];
u3(0,0.33629632679489674,0) q[2];
cx q[5],q[2];
u3(1.1415926535897929,3.1415926535897931,2.8672963267948974) q[2];
u3(0,1.5707963267948963,0) q[5];
cx q[5],q[2];
u3(1.1415926535897929,-1.1947036732051033,0) q[2];
cx q[5],q[2];
u3(1.5707963267949034,0,-1.3362963267948968) q[2];
u3(3.1400000000000001,-1.5707963267948966,1.5707963267948966) q[3];
measure q[0] -> c[0];

```

## 4.2 4.2 Conversion into a quantum program by QASM

You may parse the QASM text file with this function module, and extract the quantum logic gate operation information therein, to obtain the quantum program that is operable in QPanda 2.

### 4.2.1 4.2.1 QASM introduction

Reference can be made to the [QASM introduction](#) in the module of converting QASM by a quantum program for the writing format specifications and routines of QASM.

QPanda 2 provides the QASM file conversion tool interface `convert_qasm_to_qprog()`, which is easy to use. Reference can be made to the following example program.

## 4.2.2 Example

The following demonstrates the process of converting the quantum program by QASM instruction set through a simple interface call.

```
from pyqpanda import *

if __name__=="__main__":
    machine = init_quantum_machine(QMachineType.CPU)

    # Write QASM files
    f = open('testfile.txt', mode='w', encoding='utf-8')
    f.write("""// test QASM file
OPENQASM 2.0;
include "qelib1.inc";
qreg q[3];
creg c[3];
x q[0];
x q[1];
z q[2];
h q[0];
tdg q[1];
measure q[0] -> c[0];
""")
    f.close()

    # QASM converts quantum programs and returns quantum programs, quantum bits,
    # and classical registers
    prog_trans, qv, cv = convert_qasm_to_qprog("testfile.txt", machine)

    # Quantum program conversion QASM
    qasm = convert_qprog_to_qasm(prog_trans, machine)

    # Print and compare the conversion results
    print(qasm)
    destroy_quantum_machine(machine)
```

The specific steps are as follows:

- Firstly, compile QASM and save it in a designated file
- Then, initialize a quantum simulator object with `init_quantum_machine` in the main program, in order to manage a series of subsequent behaviors
- Next, call the `convert_qasm_to_qprog` interface to convert QASM into a quantum program
- Finally, call the `convert_qprog_to_qasm` interface to convert the quantum program into QASM, decide

whether QASM is converted into the quantum program correctly by comparing the execution results of the quantum program, and release system resources by `destroy_quantum_machine`

The running results are as follows:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[3];
creg c[3];
u3(1.5707963267949037,3.1415926535897931,3.1415926535897931) q[0];
u3(3.1415926535897931,2.3561944901923386,0) q[1];
u3(0,3.1415926535897931,0) q[2];
measure q[0] -> c[0];
```

---

### Note

In the above example, since QASM supports U3 gate, the quantum circuit is optimized when QProg is converted to QASM, and only U3 gate is in the output QASM, which can effectively reduce the quantum circuit depth. For the types that are not supported temporarily, errors may occur during the process of converting QASM into a quantum program.

---

## 4.3 Conversion into Quil by a quantum program

### 4.3.1 Introduction

Quil can directly describe quantum programs and quantum algorithms from a very low level, which is similar to the hardware description language or assembly language in classical computers. Quil basically adopts the design method of “instruction + parameter list”. An example of a simple quantum program is as follows:

```
X 0
Y 1
CNOT 0 1
H 0
RX(-3.141593) 0
MEASURE 1 [0]
```

- The purpose of X is to perform `Pauli-X` gate operations on target qubits. Similar keywords include Y, Z, H, etc.
- The purpose of Y is to perform `Pauli-Y` gate operations on target qubits.
- The purpose of CNOT is to perform CNOT operations on two qubits. The input parameters are the control qubit serial number and target qubit serial number.
- The purpose of H is to perform Hadamard gate operations on target qubits.

● The purpose of MEASURE is to measure target qubits and save the measuring results in a classical register. The input parameters are serial numbers of target qubits and classical registers where the measuring results are saved.

The above is just a small part of Quil instruction set syntax. Reference can be made to [pyQuil](#) for more details.

## 4.3.2 4.3.2 Interface introduction

We create a quantum program with pyqpanda at first:

```
prog = QProg()
prog << X(qubits[0]) << Y(qubits[1])\
    << H(qubits[2]) << RX(qubits[3], 3.14)\
    << Measure(qubits[0], cbits[0])
```

Then ,call the `convert_qprog_to_quil` for the transformation

```
quil = convert_qprog_to_quil(prog, qvm)
```

## 4.3.3 4.3.3 Example

```
from pyqpanda import *

if __name__ == "__main__":
    qvm = init_quantum_machine(QMachineType.CPU)
    qubits = qvm.qAlloc_many(4)
    cbits = qvm.cAlloc_many(4)
    prog = QProg()

    # Building quantum programs
    prog << X(qubits[0]) << Y(qubits[1])\
        << H(qubits[2]) << RX(qubits[3], 3.14)\
        << Measure(qubits[0], cbits[0])

    # The quantum program converts the Quil and prints the Quil
    quil = convert_qprog_to_quil(prog, qvm)
    print(quil)

    qvm.finalize()
```

Running results:

```
X 0
Y 1
```

(continues on next page)

(continued from previous page)

```
H 2
RX(3.140000) 3
MEASURE 0 [0]
```

**Warning:** The new interface `convert_qprog_to_quil()` is provided with the same function as the old one `transform_qprog_to_quil()`.

## 4.4 4.4 Serialization of quantum programs

### 4.4.1 4.4.1 Introduction

A protocol is defined to serialize quantum programs to binary data for the convenience of saving and transmitting quantum programs.

### 4.4.2 4.4.2 Interface introduction

We create a quantum program with pyqpanda at first:

```
prog = QProg()
prog << H(qubits[0]) \
    << CNOT(qubits[0], qubits[1]) \
    << CNOT(qubits[1], qubits[2]) \
    << CNOT(qubits[2], qubits[3])
```

Next, call the `convert_qprog_to_binary` interface for serialization.

```
prog_str = convert_qprog_to_binary(prog, qvm)
```

#### Note

Quantum program serialization is a two-step process that first serializes a quantum program into a binary and then converts the binary into a string encoded in base 64 format.

### 4.4.3 Example

```
from pyqpanda import *
import base64

if __name__ == "__main__":
    qvm = init_quantum_machine(QMachineType.CPU)
    qubits = qvm.qAlloc_many(4)
    cbits = qvm.cAlloc_many(4)

    # Building quantum programs
    prog = QProg()
    prog << H(qubits[0]) \
        << CNOT(qubits[0], qubits[1]) \
        << CNOT(qubits[1], qubits[2]) \
        << CNOT(qubits[2], qubits[3])

    # Quantum program serialization
    binary_data = convert_qprog_to_binary(prog, qvm)

    # The resulting binary data is encoded in Base64 and printed
    str_base64_data = base64.encodebytes(bytes(binary_data))
    print(str_base64_data)

    destroy_quantum_machine(qvm)
```

Running results:

```
b'AAAAAAQAAAAEAAAAABAAAAA4AAQAAAAAJAACAAAAQAKAAMAAQACACQABAACAAMA\n'
```

---

#### Note

The binary data cannot be output directly and should be subject to base64 encoding, to obtain corresponding character strings.

---

**Warning:** The new interface `convert_qprog_to_binary()` is provided with the same function as the old one `transform_qprog_to_binary()`.



## 4.5 Parse quantum program binary files

### 4.5.1 Introduction

Parse quantum programs that are serialized by the quantum program serialization method.

### 4.5.2 Interface introduction

We create a quantum program with pyqpanda at first:

```
prog = QProg()
prog << H(qubits[0]) \
    << CNOT(qubits[0], qubits[1]) \
    << CNOT(qubits[1], qubits[2]) \
    << CNOT(qubits[2], qubits[3])
```

After the serialization and base64 encoding, the result is (refer to the quantum program serialization for specific serialized method).

```
b'AAAAAAQAAAAEAAAABAAAAA4AAQAAAAAJAACAAAAQAKAAMAAQACACQABAACAAMA\n'
```

Now, deserialized this result. Firstly, decode the base64 result into binary data:

```
str_base64_data = b'AAAAAAQAAAAEAAAABAAAAA4AAQAAAAAJAACAAAAQAKAAMAAQACACQABAACAAMA\n'
data = [int(x) for x in bytes(base64.decodebytes(str_base64_data))]
```

We may use one interface encapsulated by QPanda2:

```
convert_binary_data_to_qprog(qvm, data, qubits_parse, cbits_parse, parseProg);
```

### 4.5.3 Example

```
from pyqpanda import *
import base64

if __name__ == "__main__":
    qvm = init_quantum_machine(QMachineType.CPU)

    # Base64 decoding of the way to get binary data
    str_base64_data = \
    ↪ 'AAAAAAQAAAAEAAAABAAAAA4AAQAAAAAJAACAAAAQAKAAMAAQACACQABAACAAMA\n';
    data = [int(x) for x in bytes(base64.decodebytes(str_base64_data))]
```

(continues on next page)

(continued from previous page)

```
# Parse binary data, get quantum programs
parseProg = QProg()
parseProg = convert_binary_data_to_qprog(qvm, data)

# The quantum program converts OriginIR and prints it
print(convert_qprog_to_originir(parseProg, qvm))

destroy_quantum_machine(qvm)
```

Running results:

```
QINIT 4
CREG 4
H q[0]
CNOT q[0],q[1]
CNOT q[1],q[2]
CNOT q[2],q[3]
```

---

### Note

If the running results are correct, serialized quantum programs can be parsed correctly

---

**Warning:** The new interface `convert_binary_data_to_qprog()` is provided with the same function as the old one `transform_binary_data_to_qprog()`.

## 4.6 Conversion into a quantum program by OriginIR

You may parse the OriginIR text file with this function module, and extract the quantum logic gate operation information therein, to obtain the quantum program that is operable in QPanda 2.

### 4.6.1 OriginIR

Reference can be made to the OriginIR introduction in the module of converting OriginIR by a quantum program for the writing format specifications and routines of OriginIR.

QPanda 2 provides the OriginIR file conversion tool interface `convert_originir_to_qprog()`, which is easy to use. Reference can be made to the following example program.

## 4.6.2 4.6.2 Example

The following demonstrates the process of converting the quantum program by OriginIR through a simple interface call.

```
from pyqpanda import *
if __name__=="__main__":
    machine = init_quantum_machine(QMachineType.CPU)

    # Write OriginIR files
    f = open('testfile.txt', mode='w',encoding='utf-8')
    f.write("""QINIT 4
        CREG 4
        DAGGER
        X q[1]
        X q[2]
        CONTROL q[1], q[2]
        RY q[0], (1.047198)
        ENDCONTROL
        ENDDAGGER
        MEASURE q[0], c[0]
        QIF c[0]
        H q[1]
        H q[2]
        RZ q[2], (2.356194)
        CU q[2], q[3], (3.141593, 4.712389, 1.570796, -1.570796)
        CNOT q[2], q[1]
        ENDQIF
        """)

    f.close()

    # OriginIR converts a quantum program and returns the converted quantum
    # program, the quantum bits used by the quantum program, and the classical
    # registers
    prog, qv, cv = convert_originir_to_qprog("testfile.txt", machine)

    # The quantum program converts OriginIR, prints and compares the conversion
    # results
    print(convert_qprog_to_originir(prog,machine))

    destroy_quantum_machine(machine)
```

The specific steps are as follows:

- Firstly, compile OriginIR and save it in a designated file

- Then, initialize a quantum simulator object with `init_quantum_machine` in the main program in order to manage a series of subsequent behaviors
- Next, call `convert_originir_to_qprog()` for the conversion
- Finally, call the `convert_qprog_to_originir()` to convert the quantum program into OriginIR, decide whether OriginIR is converted into the quantum program correctly by comparing whether the QASMs input and generated are the same, and release system resources by `destroy_quantum_machine`

The running results are as follows:

```
QINIT 4
CREG 4
DAGGER
X q[1]
X q[2]
CONTROL q[1],q[2]
RY q[0],(1.047198)
ENCONTROL
ENDDAGGER
MEASURE q[0],c[0]
QIF c[0]
H q[1]
H q[2]
RZ q[2],(2.356194)
CU q[2],q[3],(3.141593,4.712389,1.570796,-1.570796)
CNOT q[2],q[1]
ENDQIF
```

---

### Note

For the types that are not supported temporarily, errors may occur during the process of converting OriginIR into a quantum program.

---

**Warning:** The new interface `convert_originir_to_qprog()` is provided with the same function as the old one `transform_originir_to_QProg()`.

## 4.7 4.7 Conversion of OriginIR by a quantum program

You may parse the quantum program created by pyqpanda with this function module, and extract the qubit information and quantum logic gate operation information contained therein, to obtain the OriginIR saved in a fixed form.

### 4.7.1 4.7.1 OriginIR introduction

OriginIR is an intermediate representation of quantum programs based on QPanda, and plays an important role in supporting properties of QPanda. OriginIR not only indicates most of the quantum logic gate types, indicates dagger operation for quantum circuits, and adds control qubits for quantum circuits, but also supports Qif and QWhile unique to QPanda and implements classical programs where quantum programs are embedded.

OriginIR mainly includes qubit, classical register, quantum logic gate, transposed conjugate operation, adding control qubit operation, QIf, QWhile, and classical expression.

#### 4.7.1.1 4.7.1.1 Qubit

OriginIR applies for qubits with QINIT. The format is QINIT followed by space + the total number of qubits, like QINIT 6. It should be noted that QINIT must be at the first row of the OriginIR program, except notes. When qubits are used, OriginIR uses  $q[i]$  to indicate a certain specific qubit;  $i$  is the No. of the qubit, and can be a unsigned numerical constant or a variable; it can also use an alternate expression comprising  $c[i]$ , like  $q[1]$ ,  $q[c[0]]$ ,  $q[c[1]+c[2]+c[3]]$ .

#### 4.7.1.2 4.7.1.2 Classical register

OriginIR applies for classical registers with CREG. The format is CREG followed by space + the total number of classical registers, like CREG 6; when the classical registers are used, OriginIR uses  $c[i]$  to indicate a certain specific classical register;  $i$  is the No. of the classical register, and must be an unsigned numeric constant, like  $c[1]$ .

#### 4.7.1.3 4.7.1.3 Quantum logic gate

OriginIR divides quantum logic gates into the following categories: keywords of the single-gate without parameters, single-gate with one parameter, single-gate with two parameters, single-gate with three parameters, single-gate with four parameters, double-gate without parameters, double-gate with one parameter, double-gate with four parameters, and tri-gate without parameters. It should be noted that for all single-gate operations, the target qubit can be the entire qubit array or a single qubit. In case of the entire qubit array, for example:

$H \ q$

When the qubit array size is 3, it is equivalent to:

```
H q[0]
H q[1]
H q[2]
```

1.Keywords of the single-gate without parameters: H, T, S, X, Y, Z, X1, Y1, Z1, and I; indicating the single-quantum logic gate without parameters; the format is quantum logic gate keyword + space + target qubit. Example:

```
H q[0]
```

2.Keywords of the single-gate with one parameter: RX, RY, RZ, and U1; indicating the single-quantum logic gate with one parameter; the format is quantum logic gate keyword + space + target qubit + comma + (deflection angle). Example:

```
RX q[0], (1.570796)
```

3.Keywords of the single-gate with two parameters: U2 and RPhi; indicating the single-quantum logic gate with two parameters; the format is quantum logic gate keyword + space + target qubit + comma + (two deflection angles). Example:

```
U2 q[0], (1.570796, -3.141593)
```

4.Keyword of the single-gate with three parameters: U3; indicating the single-quantum logic gate with three parameters; the format is quantum logic gate keyword + space + target qubit + comma + (three deflection angles). Example:

```
U3 q[0], (1.570796, 4.712389, 1.570796)
```

5.Keyword of the single-gate with four parameters: U4; indicating the single-quantum logic gate with four parameters; the format is quantum logic gate keyword + space + target qubit + comma + (four deflection angles). Example:

```
U4 q[1], (3.141593, 4.712389, 1.570796, -3.141593)
```

6.Keywords of the double-gate without parameters: CNOT, CZ, ISWAP, SQISWAP, and SWAP; indicating the double-quantum logic gate without parameters; the format is quantum logic gate keyword + space + control qubit + comma + target qubit. Example:

```
CNOT q[0], q[1]
```

7.Keywords of the double-gate with one parameter: ISWAPTHETA and CR; indicating the single-quantum logic gate with one parameter; the format is quantum logic gate keyword + space + control qubit + comma + target qubit + comma + (deflection angle). Example:

```
CR q[0], q[1], (1.570796)
```

8.Keyword of the double-gate with four parameters: CU; indicating the single-quantum logic gate with four parameters; the format is quantum logic gate keyword + space + control qubit + comma + target qubit + comma + (four deflection angles). Example:

```
CU q[1],q[3],(3.141593,4.712389,1.570796,-3.141593)
```

9.Keyword of the tri-gate without parameters: TOFFOLI; indicating the tri-quantum logic gate without parameters; the format is quantum logic gate keyword + space + control qubit 1 + comma + control qubit 2 + comma + target qubit. Example:

```
TOFFOLI q[0],q[1],q[2]
```

#### 4.7.1.4 4.7.1.4 Transposed conjugate operation

The transposed conjugate operation can be performed for one or more quantum logic gates in OriginIR, which defines the range of transposed conjugate operation with keywords, i.e. DAGGER and ENDDAGGER. One DAGGER must match with one ENDDAGGER; for example:

```
DAGGER
H q[0]
CNOT q[0],q[1]
ENDDAGGER
```

#### 4.7.1.5 4.7.1.5 Adding control qubit operation

The control qubits can be added for one or more quantum logic gates in OriginIR, which defines the range of adding control qubits with keywords, i.e. CONTROL and ENDCONTROL. CONTROL is followed by space + control qubit list; Example:

```
CONTROL q[2],q[3]
H q[0]
CNOT q[0],q[1]
ENDCONTROL
```

#### 4.7.1.6 4.7.1.6 QIF

The judgment programs for quantum conditions can be indicated in OriginIR, which determines the range of different ranches of judgment programs for quantum conditions by QIF, ELSE, and ENDIF. QIF must match with one ENDIF; if QIF has two branches, ELSE is required; if QIF only has one branch, ELSE is not required; QIF is followed by space + judgment expression. Example:

```
1、QIF There's only one conditional branch
QIF c[0]==c[1]
H q[0]
CNOT q[0],q[1]
```

(continues on next page)

(continued from previous page)

```

ENDIF

2、QIF There are two conditional branches
QIF c[0]+c[1]<5
H q[0]
CNOT q[0],q[1]
ELSE
H q[0]
X q[1]
ENDIF

```

#### 4.7.1.7 4.7.1.7 QWHILE

The judgment programs for quantum loops can be indicated in OriginIR, which determines the range of judgment programs for loops by QIF, ELSE, and ENDIF; QWHILE is followed by space + judgment expression. Example:

```

QWHILE c[0]<5
H q[c[0]]
c[0]=c[0]+1
ENDQWHILE

```

#### 4.7.1.8 4.7.1.8 Classical expression

OriginIR can embed a classical expression in the quantum program, like  $c[0] == c[1] + c[2]$ ; Example:

```

QWHILE c[0]<5
H q[c[0]]
c[0]=c[0]+1
ENDQWHILE

```

The expression indicates the H gate operation performed for  $q[0] \sim q[4]$  qubit; the classical expression must be comprised of classical registers and constants; the operators of the classical expression include.

```

{PLUS , "+"},
{MINUS, "-"},
{MUL, "*"},
{DIV, "/"},
{EQUAL, "==" },
{ NE, "!=" },
{ GT, ">" },
{ EGT, ">=" },
{ LT, "<" },

```

(continues on next page)



(continued from previous page)

```
{ ELT, "<=" },
{AND, "&&"},
{OR, "| |"},
{NOT, "!"},
{ASSIGN, "=" }
```

#### 4.7.1.9 4.7.1.9 MEASURE operation

MEASURE indicates the measurement operation performed for designated qubits, and saving the results to the designated classical register. MEASURE is followed by space + target qubit + ‘,’ + target classical register. Example:

```
MEASURE q[0],c[0]
```

If the number of qubits applied is the same as that of the classical registers, q can be used to indicate all qubits, and c indicates all classical bits. Example:

```
MEASURE q,c
```

If the number of qubits and the number of classical bits are 3 respectively, it is equivalent to

```
MEASURE q[0],c[0]
MEASURE q[1],c[1]
MEASURE q[2],c[2]
```

#### 4.7.1.10 4.7.1.10 RESET operation

RESET operation is to reset the quantum state of qubits operated to zero. The format is RESET + space + target qubit where the target qubit can be the entire qubit array or a single qubit. Example:

```
RESET q

RESET q[1]
```

#### 4.7.1.11 4.7.1.11 BARRIER operation

BARRIER operation is to block the qubits operated, so as to prevent the optimization and execution on the circuit. The format is BARRIER + space + target qubit where the target qubit can be the entire qubit array or a single qubit or multiple qubits. Example:

```
BARRIER q
BARRIER q[0]
BARRIER q[0],q[1],q[2]
```

#### 4.7.1.12 4.7.1.12 QGATE operation

QGATE is user-defined logic gate operation, during which multiple logic gates can be combined into a new one. The scope of user-defined logic gates is framed through QGATE and ENDQGATE. It should be noted that the parameter name of the user-defined logic gate can not conflict with any of the related keywords as described above.

The rules for declaration of user-defined logic gates are as below:

```
QGATE UserDefinedGateName BitParameter, (angle)
//UserDefinedGateName, user-defined logical gate name, string
//BitParameter, user-defined logical gate parameter information, string
//angle, angle information, string
// Other relevant information["(", "(" etc.]It must be written in the defined format
// Among them ",", and subsequent information can be blank, the angle information can
↪be null
```

A simplified example is shown below:

```
QGATE new_H a
H a
X a
ENDQGATE

QGATE new_RX a, (b)
RX a, (PI/2+b)
CONTROL q[0]
RX a, (-3.141593)
DAGGER
H a
ENDDAGGER
ENDCONTROL
DAGGER
H a
DAGGER
H a
ENDDAGGER
ENDDAGGER
ENDQGATE
```

The user, after applying for qubits and classical registers, can call user-defined logic gates in the following format:

```
UserDefinedGateName  argue, (angle)
//UserDefinedGateName, user-defined logical gate name, string, be consistent with the
↪definition section above
//BitParameter, user-defined logical gate parameter information, string, It must be
↪q[x], and x needs to be less than the number of qubits requested
//angle, angle information, string, it can be a number, or an expression related to PI
```

A simplified example is shown below:

```
new_H q[0]
new_RX q[1], (PI/4)
```

#### 4.7.1.13 4.7.1.13 OriginIR program example

OPE algorithm

```
QINIT 3
CREG 2
H q[2]
H q[0]
H q[1]
CONTROL q[1]
RX q[2], (-3.141593)
ENCONTROL
CONTROL q[0]
RX q[2], (-3.141593)
RX q[2], (-3.141593)
ENCONTROL
DAGGER
H q[1]
CR q[0], q[1], (1.570796)
H q[0]
ENDDAGGER
MEASURE q[0], c[0]
MEASURE q[1], c[1]
```

QPanda2 provides an OriginIR conversion interface (`std::string convert_qprog_to_originir()`) which is easy to use. See the following example program for details.

## 4.7.2 Example

The following example demonstrates the process of converting OriginIR by a quantum program through a simple interface call.

```
from pyqpanda import *

if __name__ == "__main__":
    machine = init_quantum_machine(QMachineType.CPU)
    qlist = machine.qAlloc_many(4)
    clist = machine.cAlloc_many(4)
    prog = create_empty_qprog()
    prog_cir = create_empty_circuit()

    # Building quantum circuits
    prog_cir << Y(qlist[2]) << H(qlist[2]) << CNOT(qlist[0],qlist[1])

    # Build a QWhile that uses quantum circuitry for loop branching
    qwhile = create_while_prog(clist[1], prog_cir)

    # Build the quantum program and insert QWhile into the quantum program
    prog << H(qlist[2]) << Measure(qlist[1],clist[1]) << qwhile

    # The quantum program converts the QoriginIR and prints the OriginIR
    print(convert_qprog_to_originir(prog,machine))

    destroy_quantum_machine(machine)
```

The specific steps are as follows:

- Firstly, initialize a quantum simulator object with `init_quantum_machine` in the main program in order to manage a series of subsequent behaviors.
- Then, initialize the number of qubits and classical registers with `qAlloc_many` and `cAlloc_many`.
- Next, call `create_empty_qprog` to create the quantum program.
- Finally, Call the `convert_qprog_to_originir` interface to output the OriginIR, string and use `destroy_quantum_machine` to release system resources.

The running results are as follows:

```
QINIT 4
CREG 4
H q[2]
MEASURE q[1],c[1]
QWHILE c[1]
```

(continues on next page)

(continued from previous page)

```
Y q[2]
H q[2]
CNOT q[0],q[1]
ENDQWHILE
```

**Note**

For operations which are not currently supported, OriginIR will display Unsupported XXXNode where XXX indicates the type of nodes.

**Warning:** The new interface `convert_qprog_to_originir()` is provided with the same function as the old one `transform_qprog_to_originIR()`.

## 4.8 4.8 Quantum program matching topology

A quantum computing device has finite connections between qubits, allowing only two qubit gates to be applied to finite pairs of qubits. When the quantum program is applied to the target device, the original quantum program must be converted to adapt to the hardware restriction to allow the two qubits in the double qubit gates in compliance with the physical topology, thus to ensure normal operation of the double qubit gates. Most of the existing solutions require inserting additional SWAP operations between two qubits failing to interact in order to “move” the logic qubits to a position where they can interact. This solution is called quantum program matching topology.

### 4.8.1 4.8.1 Interface description

Two matching topology methods are available in the current version:

Interface `topology_match`

By adopting circuit layering and A\* search algorithms, the number of inserted SWAP operations is approximately minimized in the process of matching thus to minimize the overall approximate consumption of the algorithms. The interface requires input of 5 parameters: the quantum program constructed, the set of qubits used, the simulator pointer initialized, the mode of SWAP operation used, and the type of topology. And also the interface can return to the mapped quantum program.

## 4.8.2 4.8.2 Example

```
from pyqpanda import *

if __name__=="__main__":
    qvm = CPUQVM()
    qvm.init_qvm()
    qv = qvm.qAlloc_many(16)
    c = qvm.cAlloc_many(16)
    src_prog = QProg()
    # Building quantum programs
    src_prog << CNOT(qv[0], qv[3]) \
        << CNOT(qv[0], qv[2]) \
        << CNOT(qv[1], qv[3]) \
        << CZ(qv[1], qv[2]) \
        << CZ(qv[0], qv[2]) \
        << T(qv[1]) \
        << S(qv[2]) \
        << H(qv[3])

    # The probability measurement of src_prog results in results_1
    qvm.directly_run(src_prog)
    results_1 = qvm.pmeasure_no_index(qv)

# The quantum program out_prog matching IBM_QX5_ARCH topology is obtained
# by topology matching src_prog
    out_prog, out_qv = topology_match(src_prog, qv, qvm, CNOT_GATE_METHOD, \ IBM_QX5_
    ↪ARCH)

    # The probability of out_prog is measured and the result is results_2
    qvm.directly_run(out_prog)
    results_2 = qvm.pmeasure_no_index(out_qv)

# Compare the probability measurement results_1 and Results_2, and print
# the same result
    len = min(len(results_1), len(results_2))
    for index in range(len):
        if abs(results_1[index] - results_2[index]) < 1.0e-6 :
            print(results_1[index])
        else:
            print("The results are different")
    destroy_quantum_machine(qvm)
```

The specific steps are as follows:

- Firstly, create a quantum simulator, a quantum register and a classical register.
- Then, write the quantum program `src_prog`, and measure the probability of the program to obtain the `result_1`.
- Next, call `topology_match()` to map `src_prog` to a circuit in compliance with the specific physical structure thus to get the quantum program `out_prog` matching with the specific physical structure.
- Finally, measure the probability of the program `out_prog` to obtain the `result_2`.
- The mapping of quantum program, as an additional SWAP operation to the original circuit to adapt to the physical topology, does not affect the structure of the circuit. Therefore, the mapping of the circuit is correct if `result_1` and `result_2` are the same.

The results are as below:

```
0.499999999999999645
The results are different
0.0
0.0
0.0
0.0
0.0
0.0
The results are different
0.0
0.0
0.0
0.0
0.0
0.0
0.0
```





## 5 UTILITY TOOL

### 5.1 Quantum circuit query and replacement

In quantum computing, there are some quantum logic gates or quantum circuits which are interchangeable, including the substitution process below:

$H(0) \rightarrow CNOT(1,0) \rightarrow H(0)$  can be substituted by  $CZ(1,0)$ .

#### 5.1.1 Introduction to interface used

There may be multiple sub-quantum circuits with the same structure or multiple quantum logic gates with the same structure in the quantum program. The function of querying and substituting the quantum circuits with the specified structure in the quantum program is to find these sub-quantum circuits with the same structure and substitute them with the target quantum circuits.

The `circuit_optimizer` provides a unified quantum circuit optimization interface, which can realize query and replacement of various quantum circuits. The corresponding interface parameters are as follows: Parameter 1: QProg parameter 2 of the original quantum program to be optimized: Vector sublines query replacement queues, and each queue element contains the target search line and the corresponding replacement line.

**Warning:** The `graph_query_replace` interface is deprecated.

#### 5.1.2 Example

```
from pyqpanda import *
if __name__ == "__main__":
    machine = init_quantum_machine(QMachineType.CPU)
    q = machine.qAlloc_many(4)
    c = machine.cAlloc_many(4)
    # Building quantum programs
```

(continues on next page)

(continued from previous page)

```

prog = QProg()
prog << H(q[0])\
    << H(q[2])\
    << H(q[3])\
    << CNOT(q[1], q[0])\
    << H(q[0])\
    << CNOT(q[1], q[2])\
    << H(q[2])\
    << CNOT(q[2], q[3])\
    << H(q[3])

# Build a query line
query_cir = QCircuit()
query_cir << H(q[0])\
    << CNOT(q[1], q[0])\
    << H(q[0])

# Build alternate lines
replace_cir = QCircuit()
replace_cir << CZ(q[1], q[0])
print("Query before replacement: ")
print(convert_qprog_to_originir(prog, machine))
# Search the query line in the quantum program and replace it with a
# replacement line
update_prog = circuit_optimizer(prog, [[query_cir, replace_cir]])
print("Query after replacement: ")
print(convert_qprog_to_originir(update_prog, machine))

```

The running results are as follows:

Query before replacement:

```

QINIT 4
CREG 4
H q[0]
H q[2]
H q[3]
CNOT q[1],q[0]
H q[0]
CNOT q[1],q[2]
H q[2]
CNOT q[2],q[3]
H q[3]

```

Query after replacement:

```

QINIT 4
CREG 4
CZ q[1],q[0]
CZ q[1],q[2]
CZ q[2],q[3]

```

**Warning:**

1. The qubits controlled by the queried quantum circuit and the substituted quantum circuit must be one-to-one corresponding.
2. The directed acyclic graph corresponding to the queried quantum circuit and the substituted quantum circuit must be connected.

## 5.2 5.2 Filling QProg by gate I

The interface `fill_qprog_by_I` realizes the function of filling QProg (quantum program) by I gate.

### 5.2.1 5.2.1 Example

```

import pyqpanda.pyQPanda as pq
import math
from pyqpanda.Visualization.circuit_draw import *
import numpy as np
class InitQMachine:
    def __init__(self, quBitCnt, cBitCnt, machineType = pq.QMachineType.CPU):
        self.m_machine = pq.init_quantum_machine(machineType)
        self.m_qlist = self.m_machine.qAlloc_many(quBitCnt)
        self.m_clist = self.m_machine.cAlloc_many(cBitCnt)
    def __del__(self):
        pq.destroy_quantum_machine(self.m_machine)
def test_fill_I(q, c):
    # Building quantum programs
    prog = pq.QCircuit()
    prog << pq.CU(1, 2, 3, 4, q[0], q[5]) << pq.H(q[0]) << pq.S(q[2])\
    << pq.CNOT(q[0], q[1]) << pq.CZ(q[1], q[2])\
    << pq.CR(q[2], q[1], math.pi/2)
    prog.set_dagger(True)
    # Output the original quantum program
    print('source prog:')
    draw_qprog(prog, 'text', console_encode_type='gbk')

```

(continues on next page)

(continued from previous page)

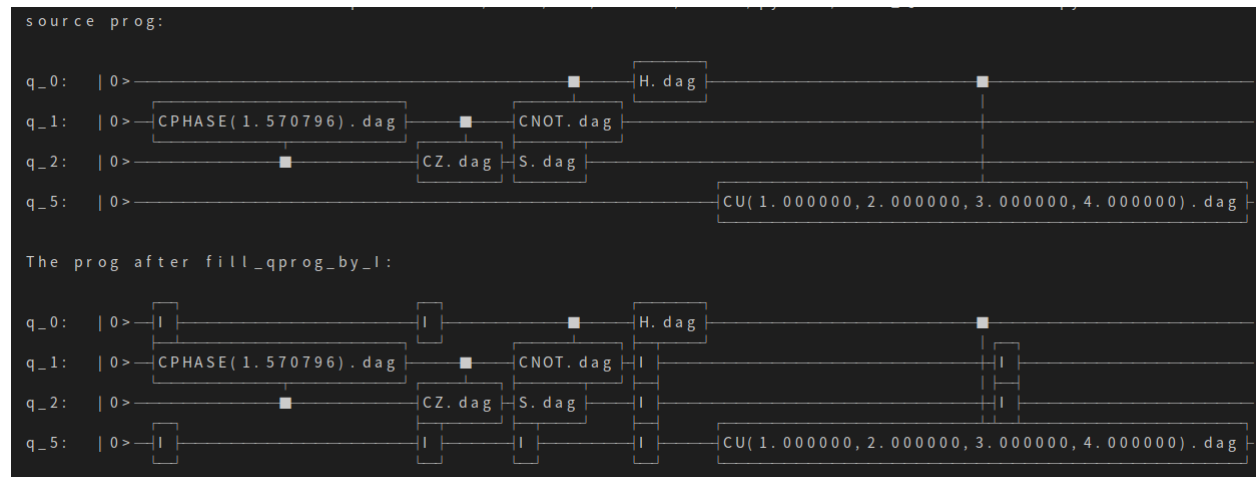
```

"""
console_encode_type='utf8' or 'gbk'(默认 'utf8')
"""

# Quantum program filling I gate
prog = pq.fill_qprog_by_I(prog)
# Output fill I gate quantum program
print('The prog after fill_qprog_by_I:')
draw_qprog(prog, 'text', console_encode_type='gbk')
draw_qprog(prog, 'pic', filename='D:/test_cir_draw.png')
if __name__ == "__main__":
    init_machine = InitQMachine(16, 16)
    qlist = init_machine.m_qlist
    clist = init_machine.m_clist
    machine = init_machine.m_machine
    test_fill_I(qlist, clist)

```

The above example program demonstrates how to use the `fill_qprog_by_I` interface. We can see that we only need input a parameter of QProg type. The interface returns a new filled QProg, and the input QProg remains unchanged. The character drawing of the example program above shows the output results as below:



## 5.3 Unitary matrix decomposition

Currently, a quantum computing algorithm is usually represented by a quantum circuit which includes quantum logic gate operations. A continuous quantum circuit generally includes dozens or hundreds or even thousands of quantum logic gate operations. The larger the number of quantum logic gates or the number of qubits operated by a single quantum logic gate, the more complex the computing process is, thus resulting in low simulation efficiency of quantum circuits and great occupancy of hardware resources.

### 5.3.1 Objective of algorithm

For the above problem, equivalent transformation is necessary for the quantum circuit and the number of logic gates in the quantum circuit shall be reduced. Meanwhile, on this basis, we shall ensure that the unitary matrix corresponding to the whole quantum circuit before the transformation is exactly the same as that after the transformation.

### 5.3.2 Overview of algorithm

The algorithm introduced herein is to decompose a unitary matrix of order  $N$  into no more than  $r = N(N-1)/2$  single-quantum logic gate sequences with a few controls where  $N=2^n$ , and the decomposed products satisfy the equation relations below:

$$U_r U_{r-1} \dots U_3 U_2 U_1 U = I_N$$

Thus, we can obtain the decomposition result of the original matrix  $U$

$$U = U_1^\dagger U_2^\dagger U_2^\dagger \dots U_{r-1}^\dagger U_r^\dagger$$

### 5.3.3 Instructions

The pyqpanda is designed with `matrix_decompose` interface which is used for unitary matrix decomposition and requires two parameters: the first parameter is all the qubits used and the second is the unitary matrix to be decomposed. The output of this function is the quantum circuit after transformation.

### 5.3.4 Example

The following example shows how to use the partial-amplitude quantum simulator.

```
import pyqpanda as pq
import numpy as np

if __name__ == "__main__":

    machine = pq.init_quantum_machine(pq.QMachineType.CPU)
    q = machine.qAlloc_many(2)
    c = machine.cAlloc_many(2)

source_matrix = [(0.6477054522122977+0.1195417767870219j), \
(-0.16162176706189357-0.4020495632468249j), \
(-0.19991615329121998-0.3764618308248643j), \
(-0.2599957197928922-0.35935248873007863j), \
(-0.16162176706189363-0.40204956324682495j), (0.7303014482204584-
↪0.4215172444390785j), \
```

(continues on next page)

(continued from previous page)

```

(-0.15199187936216693+0.09733585496768032j),\
(-0.22248203136345918-0.1383600597660744j),
        (-0.19991615329122003-0.3764618308248644j),\
(-0.15199187936216688+0.09733585496768032j),\
(0.6826630277354306-0.37517063774206166j),\
(-0.3078966462928956-0.2900897445133085j),
        (-0.2599957197928923-0.3593524887300787j),\
(-0.22248203136345912-0.1383600597660744j),\
(-0.30789664629289554-0.2900897445133085j),\
(0.6640994547408099-0.338593803336005j)]

print("source matrix : ")
print(source_matrix)

out_cir = pq.matrix_decompose(q, source_matrix)
circuit_matrix = pq.get_matrix(out_cir)

print("the decomposed matrix : ")
print(circuit_matrix)

source_matrix = np.round(np.array(source_matrix),3)
circuit_matrix = np.round(np.array(circuit_matrix),3)

if np.all(source_matrix == circuit_matrix):
    print('matrix decompose ok !')
else:
    print('matrix decompose false !')

```

The results are as below:

```

source matrix :
[(0.6477054522122977+0.1195417767870219j), (-0.16162176706189357-0.4020495632468249j),
(-0.19991615329121998-0.3764618308248643j), (-0.2599957197928922-0.
↪35935248873007863j),
(-0.16162176706189363-0.40204956324682495j), (0.7303014482204584-0.4215172444390785j),
(-0.15199187936216693+0.09733585496768032j), (-0.22248203136345918-0.
↪1383600597660744j),
(-0.19991615329122003-0.3764618308248644j), (-0.15199187936216688+0.
↪09733585496768032j),
(0.6826630277354306-0.37517063774206166j), (-0.3078966462928956-0.2900897445133085j),
(-0.2599957197928923-0.3593524887300787j), (-0.22248203136345912-0.1383600597660744j),
(-0.30789664629289554-0.2900897445133085j), (0.6640994547408099-0.338593803336005j)]

the decomposed matrix :

```

(continues on next page)

(continued from previous page)

```
[ (0.6477054522122979+0.11954177678702192j), (-0.16162176706189357-0.402049563246825j),
(-0.19991615329122003-0.37646183082486445j), (-0.2599957197928924-0.
↪3593524887300788j),
(-0.16162176706189368-0.40204956324682506j), (0.7303014482204584-0.4215172444390785j),
(-0.1519918793621669+0.09733585496768038j), (-0.22248203136345918-0.
↪13836005976607446j),
(-0.19991615329122003-0.3764618308248644j), (-0.151991879362167+0.09733585496768042j),
(0.6826630277354307-0.37517063774206155j), (-0.30789664629289576-0.2900897445133086j),
(-0.2599957197928924-0.35935248873007875j), (-0.22248203136345918-0.
↪13836005976607443j),
(-0.30789664629289576-0.2900897445133086j), (0.6640994547408103-0.3385938033360052j) ]

matrix decompose ok !
```

Based on the output results, the matrix before transformation is exactly the same as that after transformation. For a quantum system where the number of qubits is determined, the interface can control the complexity of the decomposed quantum circuit within a reasonable range as not affected by the complexity of the pre-decomposed quantum circuit despite that the pre-decomposed quantum circuit contains thousands of quantum logic gates.

---

#### Note

1. The input parameter of the interface must be a unitary matrix.
  2. Limiting the number of decomposition results to a limited range effectively reduces the number of quantum logic gates in the quantum circuit and significantly improves the simulation efficiency of the quantum algorithm.
  3. In the example program, the `get_matrix` interface is used to acquire the corresponding matrix of a quantum circuit.
-








## 6 COMPONENT

## 6.1 6.1 Class of Pauli operators

The Pauli operators are a set of three  $2 \times 2$  complex matrices which are Hermitian and unitary, also called unitary matrices. The matrices are usually indicated by the Greek letter  $\sigma$  (sigma), denoted as

	$\sigma_x$	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
	$\sigma_y$	$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
	$\sigma_z$	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

The operational rules of Pauli operators are as below:

1. The Pauli operators are multiplied by themselves to get an identity matrix.

$$\sigma_x \sigma_x = I$$

$$\sigma_y \sigma_y = I$$

$$\sigma_z \sigma_z = I$$

2. The Pauli operators remain unchanged when multiplied (either pre-multiplied or post-multiplied) by the identity matrix.

$$\sigma_x I = I \sigma_x = \sigma_x$$

$$\sigma_y I = I \sigma_y = \sigma_y$$

$$\sigma_z I = I \sigma_z = \sigma_z$$

3. Two Pauli operators are multiplied in sequence to be  $i$  times as large as the operators not involved in the computing.

$$\sigma_x \sigma_y = i \sigma_z$$

$$\sigma_y \sigma_z = i \sigma_x$$

$$\sigma_z \sigma_x = i \sigma_y$$

4. Two Pauli operators are multiplied in inverted sequence to be  $-i$  times as large as the operators not involved in the computing.

$$\sigma_y \sigma_x = -i \sigma_z$$

$$\sigma_z \sigma_y = -i \sigma_x$$

$$\sigma_x \sigma_z = -i \sigma_y$$

### 6.1.1 Interface introduction

According to the above properties of the Pauli operators, we implement `PauliOperator` in pyQPanda. We can construct the class of Pauli operators easily, for example

```
from pyqpanda import *

if __name__=="__main__":
    # Construct an empty PauliOperator class
    p1 = PauliOperator()

    # Two times the tensor product of Pauli Z0 Pauli Z1
    p2 = PauliOperator("Z0 Z1", 2)

    #2 times the Pauli Z0 tensor product Pauli Z1 plus 3 times the Pauli X1
    # tensor product Pauli Y2
    p3 = PauliOperator({"Z0 Z1": 2, "X1 Y2": 3})

    # Construct an identity matrix with coefficients of 2, equivalent to p4
    # = PauliOperator("", 2)
    p4 = PauliOperator(2)
```

Where `PauliOperator p2( "Z0 Z1" , 2)` represents

$$2\sigma_0^Z \otimes \sigma_1^Z$$

Pauli operators can be added, subtracted, multiplied or subjected to other operations, and the computed return result still belongs to the class of Pauli operators.

```
a = PauliOperator("Z0 Z1", 2)
b = PauliOperator("X5 Y6", 3)

plus = a + b
minus = a - b
multiply = a * b
```

The class of Pauli operators supports printing, and we can print the class of Pauli operators on the screen to view their values.

```
a = PauliOperator("Z0 Z1", 2)
print(a)
```

In actual applications, we often need to know how many qubits are operated by the class of Pauli operators, which can be obtained by calling the getMaxIndex interface of the class of Pauli operators. If an empty class of Pauli operators calls the getMaxIndex interface, the result returned will be 0. Otherwise, the result will be the maximum subscript index value plus 1.

```
a = PauliOperator("Z0 Z1", 2)
b = PauliOperator("X5 Y6", 3)
# The output value is 2
print(a.getMaxIndex())
# The output value is 7
print(b.getMaxIndex())
```

We construct a class of Pauli operators where the subscript index of the Pauli operator is not allocated from 0. For example, the number of qubits returned and used by the getMaxIndex interface called by PauliOperator b( "X5 Y6" , 3) is 7, but actually only 2 qubits are used. How can we return the number of qubits actually used? We can call the remapQubitIndex interface in the class of Pauli operators which is used to allocate and map the indexes in the Pauli operator from 0 qubit and returns the new class of Pauli operator. This interface requires input of a map to save the mapping relationship between the subscript before saving the that after saving.

```
b = PauliOperator("X5 Y6", 3)
index_map = []
c = b.remapQubitIndex(index_map)

# The output value is 7
print(b.getMaxIndex())
# The output value is 2
print(a.getMaxIndex())
```

## 6.1.2 6.1.2 Example

The following example mainly demonstrate how to use the `PauliOperator` interface.

```
from pyqpanda import *
if __name__=="__main__":
    a = PauliOperator("Z0 Z1", 2)
    b = PauliOperator("X5 Y6", 3)
    plus = a + b
    minus = a - b
    multiply = a * b
    print("a + b = ", plus)
    print("a - b = ", minus)
    print("a * b = ", multiply)
    print("Index : ", multiply.getMaxIndex())
    index_map = {}
    remap_pauli = multiply.remapQubitIndex(index_map)
    print("remap_pauli : ", remap_pauli)
    print("Index : ", remap_pauli.getMaxIndex())
```

The results are as below:

```
a + b = {
X5 Y6 : 3.000000
Z0 Z1 : 2.000000
}
a - b = {
X5 Y6 : -3.000000
Z0 Z1 : 2.000000
}
a * b = {
Z0 Z1 X5 Y6 : 6.000000
}
Index : 7
remap_pauli : {
Z0 Z1 X2 Y3 : 6.000000
}
Index : 4
```

## 6.2 Class of Fermionic operators

We use the following notations to indicate the two forms of fermions, annihilation:  $X$  denotes  $a_x$ , creation  $X_+$  denotes  $a_x^\dagger$ . For example, “1+ 3 5+ 1” denotes  $a_1^\dagger a_3 a_5^\dagger a_1$ .

The rules are organized as below:

### 1. Different numbers

$$\begin{aligned} "1 \ 2" &= -1 * "2 \ 1" \\ "1 + 2 + " &= -1 * "2 + 1 + " \\ "1 + 2" &= -1 * "2 \ 1 + " \end{aligned}$$

### 2. Same number

$$\begin{aligned} "1 \ 1 + " &= 1 - "1 + 1" \\ "1 + 1 + " &= 0 \\ "1 \ 1" &= 0 \end{aligned}$$

Similar with `PauliOperator`, the class of `FermionOperator` provides basic operations of Fermionic operators including addition, subtraction and multiplication. An ordered list of results can be obtained through organization.

### 6.2.1 Example

```
from pyqpanda import *

if __name__=="__main__":

    a = FermionOperator("0 1+", 2)
    b = FermionOperator("2+ 3", 3)

    plus = a + b
    minus = a - b
    multiply = a * b

    print("a + b = ", plus)
    print("a - b = ", minus)
    print("a * b = ", multiply)

    print("normal_ordered(a + b) = ", plus.normal_ordered())
    print("normal_ordered(a - b) = ", minus.normal_ordered())
    print("normal_ordered(a * b) = ", multiply.normal_ordered())
```

The results are as below:

```
a + b = {
0 1+ : 2.000000
2+ 3 : 3.000000
}
a - b = {
0 1+ : 2.000000
2+ 3 : -3.000000
}
a * b = {
0 1+ 2+ 3 : 6.000000
}
normal_ordered(a + b) = {
1+ 0 : -2.000000
2+ 3 : 3.000000
}
normal_ordered(a - b) = {
1+ 0 : -2.000000
2+ 3 : -3.000000
}
normal_ordered(a * b) = {
2+ 1+ 3 0 : 6.000000
}
```

## 6.3 Optimization algorithm (direct search)

This chapter will explain the application of optimization algorithms, including the Nelder-Mead algorithm and the Powell algorithm which are direct search ones. In QPanda, we have implemented `OriginNelderMead` and `OriginPowell` which are both inherited from `AbstractOptimizer`.

### 6.3.1 Interface introduction

An optimizer of a specified type can be generated through the optimizer factory. For instance, the type of the optimizer can be specified as Nelder-mead.

```
optimizer = OptimizerFactory.makeOptimizer(OptimizerType.NELDER_MEAD)
#optimizer = OptimizerFactory.makeOptimizer('NELDER_MEAD')
```

We need to register with the optimizer a function used to compute the loss value and the parameters to be optimized.

```
init_para = [0, 0]
optimizer.registerFunc(lossFunc, init_para)
```

Then, we can set the ending condition. Moreover, we can set the convergence thresholds of variables and function values, the maximum number of function calls and the number of iterations in an optimization. The optimization ends as long as the above conditions are met.

```
optimizer.setXatol(1e-6)
optimizer.setFatol(1e-6)
optimizer.setMaxFCalls(200)
optimizer.setMaxIter(200)
```

Then, we can conduct optimization through the exec interface, and obtain the optimized results through the getResult interface.

```
optimizer.exec()

result = optimizer.getResult()
print(result.message)
print(" Current function value: ", result.fun_val)
print(" Iterations: ", result.iters)
print(" Function evaluations: ", result.fcalls)
print(" Optimized para: W: ", result.para[0], " b: ", result.para[1])
```

### 6.3.2 Example

Given with some hash points, we fit a line where the sum of the distances between the hash points and the line is minimized. The expression of the function which defines the line is  $y = w \cdot x + b$ . Next, we will get the optimization values of  $w$  and  $b$  by using the optimization algorithm. First, we define the function to obtain expectation.

```
from pyqpanda import *
import numpy as np

x = np.array([3.3, 4.4, 5.5, 6.71, 6.93, 4.168, 9.779, 6.182, 7.59,\
              2.167, 7.042, 10.791, 5.313, 7.997, 5.654, 9.27, 3.1])
y = np.array([1.7, 2.76, 2.09, 3.19, 1.694, 1.573, 3.366, 2.596, 2.53,\
              1.221, 2.827, 3.465, 1.65, 2.904, 2.42, 2.94, 1.3])

def lossFunc(para, grad, inter, fcall):
    y_ = np.zeros(len(y))

    for i in range(len(y)):
        y_[i] = para[0] * x[i] + para[1]

    loss = 0
    for i in range(len(y)):
        loss += (y_[i] - y[i])**2/len(y)
```

(continues on next page)

(continued from previous page)

```
return (" ", loss)
```

We use the Nelder–Mead algorithm for optimization.

```
optimizer = OptimizerFactory.makeOptimizer('NELDER_MEAD')

init_para = [0, 0]
optimizer.registerFunc(lossFunc, init_para)
optimizer.setXatol(1e-6)
optimizer.setFatol(1e-6)
optimizer.setMaxIter(200)
optimizer.exec()

result = optimizer.getResult()
print(result.message)
print(" Current function value: ", result.fun_val)
print(" Iterations: ", result.iters)
print(" Function evaluations: ", result.fcalls)
print(" Optimized para: W: ", result.para[0], " b: ", result.para[1])
```

```
Optimization terminated successfully.
Current function value: 0.1538576740419577
Iterations: 84
Function evaluations: 161
Optimized para: W: 0.2516349997921341  b: 0.7988010530189995
```

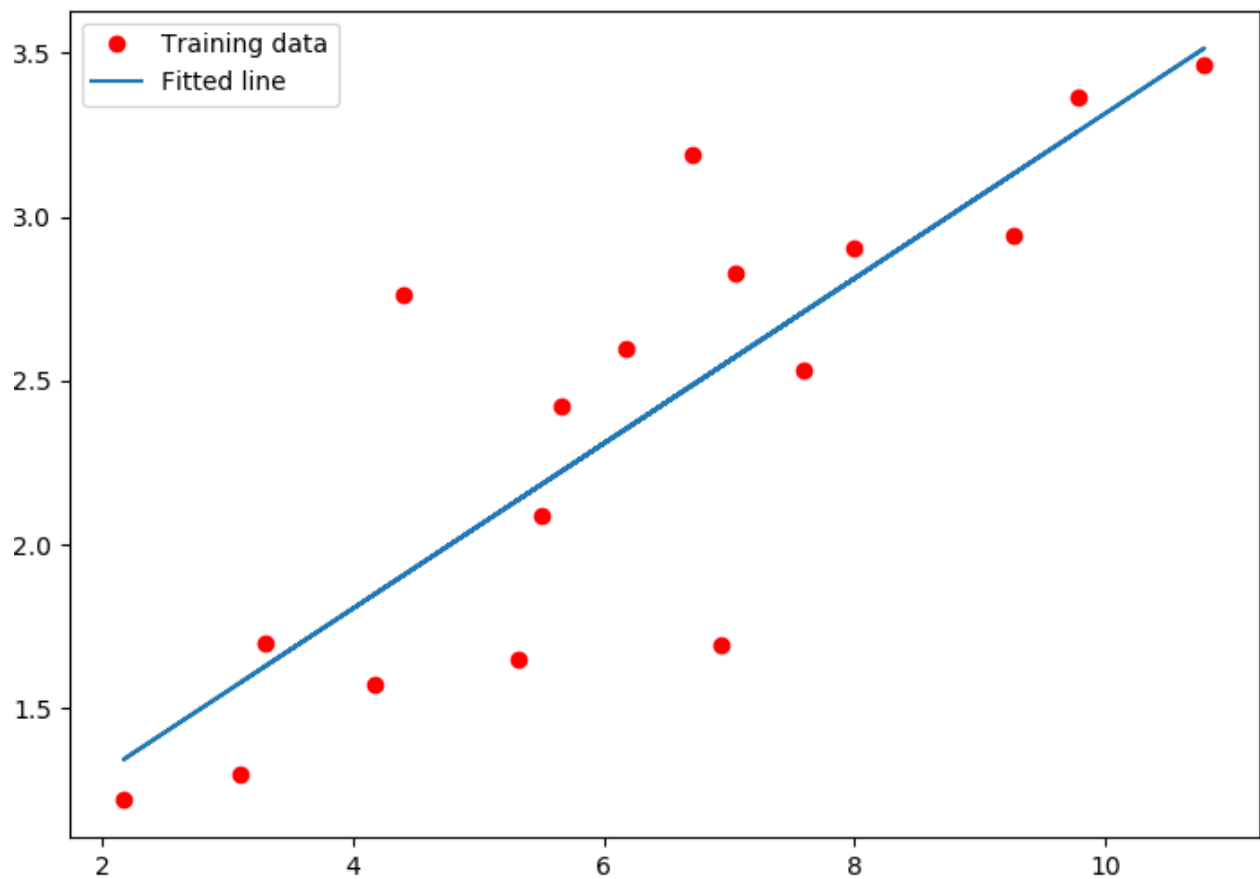
We plot a graph with the hash points and the fitted lines.

```
import matplotlib.pyplot as plt

w = result.para[0]
b = result.para[1]

plt.plot(x, y, 'o', label = 'Training data')
plt.plot(x, w*x + b, 'r', label = 'Fitted line')
plt.legend()
plt.show()
```







## 7.1 7.1 Variable

The class of variables is the user class to implement symbolic computing and used to store the variables of the specific hybrid quantum classical network. Generally, its task is to optimize variables to minimize the cost function. A variable can be a scalar, vector or matrix.

A variable is provided with a tree structure, including child or parent nodes. The variable which has no child nodes is called a leaf node. On one hand, we can set a variable to a specific value. On the other hand, we can get the value of a variable if the values of all of its leaf nodes have been set.

### 7.1.1 7.1.1 Interface introduction

We can construct a scalar variable by inputting floating-point data or construct a vector or matrix variable by inputting a multi-dimensional array generated by the numpy library.

```
v1 = var(1)

a = np.array([[1.],[2.],[3.],[4.]])
v2 = var(a)
b = np.array([[1.,2.],[3.,4.]])
v3 = var(b)
```

---

#### Note

When defining a variable, we can define whether the type of the variable is differentiable, and by default the type is non-differentiable. A non-differentiable variable is the same as a `placeholder`. When defining a differentiable variable, we should the second argument to the constructor as `True` such as `v1 = var(1, True)`.

---

We can define and compute the corresponding expression which is composed of such operations as addition, subtraction, multiplication and division between the variables. Also, the expression is a variable.

```
v1 = var(10)
v2 = var(5)

add = v1 + v2
minus = v1 - v2
multiply = v1 * v2
divide = v1 / v2
```

We can change the value of a certain variable through the `set_value` interface to get different results, without changing the structure of the expression. We can call the `eval` interface to compute the current value of the variable.

```
v1 = var(1)
v2 = var(2)

add = v1 + v2
print(eval(add)) # The output is [[3.]]

v1.set_value([[3.]])
print(eval(add)) # The output is [[5.]]
```

---

### Note

The `get_value` interface of any variable returns the current value of the variable and does not compute the current value based on its leaf nodes. If the variable is an expression of which the value should be computed based on its leaf nodes, the `eval` interface should be called for forward computing.

---

## 7.1.2 Example

We will show the applications of the interfaces related to the class of variables through more examples.

```
from pyqpanda import *
import numpy as np

if __name__=="__main__":

    m1 = np.array([[1., 2.],[3., 4.]])
    v1 = var(m1)

    m2 = np.array([[5., 6.],[7., 8.]])
    v2 = var(m2)

    sum = v1 + v2
```

(continues on next page)

(continued from previous page)

```

minus = v1 - v2
multiply = v1 * v2

print("v1: ", v1.get_value())
print("v2: ", v2.get_value())
print("sum: " , eval(sum))
print("minus: " , eval(minus))
print("multiply: " , eval(multiply))

m3 = np.array([[4., 3.],[2., 1.]])
v1.set_value(m3)

print("sum: " , eval(sum))
print("minus: " , eval(minus))
print("multiply: " , eval(multiply))

```

```

v1:  [[1. 2.]
      [3. 4.]]
v2:  [[5. 6.]
      [7. 8.]]
sum:  [[ 6.  8.]
      [10. 12.]]
minus: [[-4. -4.]
        [-4. -4.]]
multiply: [[ 5. 12.]
           [21. 32.]]
sum:  [[9. 9.]
      [9. 9.]]
minus: [[-1. -3.]
        [-5. -7.]]
multiply: [[20. 18.]
           [14.  8.]]

```

## 7.2 7.2 Operator

VQNet contains many types of operators which can manipulate variables or placeholders. Some of the operators are classical ones, such as addition, subtraction, multiplication, division, exponent, logarithm and point multiplication. In addition, VQNet contains all the common operators from classic machine learning.

And VQNet contains quantum operators which are qop and qop pmeasure. What is more, qop and qop pmeasure are related to quantum computer chips, and they can only be used in a quantum environment. For this, we need to provide additional parameters when using the two operators. Generally, we need to add the qubits referenced to and applied for

by quantum machines to create the quantum environment.

VQNet defines the operators as shown below, all of which return variables of `var` type.

The op- er- a- tor	Describe
<b>plus</b>	Plus. Example: $a + b$ , where $a$ and $b$ are variables of type <b>var</b> .
<b>mi- nus</b>	Minus. Example: $a - b$ .
<b>mul- ti- ply</b>	Multiply. Example: $a * b$ .
<b>di- vide</b>	Divide. Example: $a / b$ .
<b>ex- po- nent</b>	Index. Example: $\exp(a)$ .
<b>log</b>	Logarithmic. Example: $\log(a)$ .
<b>poly- no- mial</b>	Power. Example: $\text{poly}(a, 2)$ .
<b>dot</b>	Matrix dot. Example: $\text{dot}(a, b)$ .
<b>in- verse</b>	Matrix inversion. Example: $\text{inverse}(a)$ .
<b>trans- pose</b>	Matrix transpose. Example: $\text{transpose}(a)$ .
<b>sum</b>	Matrix sum. Example: $\text{sum}(a)$ .
<b>stack</b>	The matrix is concatenated in either column (axis = 0) or row (axis = 1) direction. Example: $\text{stack}(\text{axis}, a, b, c)$ .
<b>sub- script</b>	The subscript operation. Example: $a[0]$ .
<b>qop</b>	Quantum operation, which takes <b>VQC</b> and several Hamiltonians as inputs and outputs the input Hamiltonian expectations. Example: If $w$ represents the variable of <b>VQC</b> , $\text{qop}(\text{VQC}(w), \text{Hamiltonians}, [\text{quantum environment}])$ .
<b>qop_pmeasure</b>	Quantum operation, which is similar to qop. The input of qop_pmeasure consists of the VQC, the qubit to be measured and the component. It calculates the probability of all projected states in a subspace made up of qubits to be measured, and returns some of them. Components store labels for the projected state of the target. Obviously, the probability of any projected state can be seen as the expectation of the Hamiltonian, so qop_pmeasure is a special case of qop. Example: $\text{qop\_pmeasure}(\text{VQC}(w), \text{components}, [\text{quantum environment}])$ .
<b>sig- moid</b>	The activation function. Example: $\text{sigmoid}(a)$ .
<b>soft-</b>	The activation function. Example: $\text{softmax}(a)$ .
<b>7.2.7 max</b>	<b>7.2 Operator</b>
<b>cross_entropy</b>	Cross entropy. Example: $\text{crossEntropy}(a, b)$ .
<b>dropout</b>	Dropout function. Example: $\text{dropout}(a, b)$ .

## 7.3 Variational quantum logic gate

To use quantum operations `qop` or `qop_pmeasure` in VQNet, a variational quantum circuit (VQC) shall be included. And variational quantum logic gates are the basic unit that constitute the variational quantum circuit. The variational quantum logic gate (VQG) maintains a set of variable parameters and a set of constant parameters internally. Only one set of parameters can be assigned during the creation of VQG. Where the VQG contains a set of constant parameters, general quantum logic gates containing determined parameters can be generated through the VQG. Where the VQG contains variational parameters, the parameter values can be modified dynamically and general quantum logic gates corresponding to the parameters can be generated.

Currently, the following variational quantum logic gates are defined in pyQPanda, which are all inherited from the `VariationalQuantumGate`.

VQG	Alias
VariationalQuantumGate_H	VQG_H
VariationalQuantumGate_RX	VQG_RX
VariationalQuantumGate_RY	VQG_RY
VariationalQuantumGate_RZ	VQG_RZ
VariationalQuantumGate_CZ	VQG_CZ
VariationalQuantumGate_CNOT	VQG_CNOT

### 7.3.1 Interface introduction

We can use the variational quantum logic gates by inserting them into variational quantum circuits. Also, we can input variable parameters to the variational quantum logic gates requiring input of parameters. For example, we input variable parameters `x` and `y` to variational quantum logic gates `RX` and `RY`. Alternatively, we can input constant parameters to variational quantum logic gates. For instance, we input a constant parameter `0.12` to `RZ`. Furthermore, we can change the parameters in the variational quantum logic gates by modifying the parameters of the variables.

```
x = var(1)
y = var(2)

vqc = VariationalQuantumCircuit()
vqc.insert(VariationalQuantumGate_H(q[0]))
vqc.insert(VariationalQuantumGate_RX(q[0], x))
vqc.insert(VariationalQuantumGate_RY(q[1], y))
vqc.insert(VariationalQuantumGate_RZ(q[0], 0.12))
vqc.insert(VariationalQuantumGate_CZ(q[0], q[1]))
vqc.insert(VariationalQuantumGate_CNOT(q[0], q[1]))

circuit1 = vqc.feed()
```

(continues on next page)



(continued from previous page)

```
x.set_value(3)
y.set_value(4)

circuit2 = vqc.feed()
```

### 7.3.2 Example

```
from pyqpanda import *

if __name__ == "__main__":

    machine = init_quantum_machine(QMachineType.CPU)
    q = machine.qAlloc_many(2)

    x = var(1)
    y = var(2)

    vqc = VariationalQuantumCircuit()
    vqc.insert(VariationalQuantumGate_H(q[0]))
    vqc.insert(VariationalQuantumGate_RX(q[0], x))
    vqc.insert(VariationalQuantumGate_RY(q[1], y))
    vqc.insert(VariationalQuantumGate_RZ(q[0], 0.12))
    vqc.insert(VariationalQuantumGate_CZ(q[0], q[1]))
    vqc.insert(VariationalQuantumGate_CNOT(q[0], q[1]))

    circuit1 = vqc.feed()

    prog = QProg()
    prog.insert(circuit1)

    print(convert_qprog_to_originir(prog, machine))

    x.set_value([3.])
    y.set_value([4.])

    circuit2 = vqc.feed()
    prog2 = QProg()
    prog2.insert(circuit2)
    print(convert_qprog_to_originir(prog2, machine))
```

```
QINIT 2
CREG 0
H q[0]
RX q[0], (1)
RY q[1], (2)
RZ q[0], (0.12)
CZ q[0], q[1]
CNOT q[0], q[1]
QINIT 2
CREG 0
H q[0]
RX q[0], (3)
RY q[1], (4)
RZ q[0], (0.12)
CZ q[0], q[1]
CNOT q[0], q[1]
```

## 7.4 Variational quantum logic gate (VQG)

To use quantum operations `qop` or `qop_pmeasure` in VQNet, a variational quantum circuit (VQC) shall be included. And variational quantum logic gates are the basic unit that constitute the VQC. The variational quantum logic gate (VQG) maintains a set of variable parameters and a set of constant parameters internally. Only one set of parameters can be assigned during the creation of VQG. Where the VQG contains a set of constant parameters, general quantum logic gates containing determined parameters can be generated through the VQG. Where the VQG contains variational parameters, the parameter values can be modified dynamically and general quantum logic gates corresponding to the parameters can be generated.

Currently, the following variational quantum logic gates are defined in QPanda::Variational, which are all derived from the VQG.

VQG	Alias
VariationalQuantumGate_I	VQG_I
VariationalQuantumGate_H	VQG_H
VariationalQuantumGate_T	VQG_T
VariationalQuantumGate_S	VQG_S
VariationalQuantumGate_X	VQG_X
VariationalQuantumGate_Y	VQG_Y
VariationalQuantumGate_Z	VQG_Z
VariationalQuantumGate_X1	VQG_X1
VariationalQuantumGate_Y1	VQG_Y1
VariationalQuantumGate_Z1	VQG_Z1
VariationalQuantumGate_U1	VQG_U1
VariationalQuantumGate_U2	VQG_U2
VariationalQuantumGate_U3	VQG_U3
VariationalQuantumGate_U4	VQG_U4
VariationalQuantumGate_RX	VQG_RX
VariationalQuantumGate_RY	VQG_RY
VariationalQuantumGate_RZ	VQG_RZ
VariationalQuantumGate_CRX	VQG_CRX
VariationalQuantumGate_CRY	VQG_CRY
VariationalQuantumGate_CRZ	VQG_CRZ
VariationalQuantumGate_CNOT	VQG_CNOT
VariationalQuantumGate_CZ	VQG_CZ
VariationalQuantumGate_SWAP	VQG_SWAP
VariationalQuantumGate_iSWAP	VQG_iSWAP
VariationalQuantumGate_SqiSWAP	VQG_SqiSWAP

### 7.4.1 Interface introduction

class **VariationalQuantumGate**

**VariationalQuantumGate** ()

**Function** Constructing a function

**Parameter** N/A

size\_t **n\_var** ()

**Function** Number of internal variables of the variational quantum logic gate.

**Parameter** N/A

**Return value** Number of variables

```
std::vector<var> &get_vars ()
```

**Function** Obtaining the internal variables of the variational quantum logic gate.

**Parameter** N/A

**Return value** Internal variables of the variational quantum logic gate.

```
std::vector<double> &get_constants ()
```

**Function** Obtaining the internal constants of the variational quantum logic gate.

**Parameter** N/A

**Return value** Internal constants of the variational quantum logic gate.

```
int var_pos (var_var)
```

**Function** Obtaining the internal constants of the variational quantum logic gate.

**Parameter** ● `_var`: variable

**Return value** If no internal index is available, the return result will be -1.

```
virtual QGate feed () const = 0
```

**Function** Instantiating ``QGate``

**Parameter** N/A

**Return value** General quantum logic gate

```
virtual QGate feed (std::map<size_t, double> offset) const
```

**Function** Instantiating `QGate` by specifying offsets.

**Parameter** ● `offset`: the offset mapping corresponding to the variable

**Return value** General quantum logic gate

A brief introduction will be given to the construction of variational quantum logic gates below:

```
class VariationalQuantumGate_H
```

```
VariationalQuantumGate_H (Qubit *q)
```

**Function** Constructing a function through H gate.

**Parameter** ● q: target qubit

class **VariationalQuantumGate\_RX**

**VariationalQuantumGate\_RX** (Qubit \*q, var\_var)

**Function** Constructing a function through RX gate.

**Parameter** ● q: target qubit ● \_var: variable

**VariationalQuantumGate\_RX** (Qubit \*q, double angle)

**Function** Constructing a function through RX gate.

**Parameter** ● q: target qubit ● angle: parameter

class **VariationalQuantumGate\_RY**

**VariationalQuantumGate\_RY** (Qubit \*q, var\_var)

**Function** Constructing a function through RY gate.

**Parameter** ● q: target qubit ● \_var: variable

**VariationalQuantumGate\_RX** (Qubit \*q, double angle)

**Function** Constructing a function through RY gate.

**Parameter** ● q: target qubit ● angle: parameter

class **VariationalQuantumGate\_RZ**

**VariationalQuantumGate\_RZ** (Qubit \*q, var\_var)

**Function** Constructing a function through RZ gate.

**Parameter** ● q: target qubit ● \_var: variable

**VariationalQuantumGate\_RZ** (Qubit \*q, double angle)

**Function** Constructing a function through RZ gate.

**Parameter** ● q: target qubit ● angle: parameter

```
class VariationalQuantumGate_CZ
```

```
    VariationalQuantumGate_CZ (Qubit *q1, Qubit *q2)
```

**Function** Constructing a function through CZ gate.

**Parameter** ● q1: control qubit ● q2: target qubit

```
class VariationalQuantumGate_CNOT
```

```
    VariationalQuantumGate_CNOT (Qubit *q1, Qubit *q2)
```

**Function** Constructing a function through CNOT gate.

**Parameter** ● q1: control qubit ● q2: target qubit

The variational quantum logic gates are used in a way similar to quantum logic gates, which will not be repeated here. In case of any question, see the relevant contents of quantum logic gates.

## 7.4.2 7.4.2 Dynamic modification of parameters

If the constructed VQC contains variable parameters, the parameter values can be modified dynamically in the following way with the general quantum logic gates corresponding to the parameters generated.

(1): setValue(), used in the way below:

```
object.setValue(newValue);
```

(2): “=” operator re-assigned, used in the way below:

```
object = newValue;
```

### 7.4.2.1 7.4.2.1 Example:

```
import pyqpanda as pq
import numpy as np

if __name__=="__main__":

    machine = pq.init_quantum_machine(pq.CPU)
    q = machine.qAlloc_many(2)

    x = pq.var(1)
    y = pq.var(2)
```

(continues on next page)

(continued from previous page)

```

temp = np.matrix([5])
ss=pq.var(temp)

vqc = pq.VariationalQuantumCircuit()
vqc.insert(pq.VariationalQuantumGate_H(q[0]))
vqc.insert(pq.VariationalQuantumGate_RX(q[0], ss))
vqc.insert(pq.VariationalQuantumGate_RY(q[1], y))
vqc.insert(pq.VariationalQuantumGate_RZ(q[0], 0.12))
vqc.insert(pq.VariationalQuantumGate_CZ(q[0], q[1]))
vqc.insert(pq.VariationalQuantumGate_CNOT(q[0], q[1]))
vqc.insert(pq.VariationalQuantumGate_U1(q[0], x))
vqc.insert(pq.VariationalQuantumGate_U2(q[0], np.pi, x))
vqc.insert(pq.VariationalQuantumGate_U3(q[0], np.pi, x, y))
vqc.insert(pq.VariationalQuantumGate_U4(q[0], np.pi, x, y, ss))

circuit1 = vqc.feed()

prog = pq.QProg()
prog.insert(circuit1)

print(pq.convert_qprog_to_originir(prog, machine))

x.set_value([3.])
y.set_value([4.])

circuit2 = vqc.feed()
prog2 = pq.QProg()
prog2.insert(circuit2)
print(pq.convert_qprog_to_originir(prog2, machine))

```

The above example will get the result below:

```

QINIT 2
CREG 0
H q[0]
RX q[0], (56)
RY q[1], (2)
RZ q[0], (0.12)
CZ q[0], q[1]
CNOT q[0], q[1]

```

(continues on next page)

(continued from previous page)

```

U1 q[0], (1)
U2 q[0], (3.1415927, 1)
U3 q[0], (3.1415927, 1, 2)
U4 q[0], (3.1415927, 1, 2, 5)

QINIT 2
CREG 0
H q[0]
RX q[0], (56)
RY q[1], (4)
RZ q[0], (0.12)
CZ q[0], q[1]
CNOT q[0], q[1]
U1 q[0], (3)
U2 q[0], (3.1415927, 3)
U3 q[0], (3.1415927, 3, 4)
U4 q[0], (3.1415927, 3, 4, 5)

```

## 7.5 Optimization algorithm (gradient descent)

This chapter will describe the use of optimization algorithms in VQNet, including the classical gradient descent algorithm and the improved gradient descent algorithm which are two of the most commonly used methods to solve the model parameters of machine learning algorithms, i.e., unconstrained optimization problems. We have implemented such algorithms as `VanillaGradientDescentOptimizer`, `MomentumOptimizer`, `AdaGradOptimizer`, `RMSPropOptimizer` and `AdamOptimizer` in pyQPanda. All these algorithms are inherited from `Optimizer`.

### 7.5.1 Interface introduction

We generate an optimizer by calling the `minimize` interface of the gradient descent optimizer. The gradient descent optimizer is generally constructed as below:

```

VanillaGradientDescentOptimizer.minimize(
    loss, # Loss function
    0.01, # learning rate
    1.e-6) # The end condition

MomentumOptimizer.minimize(
    loss, # Loss function
    0.01, # learning rate
    0.9) # The momentum coefficient

```

(continues on next page)



(continued from previous page)

```

AdaGradOptimizer.minimize(
    loss, # Loss function
    0.01, # learning rate
    0.0, # Initial value of accumulation
    1.e-10) # Small numbers to avoid a zero denominator

RMSOptimizer.minimize(
    loss, # Loss function
    0.01, # learning rate
    0.9, # Historical or upcoming gradient discount factor
    1.e-10) # Small numbers to avoid a zero denominator

AdamOptimizer.minimize(
    loss, # Loss function
    0.01, # learning rate
    0.9, # First order momentum decay coefficient
    0.999, # Second order momentum decay coefficient
    1.e-10) # Small numbers to avoid a zero denominator

```

## 7.5.2 Example

The example codes mainly demonstrate the fitting of discrete points with straight lines. We define the training data X and Y which are variables and indicate the coordinates of the discrete points. Also, we define two differentiable variables w and b, which w represents the slope and b represents the y-intercept. Then, we define the underscore of variable Y which represents the slope w multiplied by the variable x plus the intercept.

Next, we define the loss function. and compute the mean square value between variable Y and its underscore.

We call the `minimize` interface of the gradient descent optimizer and construct a classical gradient descent optimizer by taking the loss function, learning rate and ending condition as parameters.

We can obtain all differentiable nodes through the `get_variables` interface of the optimizer.

We defined the number of iterations as 1000. Then, we conduct an optimization operation by calling the `run` interface of the optimizer. The second parameter indicates the current number of optimizations. Now, this parameter is only used by `AdamOptimizer`, and the other optimizers can be directly assigned with a value of 0.

We can obtain the loss value after the optimization through the `get_loss` interface of the optimizer. Through the `eval` interface, we can get the current value of a differentiable variable.

```

from pyqpanda import *
import numpy as np

x = np.array([3.3, 4.4, 5.5, 6.71, 6.93, 4.168, 9.779, 6.182, 7.59,\

```

(continues on next page)

(continued from previous page)

```

        2.167, 7.042, 10.791, 5.313, 7.997, 5.654, 9.27, 3.1])
y = np.array([1.7, 2.76, 2.09, 3.19, 1.694, 1.573, 3.366, 2.596, 2.53,\
             1.221, 2.827, 3.465, 1.65, 2.904, 2.42, 2.94, 1.3])

X = var(x.reshape(len(x), 1))
Y = var(y.reshape(len(y), 1))

W = var(0, True)
B = var(0, True)

Y_ = W*X + B

loss = sum(poly(Y_ - Y, var(2))/len(x))

optimizer = VanillaGradientDescentOptimizer.minimize(loss, 0.01, 1.e-6)
leaves = optimizer.get_variables()

for i in range(1000):
    optimizer.run(leaves, 0)
    loss_value = optimizer.get_loss()
    print("i: ", i, " loss: ", loss_value, " W: ", eval(W, True), " b: ", \ eval(B, \
↪ True))

```

We plot a graph with the hash points and the fitted lines.

```

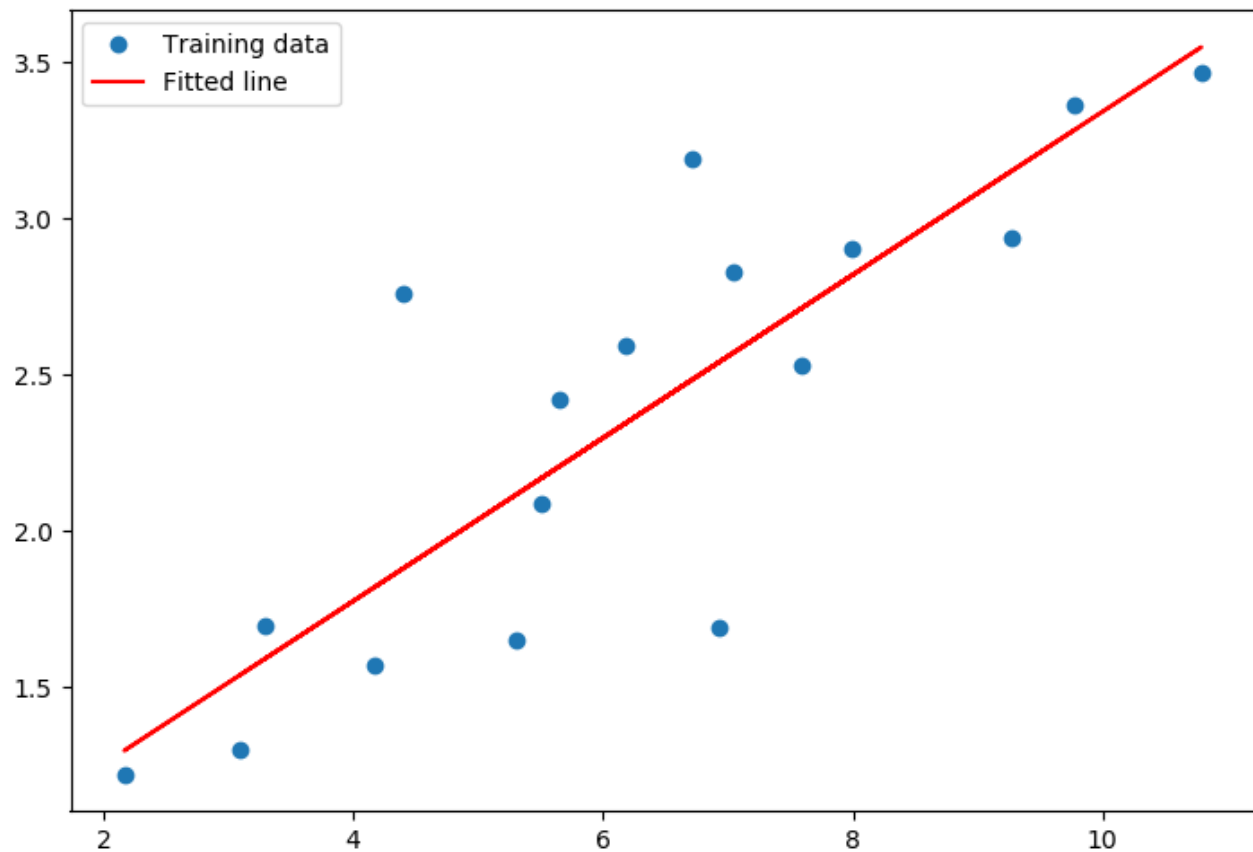
import matplotlib.pyplot as plt

w2 = W.get_value()[0, 0]
b2 = B.get_value()[0, 0]

plt.plot(x, y, 'o', label = 'Training data')
plt.plot(x, w2*x + b2, 'r', label = 'Fitted line')
plt.legend()
plt.show()

```

```
i: 971 loss: 0.15447272923278282 W: [[0.26157713]] b: [[0.72831562]]
i: 972 loss: 0.15446974657108087 W: [[0.26155299]] b: [[0.72848674]]
i: 973 loss: 0.154466778373562 W: [[0.26152891]] b: [[0.72865744]]
i: 974 loss: 0.15446382457008312 W: [[0.26150489]] b: [[0.72882772]]
i: 975 loss: 0.1544608850908416 W: [[0.26148093]] b: [[0.72899759]]
i: 976 loss: 0.15445795986637306 W: [[0.26145703]] b: [[0.72916705]]
i: 977 loss: 0.15445504882755018 W: [[0.26143319]] b: [[0.7293361]]
i: 978 loss: 0.15445215190558084 W: [[0.2614094]] b: [[0.72950474]]
i: 979 loss: 0.15444926903200643 W: [[0.26138567]] b: [[0.72967297]]
i: 980 loss: 0.15444640013870023 W: [[0.261362]] b: [[0.72984079]]
i: 981 loss: 0.15444354515786618 W: [[0.26133839]] b: [[0.7300082]]
i: 982 loss: 0.15444070402203675 W: [[0.26131483]] b: [[0.7301752]]
i: 983 loss: 0.15443787666407188 W: [[0.26129133]] b: [[0.7303418]]
i: 984 loss: 0.15443506301715673 W: [[0.26126789]] b: [[0.730508]]
i: 985 loss: 0.15443226301480056 W: [[0.2612445]] b: [[0.73067379]]
i: 986 loss: 0.15442947659083534 W: [[0.26122117]] b: [[0.73083918]]
i: 987 loss: 0.15442670367941363 W: [[0.2611979]] b: [[0.73100417]]
i: 988 loss: 0.15442394421500752 W: [[0.26117468]] b: [[0.73116876]]
i: 989 loss: 0.15442119813240665 W: [[0.26115153]] b: [[0.73133295]]
i: 990 loss: 0.15441846536671705 W: [[0.26112842]] b: [[0.73149674]]
i: 991 loss: 0.15441574585335935 W: [[0.26110538]] b: [[0.73166013]]
i: 992 loss: 0.15441303952806756 W: [[0.26108238]] b: [[0.73182313]]
i: 993 loss: 0.15441034632688708 W: [[0.26105945]] b: [[0.73198572]]
i: 994 loss: 0.15440766618617355 W: [[0.26103657]] b: [[0.73214793]]
i: 995 loss: 0.15440499904259133 W: [[0.26101375]] b: [[0.73230974]]
i: 996 loss: 0.15440234483311183 W: [[0.26099098]] b: [[0.73247116]]
i: 997 loss: 0.15439970349501206 W: [[0.26096826]] b: [[0.73263219]]
i: 998 loss: 0.15439707496587343 W: [[0.26094561]] b: [[0.73279282]]
i: 999 loss: 0.1543944591835798 W: [[0.260923]] b: [[0.73295307]]
```



## 7.6 Comprehensive example

### 7.6.1 QAOA

QAOA is a well-known hybrid quantum-classical algorithm. The max-cut problem of  $n$  objects requires  $n$  qubits to encode the result, where the measured result (binary string) indicates the cut configuration of the problem.

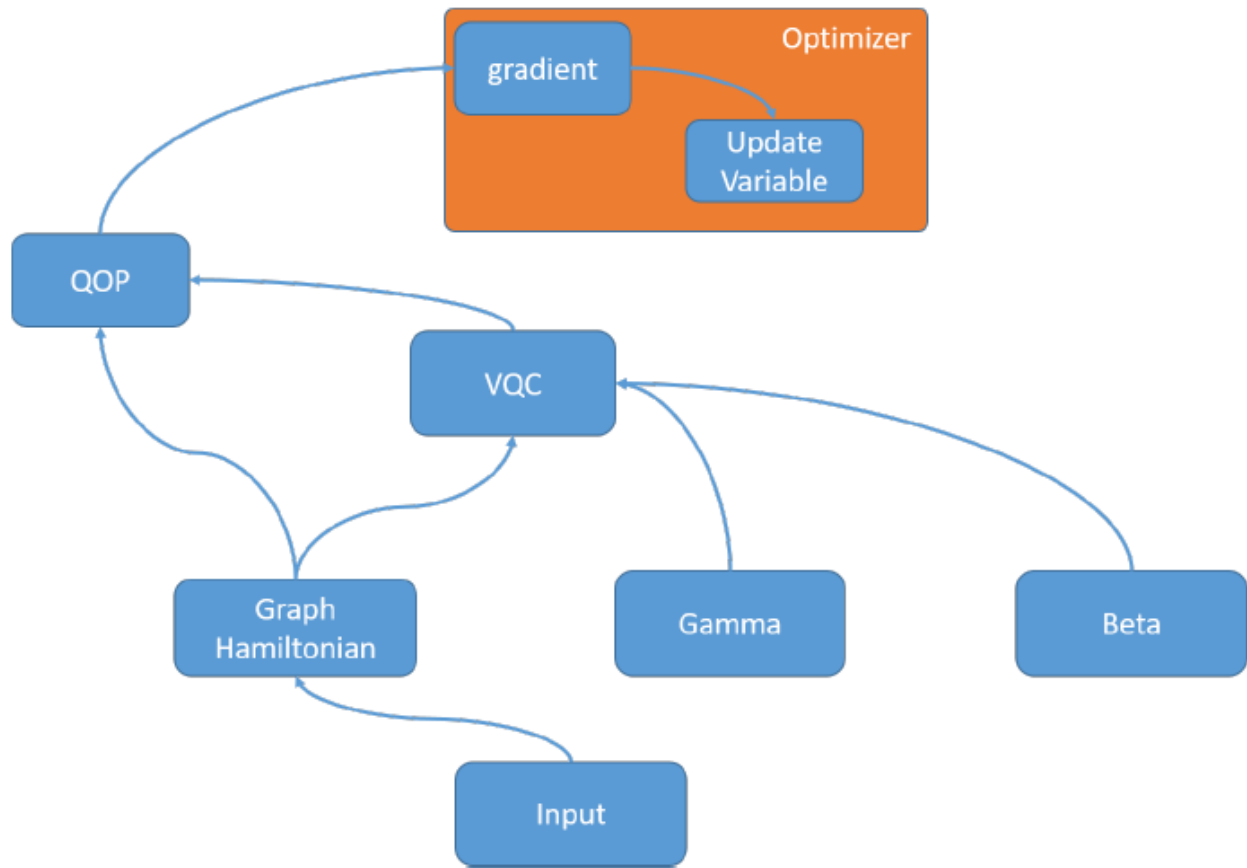
We can effectively implement QAOA algorithm of the MAX-CUT problem through VQNet. The flow chart of QAOA in VQNet is as below.

We determine a MAX-CUT problem as below:

Firstly, we input the graphic information of the MAX-CUT problem to construct the Hamiltonian of the problem.

```
problem = {'Z0 Z4':0.73, 'Z0 Z5':0.33, 'Z0 Z6':0.5, 'Z1 Z4':0.69, 'Z1 Z5':0.36,
           'Z2 Z5':0.88, 'Z2 Z6':0.58, 'Z3 Z5':0.67, 'Z3 Z6':0.43}
```

Then, we construct the vqc of QAOA with the Hamiltonian and beta and gamma, the two variable parameters to be optimized. The input parameters of QOP include the Hamiltonian of interest, VQC, a set of qubits, and the quantum operating environment. The output of QOP is the expectation of the Hamiltonian of interest. In this problem, the loss function is the expectation of the Hamiltonian of interest. Therefore, the output of QOP shall be minimized. We optimize the variables beta and gamma in the vqc with MomentumOptimizer.



```

from pyqpanda import *
import numpy as np

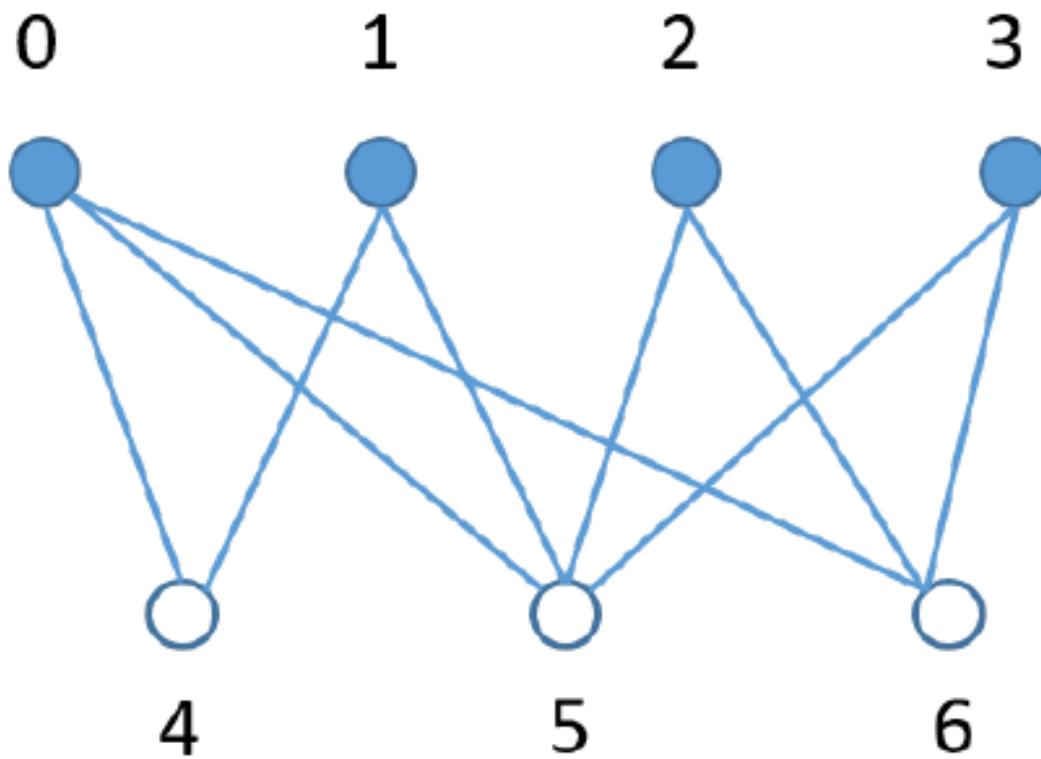
def oneCircuit(qlist, Hamiltonian, beta, gamma):
    vqc=VariationalQuantumCircuit()
    for i in range(len(Hamiltonian)):
        tmp_vec=[]
        item=Hamiltonian[i]
        dict_p = item[0]
        for iter in dict_p:
            if 'Z'!= dict_p[iter]:
                pass
            tmp_vec.append(qlist[iter])

        coef = item[1]

        if 2 != len(tmp_vec):
            pass
        vqc.insert(VariationalQuantumGate_CNOT(tmp_vec[0], tmp_vec[1]))
        vqc.insert(VariationalQuantumGate_RZ(tmp_vec[1], 2*gamma*coef))
        vqc.insert(VariationalQuantumGate_CNOT(tmp_vec[0], tmp_vec[1]))

```

(continues on next page)



(a) Max-Cut Graph

edge	weight	edge	weight
0,4	0.73	2,5	0.88
0,5	0.33	2,6	0.58
0,6	0.5	3,5	0.67
1,4	0.69	3,6	0.43
1,5	0.36	sum	5.17

(b) Weight of Graph Edge

(continued from previous page)

```

    for j in qlist:
        vqc.insert(VariationalQuantumGate_RX(j, 2.0*beta))
    return vqc
if __name__=="__main__":
    problem = {'Z0 Z4':0.73, 'Z0 Z5':0.33, 'Z0 Z6':0.5, 'Z1 Z4':0.69, 'Z1 Z5':0.36,
               'Z2 Z5':0.88, 'Z2 Z6':0.58, 'Z3 Z5':0.67, 'Z3 Z6':0.43}
    Hp = PauliOperator(problem)
    qubit_num = Hp.getMaxIndex()

    machine=init_quantum_machine(QMachineType.CPU)
    qlist = machine.qAlloc_many(qubit_num)

    step = 4

    beta = var(np.ones((step,1), dtype = 'float64'), True)
    gamma = var(np.ones((step,1), dtype = 'float64'), True)

    vqc=VariationalQuantumCircuit()

    for i in qlist:
        vqc.insert(VariationalQuantumGate_H(i))

    for i in range(step):
        vqc.insert(oneCircuit(qlist, Hp.toHamiltonian(1), beta[i], gamma[i]))

    loss = qop(vqc, Hp, machine, qlist)
    optimizer = MomentumOptimizer.minimize(loss, 0.02, 0.9)
    leaves = optimizer.get_variables()
    for i in range(100):
        optimizer.run(leaves, 0)
        loss_value = optimizer.get_loss()
        print("i: ", i, " loss:", loss_value )
    # The verification results
    prog = QProg()
    qcir = vqc.feed()
    prog.insert(qcir)
    directly_run(prog)

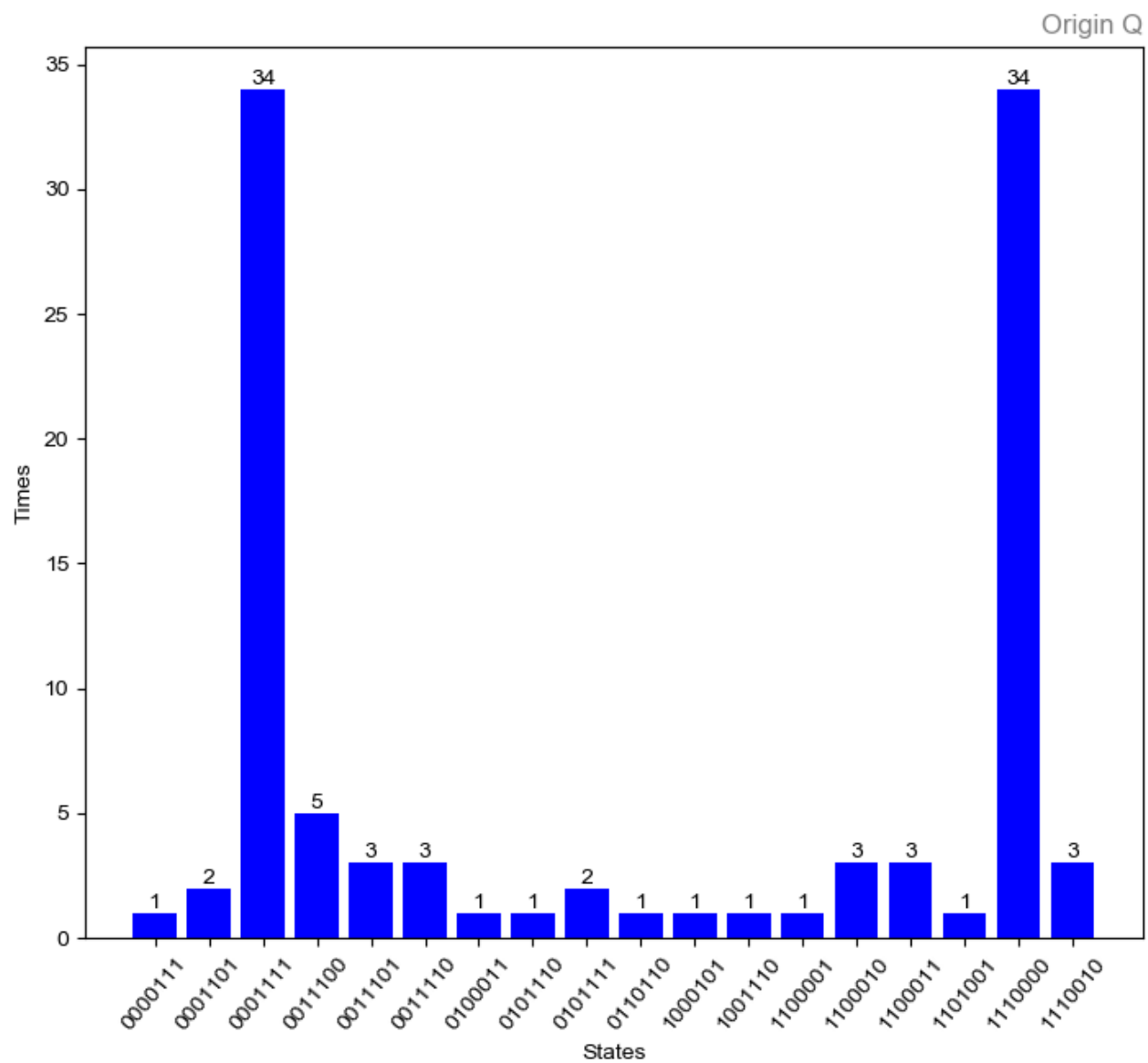
    result = quick_measure(qlist, 100)
    print(result)

```

We draw a histogram with the measured results, from which we can see that the two qubit strings ‘0001111’ and ‘1110000’ generate the largest probability which is also the solution of this problem.

```
i: 68 loss: -4.531896817440074
i: 69 loss: -4.531277056427369
i: 70 loss: -4.531523214445907
i: 71 loss: -4.532415929907813
i: 72 loss: -4.533027005538832
i: 73 loss: -4.5329515117005075
i: 74 loss: -4.532274969728897
i: 75 loss: -4.532038225807408
i: 76 loss: -4.532626207953599
i: 77 loss: -4.532897270288688
i: 78 loss: -4.533058929606472
i: 79 loss: -4.5332872059861815
i: 80 loss: -4.533141451294506
i: 81 loss: -4.532968747552253
i: 82 loss: -4.53314236860445
i: 83 loss: -4.533182138865922
i: 84 loss: -4.533282640666032
i: 85 loss: -4.533423868405955
i: 86 loss: -4.533291285780287
i: 87 loss: -4.533276955131906
i: 88 loss: -4.533501070030698
i: 89 loss: -4.53354880103886
i: 90 loss: -4.5334807644696555
i: 91 loss: -4.533445341878178
i: 92 loss: -4.53339020341782
i: 93 loss: -4.533474177066523
i: 94 loss: -4.533600470383044
i: 95 loss: -4.533600134653298
i: 96 loss: -4.533563917273043
i: 97 loss: -4.533567005919695
i: 98 loss: -4.533552117601373
i: 99 loss: -4.533566272672815
{'0000111': 1, '0001101': 2, '0001111': 34, '0011100': 5, '0011101': 3, '0011110': 3, '0100011': 1
, '0101110': 1, '0101111': 2, '0110110': 1, '1000101': 1, '1001110': 1, '1100001': 1, '1100010': 3
, '1100011': 3, '1101001': 1, '1110000': 34, '1110010': 3}
```







## 8 BASIS OF QUANTUM ALGORITHM

### 8.1 8.1 Review of basic concepts

#### 8.1.1 8.1.1 Basic definitions

In physics, a quantum is the smallest inseparable basic unit of physical quantities. Bit, as a computer term, indicates the smallest unit of information. Different from a classical bit which can only be 0 or 1, a qubit can exist in the intermediate state superimposed in any proportion between 0 and 1.

The basic operation performed on qubits is called a quantum gate.

A quantum gate can either be a single-qubit gate and a multi-qubit gate. The single-qubit gate includes Hadamard gate, Pauli-X/Y/Z gate and rotating X/Y/Z gate. Two-qubit gates included controlled single-qubit gates (such as CNOT gates) and swap gates. A single-qubit gate and a two-qubit gate can be further extended to a multi-qubit gate by such extension means as controlled operation. It should be noted that measurement is a special quantum gate which is irreversible and changes the state of qubits.

Any quantum algorithm is a combination of these basic quantum gates.

See the common quantum logic gate matrix form for the definition of general quantum gate.

#### 8.1.2 8.1.2 pyQPanda interface function

In pyQPanda the defined function of a quantum gate is as below:

```
gate = H(qubit)
```

---

#### Note

The input parameters are Qubit and other parameters, and the return value is QGate (quantum gate) which can be inserted into the quantum circuit.

---

Many kinds of quantum gates are defined in pyQPanda. Particularly, for the quantum gate U4 that supports full customization in pyQPanda, its interface functions supports simultaneous overloads as below:

As described above, the interface function of quantum gates is provided with two extension operations: transposed conjugate and controlled operation. Both of the operations can be implemented in two ways.

The two interface functions of transposed conjugate operation are defined as below:

```
gate = H(qubit)
gate1 = gate.dagger()
gate.setDagger(true)
```

---

### Note

The dagger function returns a new quantum gate based on the target quantum gate, while the setDagger function returns the target quantum gate subject to transposed conjugate.

---

The two interface functions of controlled operation are defined as below:

```
gate = H(qubit)
gate1 = gate.control(QVec)
gate.setControl(QVec)
```

---

### Note

The difference of controlled operation is similar to that of transposed conjugate operation, but the input parameter of the controlled operation function is Qvec (list of qubits) rather than a single qubit.

---

## 8.1.3 Example

Below is a program example to show the implementation of codes for basic qubit and quantum gate operations.

```
#!/usr/bin/env python

import pyqpanda as pq

if __name__ == "__main__":

    machine = pq.init_quantum_machine(pq.QMachineType.CPU)
    qubits = machine.qAlloc_many(3)
    control_qubits = [qubits[0], qubits[1]]
    prog = pq.create_empty_qprog()
```

(continues on next page)

(continued from previous page)

```

# Building quantum programs
prog.insert(pq.H(qubits[0])) \
    .insert(pq.H(qubits[1])) \
    .insert(pq.H(qubits[0]).dagger()) \
    .insert(pq.X(qubits[2]).control(control_qubits))

# Perform probability measurements on quantum programs
result = pq.prob_run_dict(prog, qubits, -1)
pq.destroy_quantum_machine(machine)

# Print measurement results
for key in result:
    print(key+":"+str(result[key]))

```

The output result shall be as shown below, with the probability of 0.5 to get  $|000\rangle$  and  $|010\rangle$ :

```

000:0.4999999999999894
001:0.0
010:0.4999999999999894
011:0.0
100:0.0
101:0.0
110:0.0
111:0.0

```

Above is the basic definitions of qubit and quantum gate and the introduction to the call in pyQPanda.

## 8.2 Experimental state preparation and quantum entanglement

### 8.2.1 Experimental state preparation

Experimental state preparation refers to the construction of the initial quantum state of any algorithm in quantum computing, which is the initial step of quantum computing.

Taking the Space two-state space with a single qubit, in actual quantum computing, we can directly get the default quantum state as the ground state  $|0\rangle$ , and also get the ground state  $|1\rangle$  indirectly through NOT gate.

For any given target superposition quantum state, we need to construct the corresponding combination of quantum gates to obtain such state. The process of preparing any given target superposition state by starting from the ground state  $|0\rangle$  is known as the initial state preparation.

### 8.2.1.1 8.2.1.1 Maximum superposition state

Taking two-qubit state space as an example, we can get uniform superposition of all ground states in the two-qubit state space by performing Hadamard gate operation for each qubit, starting from  $|0\rangle^{\otimes 2}$ .

Similarly, in any dimensional state space, we can get the quantum state for uniform linear combinations of all ground states by starting from the multidimensional  $|0\rangle$  ground state by virtue of Hadamard gate.

Such quantum state is called the maximum superposition state which is required by the initial state of qubits in many quantum computings. Also, the parallelism of quantum computing depends on such state.

Through the experimental state preparation, we can obtain any basic quantum state, thus completing the construction of operation objects in quantum computing. However, prior to the operation, we need to make clear constraint on the qubits used in quantum computing –entanglement and correlation.

We need to introduce pure state and mixed state before giving an introduction to quantum entanglement.

All quantum states rather than ground states are superposition states. Superposition states can be divided into coherent superposition and incoherent superposition which are called pure state and mixed state respectively.

Many methods are available to distinguish a pure state and a mixed state, including Bloch Sphere by which the state space is related to Bloch Sphere and the quantum state on the Sphere is a pure state while that in the Sphere is a mixed state.

Another important method is density matrix where the non-diagonal elements of the density matrix of mixed states are all 0.

## 8.2.2 8.2.2 Quantum entanglement

If the quantum state  $|\rangle$  of a quantum system can be expressed in the form of the direct product of two quantum systems like  $|\rangle = |0\rangle \otimes |1\rangle$ , such quantum state is called the direct product state.

---

### Note

Any quantum state rather than direct product state is an entangled state.

---

For example, a two-qubit Bell state  $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$  cannot be factored as the direct product of two single-qubit quantum states.

The entangled state of quantum is given with quantum correlation beyond classical one. To give full play to the parallelism and efficiency of quantum computing, the qubits used in quantum computing shall be entangled and correlated.

### 8.2.3 Maximum superposition state preparation

The following is the code implementation for the maximum superposition state preparation based on pyQPanda, and the qubits called are entangled and correlated.

```
#!/usr/bin/env python

import pyqpanda as pq

if __name__ == "__main__":

    machine = pq.init_quantum_machine(pq.QMachineType.CPU)
    qubits = machine.qAlloc_many(3)
    prog = pq.create_empty_qprog()

    # Building quantum programs
    prog.insert(pq.H(qubits[0])) \
        .insert(pq.H(qubits[1])) \
        .insert(pq.H(qubits[2]))

    # Perform probability measurements on quantum programs
    result = pq.prob_run_dict(prog, qubits, -1)
    pq.destroy_quantum_machine(machine)

    # Print measurement results
    for key in result:
        print(key+": "+str(result[key]))
```

The results shall be that all quantum states in the 3-qubit space are obtained with the uniform probability of 1/8.

```
000, 0.125
001, 0.125
010, 0.125
011, 0.125
100, 0.125
101, 0.125
110, 0.125
111, 0.125
```

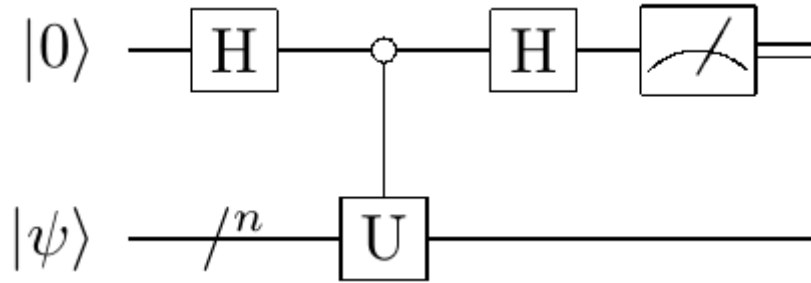
## 8.3 8.3 Hadamard Test and SWAP Test

A quantum circuit is a combination of a series of quantum gate operations. Among the numerous quantum circuits, some are used repeatedly during the construction of quantum algorithms. These frequently called quantum circuit components are called basic circuits of algorithm algorithms. Several common basic circuits will be introduced below.

### 8.3.1 8.3.1 Hadamard Test

The Hadamard Test quantum circuit is mainly used to give the projection and expectation  $\langle U \rangle$  of any given unitary operator  $U$  on the given quantum state .

The quantum circuit diagram of Hadamard Test is simple in structure, as shown below.



The whole quantum circuit can be considered as the combination of quantum gate operations performed for the  $n+1$ -dimensional quantum state  $|0\rangle| \psi \rangle$  which is composed of qubits in two register  $Q = (H \otimes I^{\otimes n}) (C - U) (H \otimes I^{\otimes n})$  where  $C-U$  represents a controlled gate based on the unitary operator  $U$ .

#### 8.3.1.1 8.3.1.1 Output results and generalization

The derivation of the output results of Hadamard Test quantum circuit generates the following conclusions.

$$\begin{aligned} Q|0\rangle|\psi\rangle &= \frac{|0\rangle + |1\rangle}{2} + \frac{|0\rangle - |1\rangle}{2} U|\psi\rangle \\ &= |0\rangle \frac{|\psi\rangle + U|\psi\rangle}{2} + |1\rangle \frac{|\psi\rangle - U|\psi\rangle}{2} \end{aligned}$$

The probability to get  $|0\rangle, |1\rangle$  by measuring the output resulting quantum state is as below:

$$P_0 = \frac{1}{4} \|(I + U)(Q|0\rangle|\psi\rangle)\|^2 = \frac{1 + \text{Re}(\langle \psi | U | \psi \rangle)}{2}, P_1 = 1 - P_0$$

By deduced by the formula, the measurement probability of Hadamard Test results is related to the real part of  $\text{Re}(\langle U \rangle)$  (unitary operator  $U$ ) mapped and expected on the quantum state .

We can replace the H gate before the measurement with  $RX(\frac{\pi}{2})$  gate to obtain the resulting quantum state related to the probability and the mapped and expected imaginary part.



### 8.3.1.2 8.3.1.2 Code example

A code of  $|\psi\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$ , Hadamard Test is as below:

```
#!/usr/bin/env python

import pyqpanda as pq

if __name__ == "__main__":

    machine = pq.init_quantum_machine(pq.QMachineType.CPU)
    cqv = machine.qAlloc_many(1)
    tqv = machine.qAlloc_many(1)
    prog = pq.create_empty_qprog()

    # Building quantum programs
    prog.insert(pq.H(cqv[0])) \
        .insert(pq.H(tqv[0])) \
        .insert(pq.H(tqv[0]).control([cqv[0]])) \
        .insert(pq.H(cqv[0]))

    # Perform probability measurements on quantum programs
    result = pq.prob_run_dict(prog, cqv, -1)
    pq.destroy_quantum_machine(machine)

    # Print measurement results
    for key in result:
        print(key+": "+str(result[key]))
```

The output result shall be as shown below, with the probability of  $\frac{1+\sqrt{2}/2}{2}$  and that of  $1 - \frac{1+\sqrt{2}/2}{2}$  to get  $|0\rangle$  and  $|1\rangle$  respectively:

```
0:0.853553
1:0.146447
```

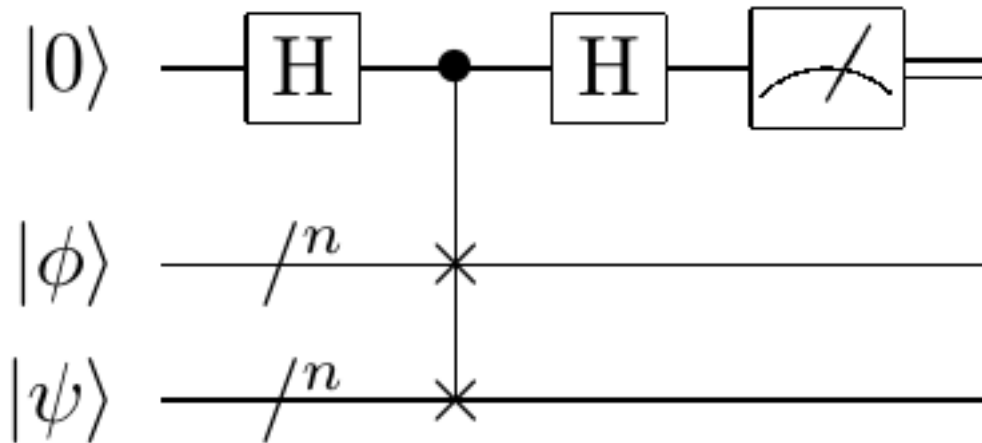
Hadamard Test can be used for a wide range of purposes and has many forms among which one is basic SWAP Test of quantum circuits.

### 8.3.2 SWAP Test

With any two quantum states with the same dimension given, the fidelity of the two quantum states can be obtained through the SWAP Test circuit, which reflects the overlapping of the states.

The fidelity of the two quantum states refers to the square value of inner product norm of the quantum states,  $|\langle\phi|\psi\rangle|^2$

The quantum circuit diagram of SWAP Test is as below.



The formula derivation and verification process for SWAP Test is completely similar to that for Hadamard Test, and the probability of  $|0\rangle, |1\rangle$  measured by the first register for the resulting quantum state is related to the fidelity of the two given quantum states.

$$P_0 = \frac{1 + |\langle\psi|\phi\rangle|^2}{2}, P_1 = 1 - P_0$$

SWAP Test, as a special form of Hadamard, provides the measurement results related to the fidelity for the two given quantum states, which shows significance of application. Also, it plays an important role in the study on inner product of quantum states.

If the controlled SWAP gate is replaced with a general controlled gate  $F$ , we can obtain the resulting quantum state of the general form of Hadamard Test.

$$\frac{|0\rangle}{2}(I + F)|\phi\rangle|\psi\rangle + \frac{|1\rangle}{2}(I - F)|\phi\rangle|\psi\rangle$$

#### 8.3.2.1 Code example

There lies a minor difference between the code example of SWAP Test and that of Hadamard Test.

Take  $|\phi\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}, |\psi\rangle = |1\rangle$ , an example of SWAP Test code is as below:

```
#!/usr/bin/env python
```

(continues on next page)

(continued from previous page)

```

import pyqpanda as pq

if __name__ == "__main__":

    machine = pq.init_quantum_machine(pq.QMachineType.CPU)
    cqv = machine.qAlloc_many(1)
    tqv = machine.qAlloc_many(1)
    qvec = machine.qAlloc_many(1)
    prog = pq.create_empty_qprog()

    # Building quantum programs
    prog.insert(pq.H(cqv[0])) \
        .insert(pq.H(tqv[0])) \
        .insert(pq.X(qvec[0])) \
        .insert(pq.SWAP(tqv[0], qvec[0]).control([cq[0]])) \
        .insert(pq.H(cqv[0]))

    # Perform probability measurements on quantum programs
    result = pq.prob_run_dict(prog, cqv, -1)
    pq.destroy_quantum_machine(machine)

    # Print measurement results
    for key in result:
        print(key+": "+str(result[key]))

```

The output result shall be as shown below, with the probabilities of 0.75 and 0.25 to get  $|0\rangle$  and  $|1\rangle$ :

```

0:0.75
1:0.25

```

## 8.4 Amplitude magnification

The Amplitude Amplification circuit is mainly used to amplify the given pure state so as to adjust the probability distribution of its measured results.

### 8.4.1 Background of algorithm

For a finite set of which the size is known and for which binary classification is available and standard  $f$  is determined by standard  $f$ , any of element from the set  $|\Psi\rangle$  can be expressed as the linear combination of two orthogonal ground states  $|\Psi_0\rangle, |\Psi_1\rangle$  based on the  $f$ .

$$|\psi\rangle = \sin\theta |\varphi_1\rangle + \cos\theta |\varphi_0\rangle, |\varphi_0\rangle = |\varphi_1^\perp\rangle$$

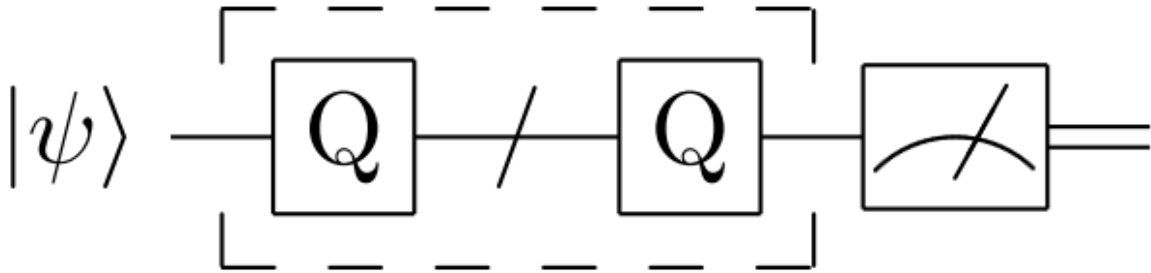
The amplitude amplification quantum circuit can amplify the amplitude of  $|\Psi_1\rangle$  in the expression of superposition state  $|\Psi\rangle$ , thus obtaining a resulting quantum state, so as to get the target quantum state  $|\Psi_1\rangle$  with a large probability.

Suppose that we can construct a combination of quantum gate operations which is the amplitude amplification operator  $Q$ , and we can obtain the quantum state in the following form by acting  $Q$  on the quantum state  $|\Psi\rangle$  for  $k$  times.

$$|\psi_k\rangle = \sin k\theta |\varphi_1\rangle + \cos k\theta |\varphi_0\rangle, k\theta \approx \frac{\pi}{2}$$

Then, we complete the required construction of amplitude amplification quantum circuits.

The quantum circuit diagram is as follows:



Suppose that the quantum state  $|\Psi\rangle$  based on the set and classification standard  $f$  has been prepared, which depends on the construction of amplitude amplification operator  $Q$ .

The amplitude amplification operator is defined as below:

$$P_1 = I - 2|\psi_1\rangle\langle\psi_1|, P = I - 2|\psi\rangle\langle\psi|, Q = -PP_1$$

---

#### Note

How to prepare the quantum state through the set and classification standard  $f$ ? How are  $P_1$  and  $P$  implemented through the quantum circuit?

---

Through simple verification, we can know that the operation  $Q$  in the space formed by  $\{|\varphi_1\rangle, |\varphi_0\rangle\}$  can be expressed as follows:

$$Q = \begin{bmatrix} \cos(2\theta) & -\sin(2\theta) \\ \sin(2\theta) & \cos(2\theta) \end{bmatrix}$$

Essentially, it can be considered as a rotating quantum gate operation with an angle of 2. Therefore, we can get the following formula:

$$Q^n|\psi\rangle = \sin(2n+1)\theta|\varphi_1\rangle + \cos(2n+1)\theta|\varphi_0\rangle$$

The amplitude amplification quantum circuit can be completed by selecting a proper number of rotation  $n$  to make  $\sin^2(2n+1)\theta$  be closest to 1.

Compared with the classical traversal classification method, the amplitude amplification quantum circuit can fully reflect the advantages of quantum computing.

### 8.4.2 Code example

Take  $\Omega = \{0, 1\}$ ,  $|\psi\rangle = \frac{|0\rangle + |1\rangle}{2}$ ,  $P_1 = I - 2|1\rangle\langle 1| = Z$

Below is an example of codes corresponding to the amplitude amplification quantum circuit:

```
#!/usr/bin/env python

import pyqpanda as pq
from numpy import pi

if __name__ == "__main__":

    machine = pq.init_quantum_machine(pq.QMachineType.CPU)
    qvec = machine.qAlloc_many(1)
    prog = pq.create_empty_qprog()

    # Building quantum programs
    prog.insert(pq.H(qvec[0]))
    for i in range(7):
        prog.insert(pq.RY(qvec[0], pi/2))

    # Perform probability measurements on quantum programs
    result = pq.prob_run_dict(prog, qvec, -1)
    pq.destroy_quantum_machine(machine)

    # Print measurement results
    for key in result:
        print(key+": "+str(result[key]))
```

The output result shall be as shown below, with the probabilities of 1 and 0 to get  $|0\rangle$  and  $|1\rangle$  respectively:

```
0:1
1:0
```

## 8.5 8.5 Quantum Fourier transform

The quantum Fourier transform (QFT) is the quantum version of the classical inverse discrete Fourier transform.

The quantum Fourier transform converts the data in the base vector to the data in the amplitude under certain conditions and vice versa.

### 8.5.1 8.5.1 Basic definition

QFT can be obtained by simply substituting IDFT. Both QFT and DFT are essentially different forms of expression of the same vector in two equivalent spaces, i.e., the substitution of base vectors.

$$y_k \rightarrow \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{\frac{2\pi i}{N} jk}$$

$$|x\rangle \rightarrow \frac{1}{2^{\frac{n}{2}}} \sum_{k=0}^{2^n-1} e^{\frac{2\pi i}{2^n} xk} |k\rangle$$

Based on the definition, a certain vector  $\sum_x \alpha_x |x\rangle$  in the space  $\text{span}|x\rangle$  can be represented as the linear combination  $\sum_k \beta_k |k\rangle$  of base vectors in another equivalent space  $\text{span}|k\rangle$  through Fourier transform, and the coefficient  $\beta_k$  of the linear combination depends on  $|x\rangle$  and  $\alpha_k$ .

---

#### Note

The quantum Fourier transform/inverse transform can be essentially considered as a mutual transformation of amplitude and base vector.

---

### 8.5.2 8.5.2 Construction of quantum circuit

The implementation of quantum circuits of QFT requires the transformation of its expression to obtain the transformation process which can be implemented with the existing general quantum gate combination.

#### 8.5.2.1 8.5.2.1 Sum form and tensor product form of QFT

By any given integer  $x$ ,  $k = \sum_{i=1}^n k_i 2^{n-i}$  is expanded by the binary system, and the result of quantum Fourier transform of  $|x\rangle$  can be expressed as below:

$$\begin{aligned} QFT(|x\rangle) &= \frac{1}{2^{\frac{n}{2}}} \sum_{k=0}^{2^n-1} e^{\frac{2\pi i x k}{2^n}} |k\rangle = \frac{1}{2^{\frac{n}{2}}} \sum_{k_1=0}^1 \cdots \sum_{k_n=0}^1 e^{2\pi i x k (\sum_{l=1}^n k_l 2^{n-l})} |k_1 \cdots k_n\rangle \\ &= \frac{1}{2^{\frac{n}{2}}} \sum_{k_1=0}^1 \cdots \sum_{k_n=0}^1 \otimes_{l=1}^n e^{2\pi i x k_l 2^{n-l}} |k_l\rangle = \frac{1}{2^{\frac{n}{2}}} \otimes_{l=1}^n \left( |0\rangle + e^{2\pi i x 2^{n-l}} |1\rangle \right) \end{aligned}$$

As shown by the above formula, QFT can express the particular quantum state  $|x\rangle$  as a linear combination of another set of base vectors, and such linear combination can also be expressed as the tensor product of multiple single-qubit states  $\frac{1}{\sqrt{2}} \left( |0\rangle + e^{2\pi i x 2^{n-l}} |1\rangle \right)$ .

Therefore, for any given integer  $x$ , we, if able to construct a quantum state  $\frac{1}{\sqrt{2}} \left( |0\rangle + e^{2\pi i x 2^{-l}} |1\rangle \right)$  with binary expansion qubits, can complete the construction of corresponding QFT quantum circuits through the QFT expression in the form of tensor product.

### 8.5.2.2 Binary expansion and quantum state preparation

Binary expansion approximation for any given integer  $x$ :

$$\frac{x}{2^m} \approx \frac{[x_1 \dots x_m]}{2^m} = [0.x_1 \dots x_m] = \sum_{k=1}^m x_k 2^{-k}$$

while

$$2\pi i x 2^{-l} = 2\pi i [x_1 \dots x_n] 2^{-l} = 2\pi i [0.x_{n-l} \dots x_n]$$

*Then, the preparation of*

$\frac{1}{\sqrt{2}} \left( |0\rangle + e^{2\pi i x 2^{-l}} |1\rangle \right)$  is transformed into that of  $\frac{1}{\sqrt{2}} \left( |0\rangle + e^{2\pi i [0.x_{n-l} \dots x_n]} |1\rangle \right)$

It shall be noted that  $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{\sqrt{2}} \left( |0\rangle + e^{2\pi i [0.x_n]} |1\rangle \right)$  while

$$\frac{1}{\sqrt{2}} \left( |0\rangle + e^{2\pi i [0.x_{n-l} \dots x_n]} |1\rangle \right) = \frac{1}{\sqrt{2}} \left( |0\rangle + e^{2\pi i [0.x_{n-1}]} e^{2\pi i [0.x_n]} |1\rangle \right)$$

$$R_m|0\rangle = |0\rangle, \quad R_m|1\rangle = e^{2\pi i \frac{1}{2^m}} |1\rangle$$

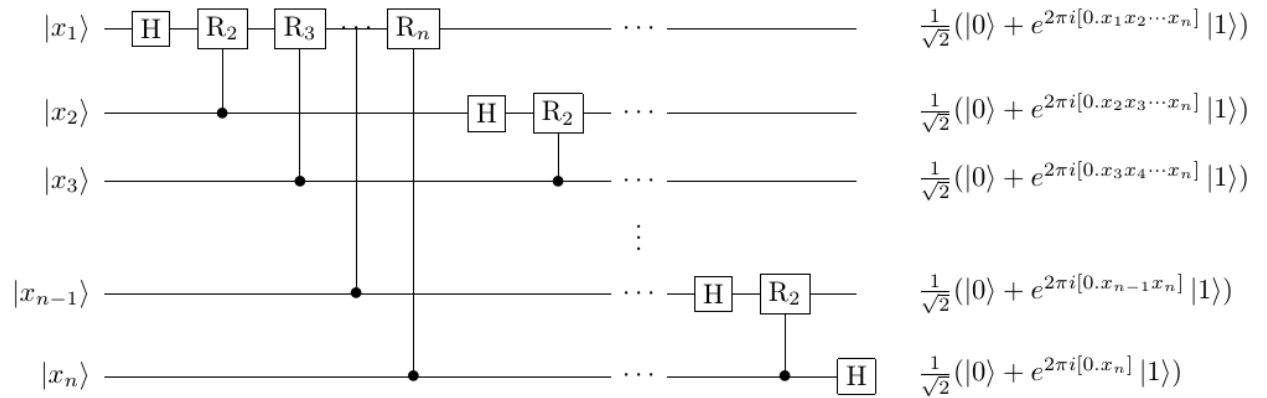
The defined controlled rotating quantum gate  $(C - R)_{j-k+1}$  meets

$$(C - R)_{j-k+1} \frac{1}{\sqrt{2}} \left( |0\rangle + e^{2\pi i [0.x_{n-j}]} |1\rangle \right) |x_{n-k}\rangle = \frac{1}{\sqrt{2}} \left( |0\rangle + e^{2\pi i [0.x_{n-j}]} |1\rangle \right)$$

*Therefore, the preparation of quantum state*

$\frac{1}{\sqrt{2}} \left( |0\rangle + e^{2\pi i x 2^{-l}} |1\rangle \right)$  can be achieved by using quantum gate H and  $(C - R)_{j-k+1}$  thus completing the quantum circuit of QFT.

The quantum circuit diagram of QFT is as below.



In particular, we have noticed that the resulting quantum state corresponding to the qubit with the initial quantum state being  $|x_i\rangle$  is  $\frac{1}{\sqrt{2}} \left( |0\rangle + e^{2\pi i x 2^{n+1-l}} |1\rangle \right)$  instead of  $\frac{1}{\sqrt{2}} \left( |0\rangle + e^{2\pi i x 2^{-l}} |1\rangle \right)$ . Thus, we need add multiple sets of SWAP gates in actual applications.

### 8.5.3 8.5.3 Code implementation

QFT in one dimension is a Hadamard quantum gate. The QFT interface function based on pyQPanda is as below:

```
QFT(qlist)
```

The example where  $|x\rangle = |000\rangle$  is taken to verify the code example of QFT is as below:

```
#!/usr/bin/env python

import pyqpanda as pq
from numpy import pi

if __name__ == "__main__":

    machine = pq.init_quantum_machine(pq.QMachineType.CPU)
    qvec = machine.qAlloc_many(3)
    prog = pq.create_empty_qprog()

    # Building quantum programs
    prog.insert(pq.QFT(qvec))

    # Perform probability measurements on quantum programs
    result = pq.prob_run_dict(prog, qvec, -1)
    pq.destroy_quantum_machine(machine)

    # Print measurement results
    for key in result:
        print(key+": "+str(result[key]))
```

According to the definition of QFT as given above and  $|x\rangle = |000\rangle$ , the output result be all the quantum states obtained based on the uniform probability of 1/8, i.e.:

```
000, 0.125
001, 0.125
010, 0.125
011, 0.125
100, 0.125
101, 0.125
110, 0.125
111, 0.125
```



## 8.6 Quantum phase estimation

Quantum phase estimation (QPE) can serve to compute the phase of the eigenvalue of a given unitary operator (U), i.e., solve  $\varphi$  in  $U|\psi\rangle = e^{2\pi i\varphi}|\psi\rangle$  where  $|\psi\rangle$  is the eigenvector of U.

The QPE in classical form is constructed on the basis of QFT

### 8.6.1 Overview of structure of quantum circuit

Suppose that the eigenvector  $|\Psi\rangle$  has been constructed. Quantum phase estimation includes the steps below:

- 1.The eigenvalue phase of U is decomposed and transferred to the amplitude of the auxiliary qubit through a series of special rotating quantum gate operations.
- 2.IQFT is conducted for the auxiliary qubit to transfer the decomposed eigenvalue phases on the amplitude to the base vectors.
- 3.The phase information of the eigenvalue can be obtained by measuring the base vectors of the auxiliary qubit.

For an eigen quantum state  $|\Psi\rangle$  of the unitary operator U, we can extract the eigenvalue phase corresponding to the quantum state to the amplitude through specific quantum gate combination, but it is hard to accurately and effectively measure the amplitude of the quantum state.

We have to integrate the eigenvalue phase data by virtue of quantum gate combinations, and can transfer the eigenvalue to the base vector by taking advantage of the function of IQFT to transfer amplitude to base vector.

---

#### Note

Quantum phase estimation is essentially used to extract the eigenvalue phases of unitary operators and output the phases in a form convenient for measurement.

---

### 8.6.2 Construction of quantum circuit

#### 8.6.2.1 Eigen quantum state and eigenvalue phase extraction

According to the definition of eigen quantum state  $U|\psi\rangle = e^{2\pi i\varphi}|\psi\rangle$

Therefore, the unitary operator U can define a controlled quantum gate (C-U) to enable

$$(C - U^{2^t}) (a|0\rangle + b|1\rangle) \otimes |\psi\rangle = (a|0\rangle + e^{2\pi i\varphi 2^t} b|1\rangle) \otimes |\psi\rangle$$

The eigenvalue phase  $\varphi$  can be extracted into the amplitude through such controlled transformation.

### 8.6.2.2 Transfer of eigenvalue phase from amplitude to base vector

We select a set of auxiliary qubits which are initialized to the maximum superposition state, and can extract the eigenvalue phase into the amplitude through the controlled quantum gate:

$$\left(C - U^{2^n}\right) \dots \left(C - U^{2^1}\right) \frac{1}{2^{\frac{n}{2}}} \otimes_{t=1}^n (|0\rangle + |1\rangle) = \left(|0\rangle + e^{2\pi i \varphi 2^{1-1}} |1\rangle\right) \dots \left(|0\rangle + e^{2\pi i \varphi 2^{n-1}} |1\rangle\right)$$

At this point, the form of quantum state in the auxiliary qubits is close to that of the resulting quantum state of QFT, and the following results can be obtained with the help of IQFT:

$$\begin{aligned} & \text{QFT}^{-1} \frac{1}{2^{\frac{n}{2}}} \otimes_{t=1}^n (|0\rangle + e^{2\pi i \varphi 2^{t-1}} |1\rangle) \\ &= \text{QFT}^{-1} \frac{1}{2^{\frac{n}{2}}} \sum_{k=0}^{2^n-1} e^{2\pi i \varphi k} |k\rangle \\ &= \frac{1}{2^n} \sum_{k=0}^{2^n-1} \sum_{x=0}^{2^n-1} e^{-\frac{2\pi i k}{2^n} (x - 2^n \varphi)} |x\rangle \end{aligned}$$

The quantum state obtained is measured and the measurement results can be divided into the two following categories:

1. Where the positive integer  $2^n \varphi \in \mathbb{Z}$  is available, can be obtained through measurement with a probability of  $|x\rangle = |2^n \varphi\rangle$ .

2. Otherwise, with probability of  $\frac{4}{\pi^2}$ , we can obtain the integer which is closest to  $2^n \varphi$ , thus to obtain the approximate solution.

from the integer which is closest to  $2^n \varphi$ ? (Tip: continued fraction expansion)

The measurement result is the approximate solution of phase  $\varphi$  of which the precision is related to the number of auxiliary qubits  $n$ .  $2^n \varphi \in \mathbb{Z}$  indicates that the number of auxiliary qubits is already greater than that of binary expansion decimal places of  $\varphi$  so that the exact solution can be gotten.

### 8.6.3 Quantum circuit diagram and code implementation

The quantum circuit diagram of QPE is as below.

As disclosed by the above definition, we can provide the function implementation of QPE directly based on QPanda-2.0.

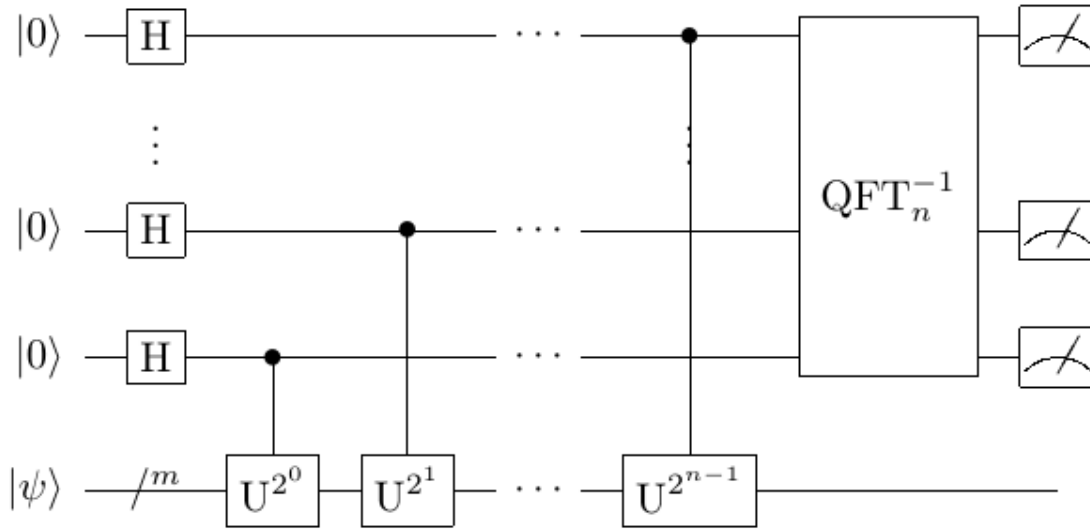
The quantum circuit can be divided into three parts, namely, eigen quantum state preparation, auxiliary qubit quantum state initialization, eigenvalue phase extraction and inverse quantum Fourier transform. The core contents of the program implementation are as follows:

```
#!/usr/bin/env python

import pyqpanda as pq
from numpy import pi

def QPE(controlqlist, targetqlist, matrix):
```

(continues on next page)



(continued from previous page)

```

circ = pq.QCircuit()
for i in range(len(controlqlist)):
    circ.insert(pq.H(controlqlist[i]))
for i in range(len(controlqlist)):
    circ.insert(controlUnitaryPower(targetqlist, controlqlist[controlqlist.
↪size() \
    - 1 - i], i, matrix))

circ.insert(pq.QFT(controlqlist).dagger())
return circ

```

The parameter matrix in the figure refers to the matrix corresponding to the unitary operator  $U$  requiring eigenvalue estimation.

When  $U = RY\left(\frac{\pi}{4}\right)$ ,  $|\psi\rangle = |0\rangle + i|1\rangle$  is selected, the eigenvalue is  $e^{-i\frac{\pi}{8}}$  and the code example for QPE verification is as below.

```

#!/usr/bin/env python

import pyqpanda as pq
from numpy import pi

if __name__ == "__main__":

    machine = pq.init_quantum_machine(pq.QMachineType.CPU)
    qvec = machine.qAlloc_many(1)
    cqv = machine.qAlloc_many(2)
    prog = pq.create_empty_qprog()

```

(continues on next page)

(continued from previous page)

```

# Building quantum programs
prog.insert(pq.H(cqv[0]))\
    .insert(pq.H(cqv[1]))\
    .insert(pq.S(qvec[0]))\
    .insert(pq.RY(qvec[0], pi/4).control(cqv[1]))\
    .insert(pq.RY(qvec[0], pi/4).control(cqv[0]))\
    .insert(pq.RY(qvec[0], pi/4).control(cqv[0]))\
    .insert(pq.QFT(cqv).dagger())

# Perform probability measurements on quantum programs
result = pq.prob_run_dict(prog, cq, -1)
pq.destroy_quantum_machine(machine)

# Print measurement results
for key in result:
    print(key+": "+str(result[key]))

```

As implied above, the output result should be the quantum state  $|0\rangle$  with a large probability

```

000, 0.821067
001, 0.0732233
010, 0.0324864
011, 0.0732233

```

## 8.7 8.7 For operations of quantum

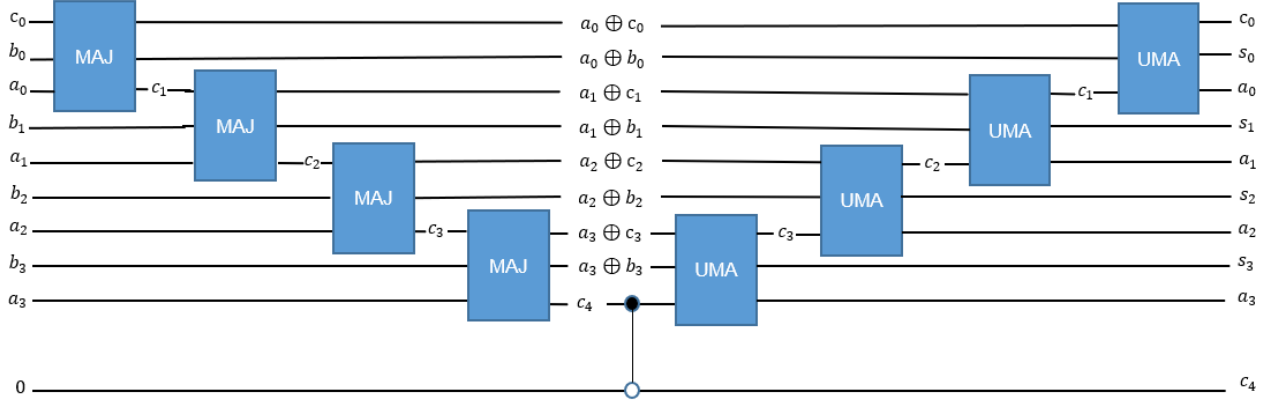
In particular cases, four basic operations shall be implemented in a quantum computer. The quantum adder and the four operations of quantum derived therefrom can meet the computing requirements

### 8.7.1 8.7.1 Background of adder algorithm

All quantum gate operations except measurement are unitary transformations, and thus the quantum circuit excluding measurement is reversible as a whole.

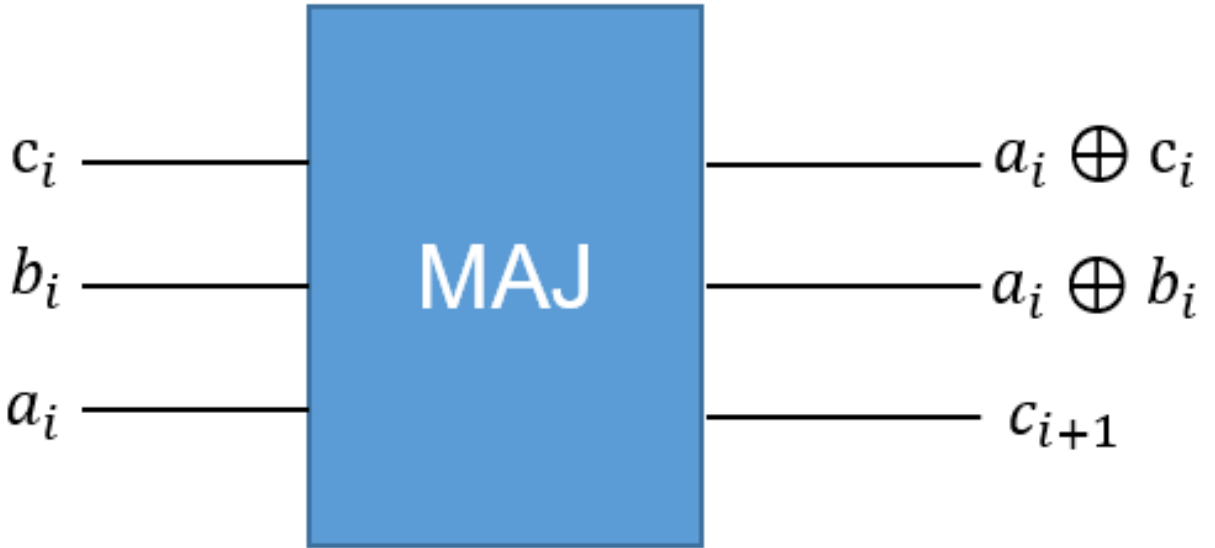
The quantum circuit of a quantum adder shall also be reversible, so that the inputs and outputs are an equal number of qubits. The quantum circuit diagram is shown below.

The figure above contains two quantum circuit modules MAJ and UMA which mainly serve to obtain the carry value and resulting value of the current binary qubit.



#### 8.7.1.1 Component of MAJ quantum circuit

The quantum circuit diagram of MAJ is as below.



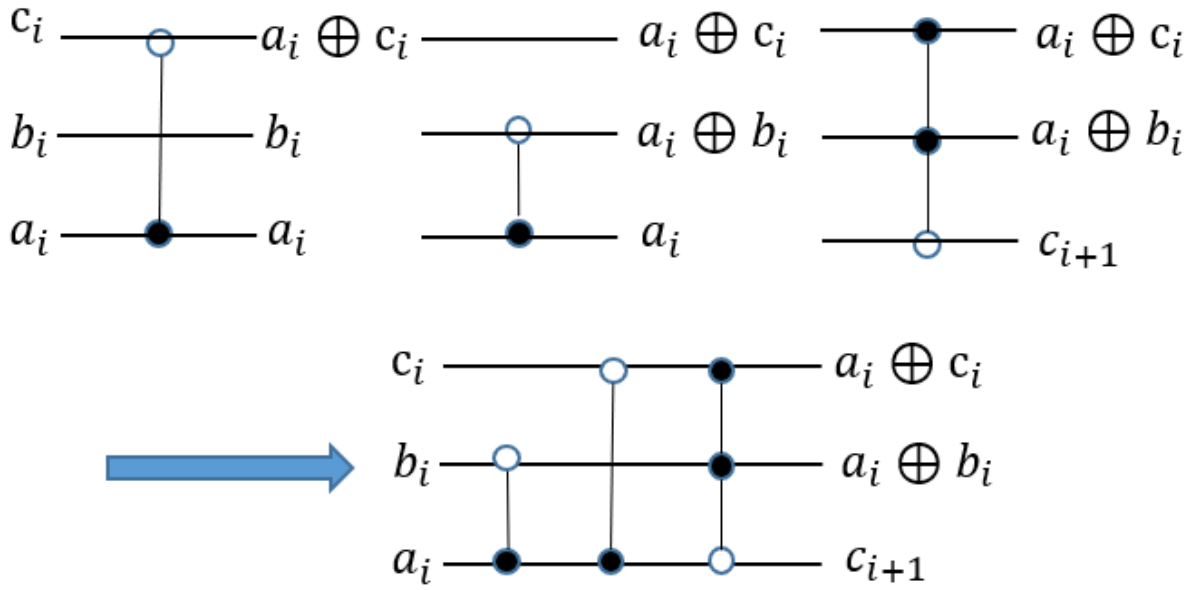
The specific functions of MAJ quantum circuits are explained as follows.

The inputs of MAJ quantum circuits are the carry value  $c_i$  of the previous qubit and the two values to be added ( $a_i$  and  $b_i$ ) of the current qubit while the outputs are  $a_i + c_i \bmod 2$ ,  $a_i + b_i \bmod 2$  and the carry value  $c_{i+1}$  of the current qubit.

The MAJ module is to achieve carry bits. We want to get carry qubit  $c_{i+1}$ , i.e., judge  $(a_i + b_i + c_i) / 2$  by starting from  $(a_i + b_i + c_i)$ .

We select a number  $a_i$  from the values to be added to enumerate the carry bits as follows:

1.  $a_i = 0$ ,  $c_i = [(a_i + b_i) \% 2] * [(a_i + c_i) \% 2]$ ;
2.  $a_i = 1$ ,  $c_i = ([ (a_i + b_i) \% 2 ] * [ (a_i + c_i) \% 2 ] + 1) \% 2$ ;



Therefore, we can judge the carry bits by only studying  $a_i$ ,

$$[(a_i + b_i) \% 2] * [(a_i + c_i) \% 2]$$

We can accurately judge the carry bits by starting from the existing quantum logic gate and preparing the quantum state  $a_i, [(a_i + b_i) \% 2], [(a_i + c_i) \% 2]$ . The subject of study selected here is not unique. Other schemes will result in corresponding quantum circuits.

The scheme for preparing three quantum states is shown in the figure above. We use CNOT gate to complete module 2 addition to obtain  $(a_i + b_i) \% 2$ ,  $(a_i + c_i) \% 2$ , and Toffoli gate to complete the XOR operation of  $a$  and  $[(a_i + b_i) \% 2] * [(a_i + c_i) \% 2]$ .

### 8.7.1.2 8.7.1.2 Component of UMA quantum circuit

The quantum circuit diagram of UMA is as below.

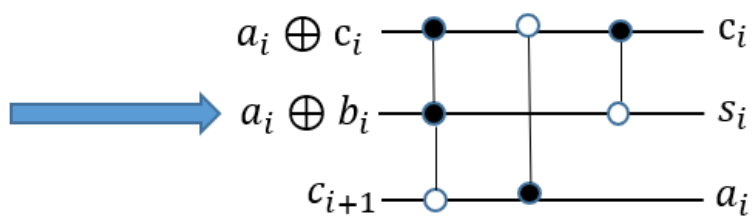
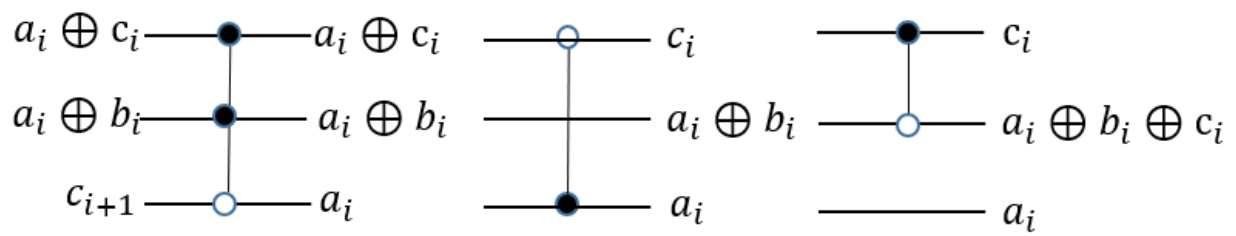
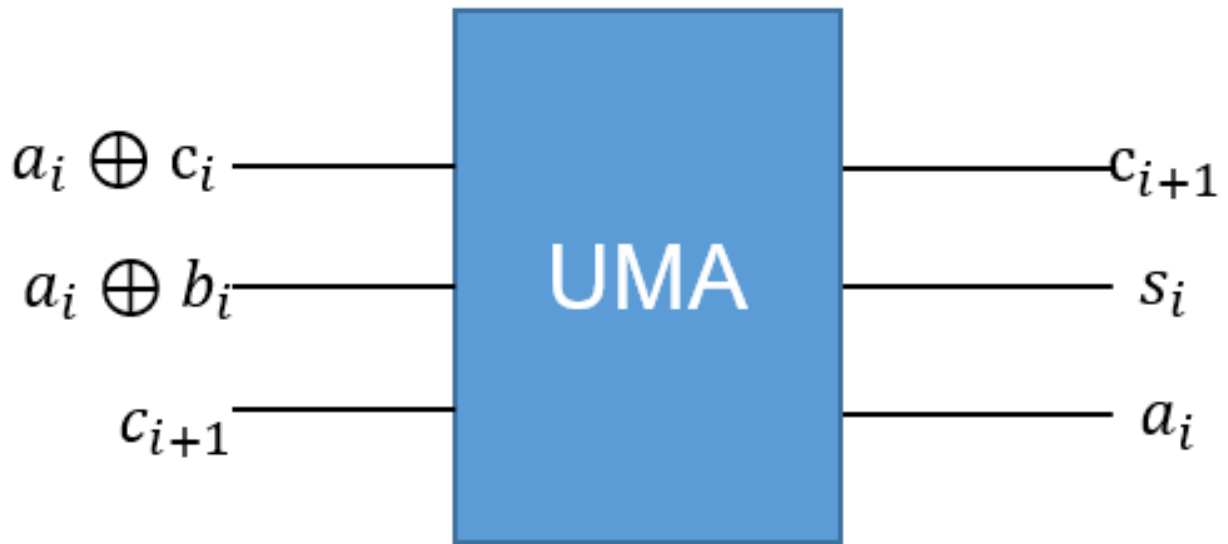
The specific functions of UMA quantum circuits are explained as follows.

The inputs of UMA quantum circuits are  $a_i + c_i \bmod 2$ ,  $a_i + b_i \bmod 2$ , and the carry value  $c_{i+1}$  of the current bit while the outputs are  $c_i, a_i + b_i + c_i \bmod 2 := s_i$  and  $a_i$ .

The UMA module is to achieve the results of current bits. We want to get current bit  $s_i$ , i.e.,  $(a_i + b_i + c_i) \% 2$ .

By referring to the MAJ module, we get  $a_i$  from  $c_{i+1}$  through the Toffoli gate which is  $c_{i+1}$  through Toffoli gate which is completely opposite to that used by MAJ, and then get  $c_i$  through CNOT transformation which is opposite to that used by MAJ, thus to get  $(a_i + b_i + c_i) \% 2$  through simple CNOT gate in combination with the existing  $a_i + b_i \bmod 2$ .

The first two steps of the whole process can be considered as the reverse transform of the corresponding quantum gate of MAJ.



---

**Note**

The implementation of quantum circuits of MAJ is not unique, so is UMA?

---

## 8.7.2 For operations of quantum

### 8.7.2.1 Quantum adder

The principle of a quantum adder is described as above.

### 8.7.2.2 Quantum subtracter

A basic adder only supports addition of non-negative integers. For decimals, the addends  $a$  and  $b$  which are required to be inputted must have the same decimal places and the same length upon decimal point alignment.

For a quantum addition with sign reversing, additional auxiliary qubits are required to record the sign bit. With any two target quantum states  $A$  and  $B$  given, specific complementary operation is performed for the second quantum state  $B$  which is then converted to  $A - B = A + (-B)$  where  $-B$  is not implemented by flipping the sign bit.

The specific complementary operation is as follows: the sign qubit will remain unchanged if it is positive and will be flipped plus 1 if it is negative. Therefore, an additional auxiliary qubit is required to control whether to conduct complementary operation.

A quantum subtracter is essentially the signed version of a quantum adder.

### 8.7.2.3 Quantum multiplier

The quantum multiplier is completed based on the adder. The multiplier  $A$  is selected as the controlled qubit and the multiplier  $B$  as the control qubit by binary expansion bit by bit, and also the operation result of the controlled adder is added up to the auxiliary qubit. Upon each controlled addition as controlled by  $B$ , the multiplier  $A$  is moved by one place to the left with zero added at the final place.

The values output by the controlled addition are then added up in the auxiliary qubits to obtain the multiplication result.

### 8.7.2.4 Quantum divider

The quantum divider is completed based on the quantum subtracter. We complete the number comparison by checking whether the sign bit of dividend changes after subtraction and determine whether to terminate the division.

If the dividend is subtracted from the divisor, the quotient shall be plus 1. After each subtraction, we re-compare the dividend and divisor until the dividend is divisible or the preset precision is reached.

Consequently, we need an additional auxiliary qubit to store the precision parameter.



## 8.7.3 8.7.3 Code implementation and use instructions

### 8.7.3.1 8.7.3.1 Quantum adder

The interface functions of adder in pyQPanda are as below:

```
QAdder (adder1, adder2, c, is_carry)

QAdderIgnoreCarry (adder1, adder2, c)

QAdd (adder1, adder2, k)
```

The difference between the first two interface functions lies in whether to retain the carry bit (`is_carry`), but both only support additions of positive numbers. The `adder1` and `adder2` among the parameters are the qubits which perform addition and in exactly the same format, and `c` is the auxiliary qubit.

The third interface function is the signed adder, which is implemented based on the quantum subtracter. Sign bits are added to the numbers to be added and the corresponding auxiliary qubits change from  $1 - 2$  single-qubits to an `adder1.size() + 2` qubit.

The output bits of addition are all `adder1` and other not-carry qubits remain unchanged.

### 8.7.3.2 8.7.3.2 Quantum subtracter

The quantum subtracter is completed based on the basic adder and is the basis of the signed adder.

The interface function of subtracter (signed adder) in pyQPanda is as below:

```
QSub (a, b, k)
```

The highest bit of the qubits of the two numbers to be subtracted is the sign bit and the auxiliary bit  $k \cdot \text{size}() = a \cdot \text{size}() + 2$ , which is the same as the signed adder.

The output qubits of subtraction are `a` and other qubits remain unchanged.

### 8.7.3.3 8.7.3.3 Quantum multiplier

The interface functions of multiplier in pyQPanda is as below:

```
QMultiplier (a, b, k, d)

QMul (a, b, k, d)
```

The input qubits to be multiplied of both the interface functions contain signed bits, but only `QMul` supports signed multiplications.

Accordingly, in QMultiplier, the auxiliary qubit  $k \cdot \text{size}() = a \cdot \text{size}() + 1$ , and the resulting qubit  $d \cdot \text{size}() = 2 \cdot a \cdot \text{size}()$ .

In QMul, the auxiliary qubit  $k \cdot \text{size}() = a \cdot \text{size}()$ , and the resulting qubit  $d \cdot \text{size}() = 2 \cdot a \cdot \text{size}() - 1$ .

The output qubits of multiplication are all d and other qubits remain unchanged.

If the input qubits a and b with equal length have any decimal point, the position coordinates of the decimal point in the output qubit d double those in the input qubits.

#### 8.7.3.4 Quantum divider

The interface functions of divider in pyQPanda are as below:

```
QDivider(a,b,c,k,t)

QDivider(a,b,c,k,f,s)

QDiv(a,b,c,k,t)

QDiv(a,b,c,k,f,s)
```

Similar with the multiplier, the divider is divided into two categories. Although the input qubits to be operated have a signed bit, the interfaces include signed operations and positive-only operations.

k is the auxiliary qubit, and t or s is the classical bit that limits the number of QWhile loops.

Moreover, the divider has the problem of indivisibility. Thus, it is provided with the above four kinds of interface functions and their corresponding input and output parameters show the following properties respectively:

1. When QDivider returns the remainder and quotient (stored in a and c respectively),  $c \cdot \text{size}() = a \cdot \text{size}()$ , but  $k \cdot \text{size}() = a \cdot \text{size}() * 2 + 2$ ;
2. When QDivider returns the precision and quotient (stored in f and c respectively),  $c \cdot \text{size}() = a \cdot \text{size}()$ , but  $k \cdot \text{size}() = 3 \cdot \text{size}() * 2 + 5$ ;
3. When QDiv returns the remainder and quotient (stored in a and c respectively),  $c \cdot \text{size}() = a \cdot \text{size}()$ , but  $k \cdot \text{size}() = a \cdot \text{size}() * 2 + 4$ ;
4. When QDiv returns the precision and quotient (stored in f and c respectively),  $c \cdot \text{size}() = a \cdot \text{size}()$ , but  $k \cdot \text{size}() = a \cdot \text{size}() * 3 + 7$ ;

If the parameters fail to satisfy the number of qubits required by the four operations of quantum, the computing will continue but the result will overflow.

The output qubits of division are c, and a, b and k in the division with precision remain unchanged. Otherwise, b and k remain unchanged but the remainder is stored in a.

## 8.7.4 8.7.4 Example

Below is a simple code example for calling the four operations of quantum based on pyQPanda.

```
#!/usr/bin/env python

import pyqpanda as pq
# from numpy import pi

if __name__ == "__main__":
    # To save qubits, auxiliary qubits will be borrowed from each other
    qvm = pq.init_quantum_machine(pq.QMachineType.CPU)

    qdivvec = qvm.qAlloc_many(10)
    qmulvec = qdivvec[:7]
    qsubvec = qmulvec[:-1]
    qvec1 = qvm.qAlloc_many(4)
    qvec2 = qvm.qAlloc_many(4)
    qvec3 = qvm.qAlloc_many(4)
    cbit = qvm.cAlloc()
    prog = pq.create_empty_qprog()

    # (4/1+1-3)*5=10
    prog.insert(pq.bind_data(4,qvec3)) \
        .insert(pq.bind_data(1,qvec2)) \
        .insert(pq.QDivider(qvec3, qvec2, qvec1, qdivvec, cbit)) \
        .insert(pq.bind_data(1,qvec2)) \
        .insert(pq.bind_data(1,qvec2)) \
        .insert(pq.QAdd(qvec1, qvec2, qsubvec)) \
        .insert(pq.bind_data(1,qvec2)) \
        .insert(pq.bind_data(3,qvec2)) \
        .insert(pq.QSub(qvec1, qvec2, qsubvec)) \
        .insert(pq.bind_data(3,qvec2)) \
        .insert(pq.bind_data(5,qvec2)) \
        .insert(pq.QMul(qvec1, qvec2, qvec3, qmulvec)) \
        .insert(pq.bind_data(5,qvec2))

    # Perform probability measurements on quantum programs
    result = pq.prob_run_dict(prog, qmulvec,1)
    pq.destroy_quantum_machine(qvm)

    # Print measurement results
    for key in result:
        print(key+": "+str(result[key]))
```

The computing performed is  $(4/1 + 1-3)*5 = 10$ , and thus the result should be  $|10\rangle$  (i.e.,  $|1010\rangle$ ) with the probability of 1.

1010:1

## 8.8 HHL algorithm

The HHL algorithm is a quantum algorithm used to solve linear equations which are widely used in many fields.

### 8.8.1 Overview of background

The problem of linear equations can be defined as follows: with matrix  $A \in C^{N \times N}$  and vector  $\vec{b} \in C^N$  given, find  $\vec{x} \in C^N$  to satisfy  $A\vec{x} = \vec{b}$ .

The system of linear equations is called sparse system of linear equations if matrix A has at most s non-zero elements per row or column. The classical algorithm (conjugate gradient method) is used to solve N-dimensional sparse system of linear equations. The time complexity required is  $O(Nsk \log(\frac{1}{\epsilon}))$  where K represents the number of conditions of the system and  $\epsilon$  means the approximation precision. HHL is A quantum algorithm. In case that A is self-conjugate matrix, the time complexity of solving linear equations with HHL algorithm is  $O(\log(N)s^2 \frac{k^2}{\epsilon})$ .

The HHL algorithm is exponentially faster than the classical algorithm, but the classical algorithm can give exact solutions while HHL can only return approximate ones.

---

#### Note

The HHL algorithm is a pure quantum algorithm. The emergence of HHL and its improved version are of great significance to prove the practicability of quantum algorithms.

---

### 8.8.2 Principle of algorithm

The HHL algorithm can be used to solve the system of linear equations subject to a certain format conversion. It mainly includes the following three steps and requires the use of three registers, i.e., right-hand item qubit, storage qubit and auxiliary qubit.

We construct the right-hand item quantum state, perform phase estimation for the parameters of the storage qubit and the right-hand item qubit including the left-hand item matrix, and transfer all the integer eigenvalues of the left-hand item matrix to the base vector of the storage qubit.

We rotate a series of parameters including eigenvalues in a controller manner to find out all the quantum states related to the eigenvalues and transfer the eigenvalues from the base vector storing qubits to the amplitude.

We conduct inverse phase estimation for the eigen storage qubit and the right-hand item qubit, and integrate the eigenvalue on the amplitude of the storage qubit into the right-hand item qubit. When the measurement of auxiliary qubit reaches a specific state, we can get the quantum state of the solution on the right-hand item qubit.

Before proceeding to the specific steps of the algorithm, we should perform specific transformation to solve the system of linear equations in classical form  $A\vec{x} = \vec{b}$ :

Assume that the matrix A is self-conjugate without loss of generality. Otherwise, take

$$C_A = \begin{bmatrix} 0 & A \\ A^H & 0 \end{bmatrix}, C_b = \begin{bmatrix} b \\ 0 \end{bmatrix}, C_x = \begin{bmatrix} 0 \\ x \end{bmatrix}$$

So that  $C_A \vec{C}_x = \vec{C}_b$  is satisfied and also satisfy  $C_{\{A\}}$  self-conjugation.

In the contents below, A will be defaulted as a self-conjugate matrix.

The vector  $\vec{b}$  and  $\vec{x}$  are mapped to the quantum states  $|b\rangle$  and  $|x\rangle$  respectively by coding to the amplitude after normalization, and the original problem is converted into  $A|x\rangle = |b\rangle$ .

Matrix A is subject to spectral decomposition to get

$$A = \sum_{j=0}^{N-1} \lambda_j |u_j\rangle \langle u_j|, \quad \lambda_j \in R$$

Where  $\lambda_j$  and  $u_j$  are the eigenpair (eigenvalue and corresponding eigenvector) of matrix A.

$|b\rangle$  is expanded as an eigen vector base to get

$$|b\rangle = \sum_{j=0}^{N-1} b_j |u_j\rangle, \quad b_j \in C$$

Then, the solution of the original system of equations can be written as below:

$$|x\rangle = A^{-1} |b\rangle = \sum_{j=0}^{N-1} \lambda_j^{-1} b_j |u_j\rangle$$

Obviously, the basic idea of the algorithm should be constructing the quantum state  $|x\rangle$  by starting from the right-hand item quantum state  $|b\rangle$ .

### 8.8.2.1 8.8.2.1 Extraction of eigenvalue through QPE

The eigenvalue extraction shall be completed in order to extract the eigenvalue of matrix A to the amplitude of the solution quantum state. As shown above, the QPE quantum circuit can be used for eigenvalue extraction.

A QPE operation is performed to  $|0\rangle^{\otimes n} |b\rangle$  to get

$$\text{QPE}(|0\rangle^{\otimes n} |b\rangle) = \sum_{j=0}^{N-1} b_j |\tilde{\lambda}_j\rangle |u_j\rangle$$

Where  $\tilde{\lambda}_j$  is the approximate integer of the corresponding eigenvalue  $\lambda_j$ . The details are shown in QPE. Thus, the eigenvalue information of matrix A is stored in the base vector  $|\tilde{\lambda}_j\rangle$ .

### 8.8.2.2 8.8.2.2 Transfer of eigenvalue through controlled rotation

We construct the following controlled rotation  $CR(k)$

$$CR(k)(|a\rangle|j\rangle) = \begin{cases} RY\left(\arccos\frac{C}{k}\right)|a\rangle|k\rangle, & j = k \\ |a\rangle|j\rangle, & j \neq k \end{cases}$$

### 8.8.2.3 8.8.2.3 Output of resulting quantum state through inverse QPE

In theory, the quantum state subject to controlled rotation can be able to get the quantum state of the solution  $|x\rangle$  through measurement.

However, to avoid the quantum state  $\frac{C}{\lambda_j}b_j|1\rangle|\tilde{\lambda}_j\rangle|u_j\rangle$  which is provided with the same  $|u_j\rangle$  but different  $|\tilde{\lambda}_j\rangle$  and requires merging, we shall choose inverse QPE operation to get the resulting quantum state in the form of  $\frac{C}{\lambda_j}b_j|1\rangle|\tilde{\lambda}_j\rangle|u_j\rangle$ .

Inverse QPE operation is performed to the rotating result to get

$$\begin{aligned} & (I \otimes QPE^\dagger) \sum_{j=0}^{N-1} \left( \sqrt{1 - \frac{C^2}{\tilde{\lambda}_j^2}}|0\rangle + \frac{C}{\tilde{\lambda}_j}|1\rangle \right) b_j |\tilde{\lambda}_j\rangle |u_j\rangle \\ &= \sum_{j=0}^{N-1} \left( b_j \sqrt{1 - \frac{C^2}{\tilde{\lambda}_j^2}}|0\rangle|0\rangle|u_j\rangle + b_j \frac{C}{\tilde{\lambda}_j}|1\rangle|0\rangle|u_j\rangle \right) \end{aligned}$$

In fact, the resulting quantum state in this form, despite of an error, is still not be able to get the quantum state  $|x\rangle = \sum_{j=0}^{N-1} \lambda_j^{-1} b_j |u_j\rangle$  of the solution with probability of 1 when the first and the second quantum registers are  $|1\rangle$  and  $|0\rangle$  respectively.

---

#### Note

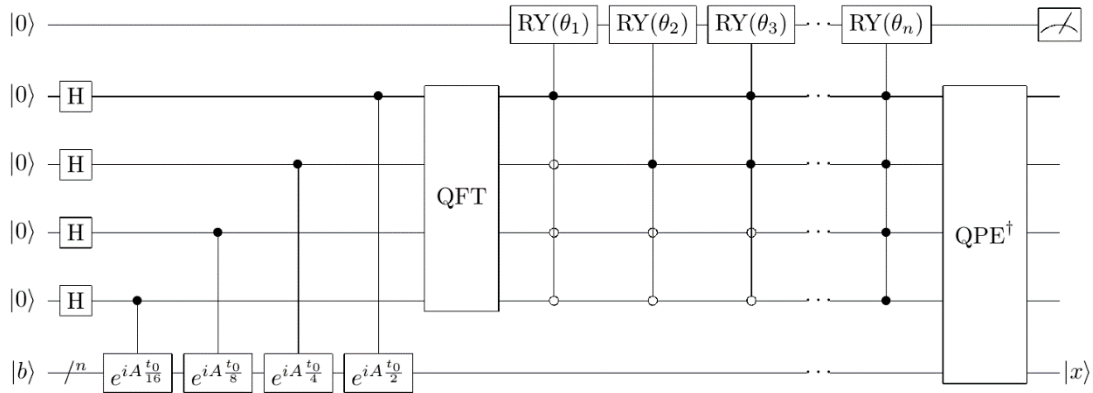
The HHL algorithm, by taking full advantage of the function of extracting eigenvalue information through quantum phase estimation, cleverly constructs a controlled rotating gate to capture eigenvalue from the base vector of the stored qubit and store it into the amplitude before restoring the stored qubit through inverse phase estimation thus to obtain the solution of the equation for which the amplitude contains eigenvalue.

---

### 8.8.3 8.8.3 Quantum circuit diagram and reference code

The quantum circuit diagram of HHL is as below.

The code implementation of HHL algorithm based on pyQPanda is quite lengthy, which will not be detailed here. The details are given in the HHL algorithm program source code under pyQPanda. Only several HHL algorithm calling interfaces provided in pyQPanda are introduced here.



```
HHL(matrix, data, QuantumMachine)

HHL_solve_linear_equations(matrix, data)
```

he first function interface is used to get the quantum circuit corresponding to the HHL algorithm while the second can input the matrix and the right-hand item of QStat format to return the solution vector.

We select the simplest two-dimensional left-hand item identity matrix example to verify the availability of HHL interface function, with the code example as follows:

```
#!/usr/bin/env python

import pyqpanda as pq
import numpy as np

if __name__ == "__main__":

    machine = pq.init_quantum_machine(pq.QMachineType.CPU)
    prog = pq.create_empty_qprog()

    # Building quantum programs
    prog.insert(pq.build_HHL_circuit([1, 0, 0, 1], [0.6, 0.8], machine))

    pq.directly_run(prog)

    result = np.array(machine.get_qstate())[:2]
    pq.destroy_quantum_machine(machine)

    # Print measurement results
    for key in result:
        print(key)
```

The output result should be  $[0.6, 0.8]$  same as the right-hand item vector because of minor disturbance of errors:

```
(0.5999999999999983+0j)
(0.79999999999999774+0j)
```

## 8.9 Grover algorithm and Quantum Counting algorithm

Both the Quantum Counting algorithm and Grover algorithm are derived from the division of set elements (into two categories). The Quantum Counting algorithm can get the number of the both types of elements in the set while the Grover algorithm can get one element of a specified type.

### 8.9.1 Overview of background

The previous study herein has introduced the problems of amplitude amplification quantum circuits and division of set elements into two categories, implying that, for a given finite set and the classification standard and  $f$ , we can represent the set elements with the following quantum states:

$$|\psi\rangle = \sin \theta |\varphi_1\rangle + \cos \theta |\varphi_0\rangle, |\varphi_0\rangle = |\varphi_1^\perp\rangle$$

Now, we perform two extensions to this problem.

#### 8.9.1.1 Quantum Counting

With  $|\Omega| = N = 2^n$ ,  $\Omega \supseteq B$ ,  $|B| = M \leq N$  given, the discrimination function satisfies:

$$\begin{cases} f : \Omega \rightarrow \{0, 1\} \\ f(x) = \begin{cases} 1, x \in B \\ 0, x \notin B \end{cases} \end{cases}$$

Find  $M$ .

The traditional algorithm simply performs ergodic counting through  $O(N)$  operation to obtain the cardinal number of the set  $M$ . The time complexity of Quantum Counting algorithm is exactly the same as that of QPE, which is expressed as  $O\left((\log_2 N)^2\right)$ .

---

#### Note

The amplitude amplification operator applied to the QPE circuit can play a filtering and extraction role which is similar to the extraction of eigenvalue from the eigen quantum state.

---



### 8.9.1.2 8.9.1.2 Search for solution elements

In the set  $\Omega$ , there is an element  $\omega \in \Omega$  which is the solution of a specific problem, the discriminant function is defined as below:

$$\begin{cases} f : \Omega \rightarrow \{0, 1\} \\ f(x) = \begin{cases} 1, x = \omega \\ 0, x \neq \omega \end{cases} \end{cases}$$

Find  $\omega \in \Omega$

The process of Grover algorithm is exactly the same as that of the amplitude amplification quantum circuit. The time complexity of Grover algorithm is  $O(\sqrt{N})$  which is greatly improved compared with  $O(N)$  of the classical algorithm.

---

#### Note

In fact, the idea of obtaining the approximate solution of amplitude and base vector through amplitude amplification is not limited to the division of set elements into two categories.

---

## 8.9.2 8.9.2 Principle of algorithm

The quantum states of set elements to be prepared by the two algorithms are of similar forms as below:

$$|\psi\rangle = \sin \theta |\varphi_1\rangle + \cos \theta |\varphi_0\rangle, |\varphi_0\rangle = |\varphi_1^\perp\rangle$$

However, their specific definitions and the targets to be solved are different. Thus, the algorithm principles derived from amplitude amplification quantum circuit are different, too.

### 8.9.2.1 8.9.2.1 QPE process based on amplitude amplification operator

The two basis quantum states in the Quantum Counting algorithm are defined on the basis of the set and discriminant function, i.e

$$|\varphi_0\rangle = \frac{1}{\sqrt{N-M}} \sum_{x \notin B} |x\rangle, |\varphi_1\rangle = \frac{1}{\sqrt{M}} \sum_{x \in B} |x\rangle$$

To convert the problem to the space  $\{|\varphi_0\rangle, |\varphi_1\rangle\}$ , we might consider  $\sin \theta = \frac{\sqrt{M}}{\sqrt{N}}$ , then we need to solve .

The amplitude amplification operator  $G = \begin{bmatrix} \cos 2\theta & -\sin 2\theta \\ \sin 2\theta & \cos 2\theta \end{bmatrix}$  is directly defined in the space  $\{|\varphi_0\rangle, |\varphi_1\rangle\}$ .

The following equation is satisfied:

$$G(\cos \theta |\varphi_0\rangle + \sin \theta |\varphi_1\rangle) = \cos 3\theta |\varphi_0\rangle + \sin 3\theta |\varphi_1\rangle$$

The eigen vector of amplitude amplification operator  $G$  can constitute a set of base vectors of space  $\{|\varphi_0\rangle, |\varphi_1\rangle\}$ , and thus  $\Psi$  can be decomposed into the linear combination of the eigen vector.

The eigenvalue of  $G$  is  $e^{\pm 2i\theta}$ . By virtue of the index qubit used in the preparation process of  $\Psi$ , we can accurately distinguish the corresponding eigen phase of the QPE process result constructed with  $G$  being 2 or  $2-2$ .

The solution to can then be completed by the QPE process based on  $G$ . With  $N$  given, the solution to  $M$  can be obtained.

---

### Note

Why can we determine that the eigen vector of amplitude amplification operator  $G$  can constitute a set of base vectors of space  $\{|\varphi_0\rangle, |\varphi_1\rangle\}$ ?

---

For the given quantum state  $|\psi\rangle = \sin\theta |\varphi_1\rangle + \cos\theta |\varphi_0\rangle$ , we can directly refer to the amplitude amplification quantum circuit and give Grover operator, thus obtaining

$$|\psi_k\rangle = \sin(2k+1)\theta |\varphi_1\rangle + \cos(2k+1)\theta |\varphi_0\rangle, (2k+1)\theta \approx \frac{\pi}{2}$$

However, the Grover operator  $G = -(I - 2|\omega\rangle\langle\omega|)(I - 2|\psi\rangle\langle\psi|)$  constructed directly through mirror transform involves large computing amount in actual programming realization and operation process. Therefore, we shall consider how to implement multiplication by using basic general quantum gates.

The original problem is converted into the space  $\{|\omega\rangle, |\psi\rangle\}$  left | Omegaright |=  $N'$ , and it can be known from  $\langle\varphi|\omega\rangle = \frac{1}{\sqrt{N}}, \langle\varphi|\varphi\rangle = 1$  that

$$U_\omega = (I - 2|\omega\rangle\langle\omega|) = \begin{bmatrix} -1 & -\frac{2}{\sqrt{N}} \\ 0 & 1 \end{bmatrix}, U_s = 2|\varphi\rangle\langle\varphi| - I = \begin{bmatrix} -1 & 0 \\ \frac{2}{\sqrt{N}} & 1 \end{bmatrix}$$

Let  $\sin\theta = \frac{1}{\sqrt{N}}, a = e^{i\theta}, \frac{1}{\sqrt{N}} = \frac{a-a^{-1}}{2i}$ , then

$$U_\omega U_s = \frac{1}{a^2+1} \begin{bmatrix} -i & i \\ a & a^{-1} \end{bmatrix} \begin{bmatrix} a^2 & 0 \\ 0 & a^{-2} \end{bmatrix} \begin{bmatrix} i & a \\ -a^2 i & a \end{bmatrix}$$

Let  $Q = U_s U_\omega$ , then  $Q|\varphi\rangle = \frac{N-4}{N}|\varphi\rangle + \frac{2}{\sqrt{N}}|\omega\rangle$  and

$$Q^k = \frac{1}{a^2+1} \begin{bmatrix} -i & i \\ a & a^{-1} \end{bmatrix} \begin{bmatrix} a^{2k} & 0 \\ 0 & a^{-2k} \end{bmatrix} \begin{bmatrix} i & a \\ -a^2 i & a \end{bmatrix}$$

Upon performing quantum gate  $Q^k$ , we measure the first register to get the probability of quantum state  $|\omega\rangle$  as

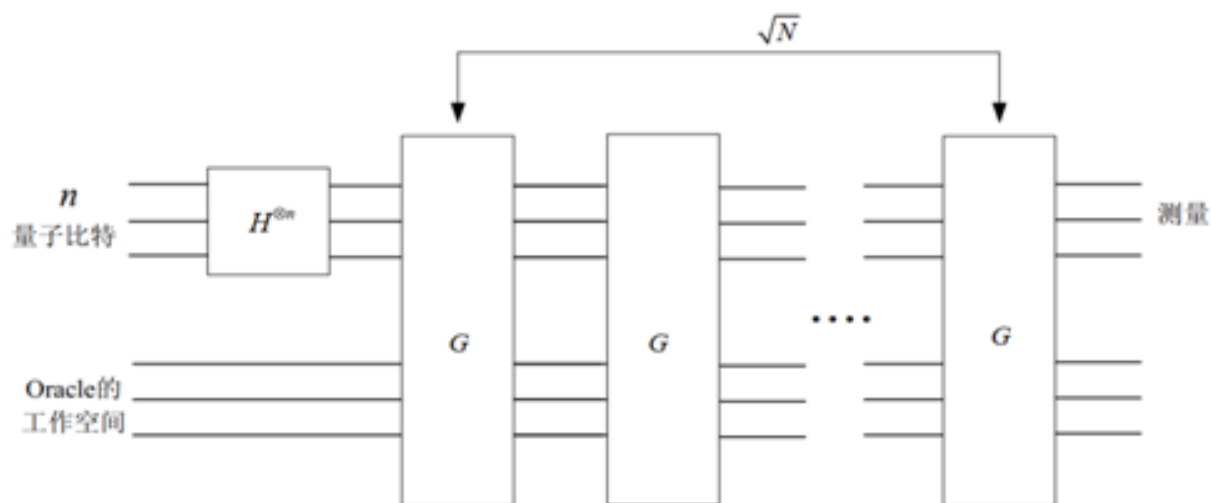
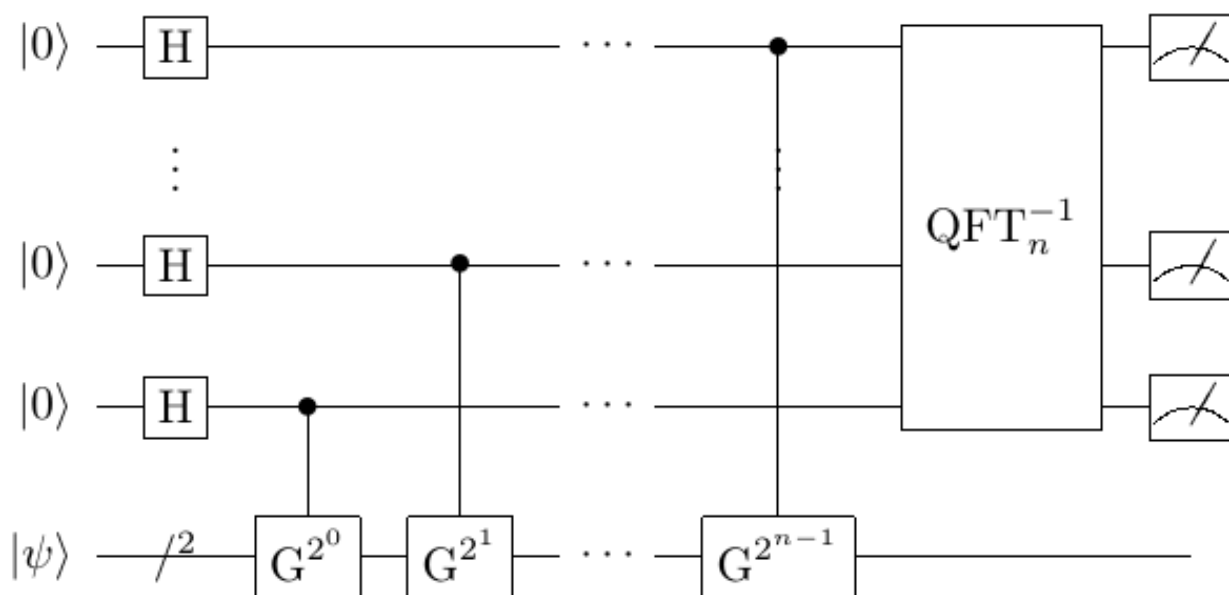
$$P(\omega) = \langle\omega|Q^k|\varphi\rangle = \begin{bmatrix} \langle\omega|\omega\rangle & \langle\omega|\varphi\rangle \end{bmatrix} (U_s U_\omega)^k \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{a^{2k+1} - a^{-(2k+1)}}{2i} = \sin((2k+1)\theta)$$

According to the solution of  $(2k+1)\theta = \frac{\pi}{2}$ , we can get the solution  $|\omega\rangle$  with the probability of approaching 1 through measurement after  $k = \left\lceil \frac{\pi}{4} \arcsin^{-1} \frac{1}{\sqrt{N}} - \frac{1}{2} \right\rceil \approx O(N)$  Q quantum gate operations.

### 8.9.3 Quantum circuit diagram and reference code

The core of Quantum Counting algorithm and Grover algorithm is the amplitude amplification operator, and the algorithm structure is basically consistent with that of QPE and amplitude amplification quantum circuit.

The quantum circuit diagram of Quantum Counting algorithm is as below.



The quantum circuit diagram of Grover algorithm is as below.

The process of implementing Quantum Counting algorithm based on pyQPanda is almost the same as the QPE process, and thus the source code is combined with the Grover algorithm. The program implementation of the two algorithms is shown in the program source code of Quantum Counting algorithm and Grover algorithm under pyQPanda.

The following is an introduction to an interface function and a example code implementation of the Grover algorithm based on pyQPanda. The program example of Quantum Counting algorithm will not be repeated here as it has no essential difference with the code implementation of QPE.

---

### Note

The experimental state preparation based on the set  $\omega$  and the discriminant function  $F$  is an important premise of both algorithms, and, together with the amplitude amplification operator, constitutes the core component of the algorithms.

---

```
Grover(data, Classical_condition, QuantumMachine, qlist, data)
```

The input parameters are algorithm search space, search condition, quantum simulator, output result storage qubit and number of iteration(s), with an executable Grover quantum circuit returned. The Grover algorithm also has other interface functions which will not be described here.

Below is a one-dimensional Grover example program code.

```
#!/usr/bin/env python

import pyqpanda as pq
import numpy as np

if __name__ == "__main__":

    machine = pq.init_quantum_machine(pq.QMachineType.CPU)
    x = machine.cAlloc()
    prog = pq.create_empty_qprog()

    data=[3, 6, 6, 9, 10, 15, 11, 6]
    grover_result = pq.Grover_search(data, x==6, machine, 1)

    print(grover_result[1])
```

The output results are the coordinates of the number 6 in the list, as shown below:

```
[1, 2, 7]
```

## 8.10 8.10 Shor' s Algorithm

Shor' s Algorithm, also known as prime factorization algorithm, plays an important role in breaking RSA encryption.

### 8.10.1 8.10.1 Background of problem

Given a large integer  $N = pq$  where  $p$  and  $q$  are unknown primes, solve  $p$  and  $q$ . Shor' s Algorithm includes three parts: solving common divisor implemented by the classical algorithm, converting prime factorization into periodic solution of function, and periodic solution of function implemented by such quantum algorithms as quantum Fourier transform.

Compared with the classical algorithm, Shor' s Algorithm greatly reduces the computing resource consumption and computing time complexity, making it possible for the quantum algorithm to solve the super-large mass factor decomposition problem which cannot be solved by the classical algorithm.

---

#### Note

The computing time and space resources theoretically required by the solving of RSA problem of extremely large number of qubits that Shor' s Algorithm tries to solve are almost unsatisfied by using the classical algorithm. In addition to reflecting the relative advantages of quantum computing, Shor' s Algorithm reveals the irreplaceability and absolute advantages of quantum computing on specific problems.

---

### 8.10.2 8.10.2 Principle of algorithm

The specific steps of Shor' s decomposition algorithm are as below:

1.  $\forall 1 < x < N, x \in \mathbb{Z}$ ;
2.  $\gcd(x, N) \neq 1$ ;
3. Finding  $r$  makes  $x^r \bmod N \equiv 1$ ;
4.  $r \bmod 2 \equiv 1$ , return 1 take  $\dot{x} \neq x$ ;
5.  $x^{\frac{r}{2}} \bmod N \equiv -1$ , return 1 take  $\dot{x} \neq x$ ;
6.  $\gcd(x^{\frac{r}{2}} - 1, N) \gcd(x^{\frac{r}{2}} + 1, N) = N$ .

Where gcd represents the Greatest Common Divisor.

In the above steps, the difficulty lies on solving the modular exponentiation inverse element of the remainder 1 specified in Step 3. Step 3 is transformed into the following problem which is solved by a quantum algorithm:

Given  $f(x) = x^a \bmod N$ ,  $f(a + r) = f(a)$ , find the minimum  $r$ .

Below is an introduction to the core content of quantum algorithm used to solve the modular exponentiation inverse element which mainly consists of three parts.

- 1.Pre-lemma required for formula deformation.
- 2.Available modular multiplication quantum gate operations are constructed to iteratively complete the construction of quantum state of the modular exponentiation inverse element.
- 3.We refer to QPE to obtain the modular exponentiation inverse element through inverse quantum Fourier transform of the results of modular multiplication in the form of summation as constructed.

Due to space constraints, the pre-lemma in Part I will be briefly introduced rather than proved.

### 8.10.2.1 8.10.2.1 Pre-lemma

Define:

$$|u_s\rangle \equiv \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{-\frac{2\pi i k s}{r}} |x^k \bmod N\rangle, x^r \bmod N \equiv 1$$

Lemma1:

$$\frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} e^{\frac{2\pi i k s}{r}} |u_s\rangle = |x^k \bmod N\rangle$$

Lemma2:

$$\exists U, U|y\rangle = |xy \bmod N\rangle, \text{ s.t. } U|u_s\rangle = e^{\frac{2\pi i s}{r}} |u_s\rangle$$

Lemma3:

$$\frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} |u_s\rangle = |1\rangle$$

With lemmas 1, 2, and 3 given, we can relate all the modular exponentiation quantum states, the special quantum state defined  $|u_s\rangle$ , the ground state  $|1\rangle$  and the modular exponentiation inverse element  $r$  through quantum Fourier transform/inverse transform and the definition transform/inverse transform of  $|u_s\rangle$

### 8.10.2.2 8.10.2.2 Construction of modular multiplication quantum gate

Define quantum gate operation  $U^j|y\rangle = |yx^j \bmod N\rangle$

For any given integer  $Z$ , through binary expansion with  $t$  digits, we know that

$$U^{2^{t-1}/t-1} U^{2^{t-2}/t-2} \dots U^{2^0 z_0} |1\rangle \approx |1 * x^z \bmod N\rangle$$

Based on the above, the modular exponentiation operation can be implemented by using the modular exponentiation quantum gate.

### 8.10.2.3 Solving of modular exponentiation inverse element

We investigate the quantum state  $|0\rangle^{\otimes t} (|0\rangle^{\otimes L-1}|1\rangle) = |0\rangle^{\otimes t}|1\rangle_L$  composed of two registers, and initialize the first register to the maximum superposition state to get

$$(H^{\otimes t} \otimes I^{\otimes L}) (|0\rangle^{\otimes t}|1\rangle_L) = |+\rangle^{\otimes t} \otimes |1\rangle_L$$

Based on the quantum gate operation  $U^j$ , we can define the controlled modular exponentiation quantum gate  $C = U^j$ . We take the  $j$ th item of  $|+\rangle^{\otimes t}$  as the control qubit to perform  $t$  times of  $C = U^{2^{j-1}}$  to  $|+\rangle^{\otimes t} \otimes |1\rangle_L$  to complete the controlled modular exponentiation quantum gate operation, thus getting

$$\begin{aligned} & \prod_{j=1}^t (C - U^{2^{j-1}}) (|+\rangle^{\otimes t} \otimes |1\rangle_L) \\ &= \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle |x^j \bmod N\rangle \\ &= \frac{1}{\sqrt{r2^2}} \sum_{j=0}^{2^t-1} \sum_{s=0}^{r-1} e^{\frac{2\pi i js}{r}} |j\rangle |u_s\rangle =: |\psi\rangle \end{aligned}$$

IQFT is performed to the first register to get

$$(\text{QFT}^{-1} \otimes I^{\otimes L}) |\psi\rangle = \frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} \left| \frac{2^t s}{r} \right\rangle |u_s\rangle$$

We measure the first register to get any quantum state rather than  $|0\rangle$ , thus to obtain the integer  $\left\lceil \frac{2^t s}{r} \right\rceil$  which is closest to the real number  $\frac{2^t s}{r}$ . Then, we get  $\frac{s}{r}$  through continued fraction expansion of the real number  $\frac{\left\lceil \frac{2^t s}{r} \right\rceil}{2^t}$ , thus obtaining the denominator  $r$ .

Here  $L = n = \lceil \log_2 N \rceil$ . If  $t = 2n + 1 + \lceil \log(2 + \frac{1}{2\varepsilon}) \rceil$ , we can obtain the phase estimation result with a binary expansion precision of  $2n+1$  bits, and the probability of the result obtained is at least  $\frac{1-\varepsilon}{r}$ . Generally, we take  $t=2n$ .

### 8.10.3 Quantum circuit diagram and reference code

The quantum circuit diagram of Shor's Algorithm is as below.

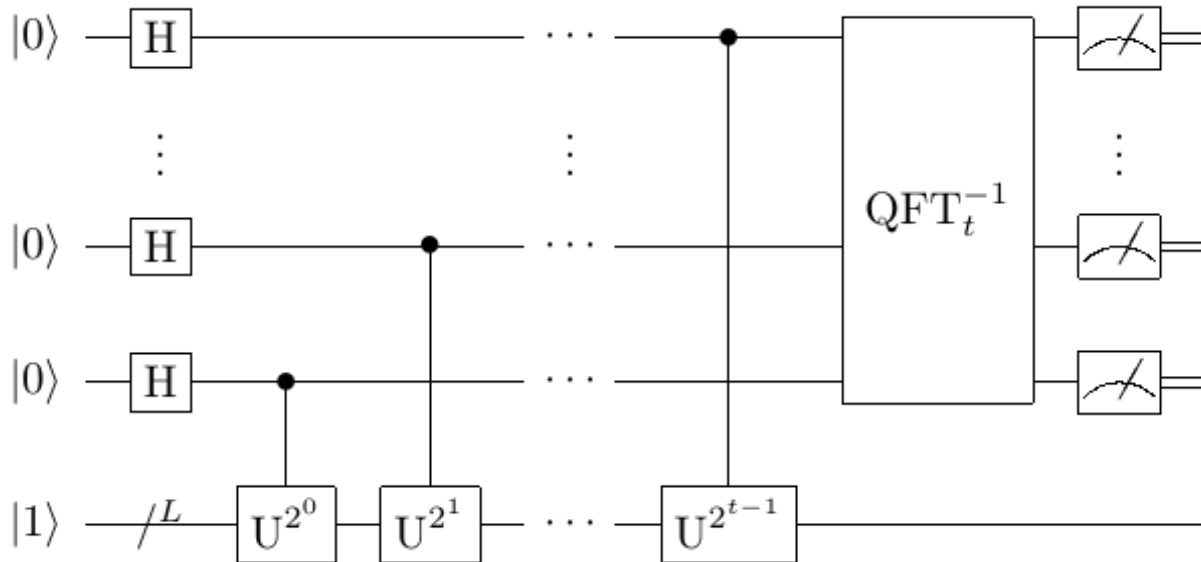
The source code of Shor's Algorithm based on pyQPanda is shown in the Shor's Algorithm program source code under pyQPanda.

Below is the Shor's Algorithm calling interface provided in pyQPanda.

```
Shor_factorization(int)
```

The input parameter is the large number decomposed by prime factorization, with a 2D list returned. The contents are whether the computing process is successful and the list of decomposed prime factor pairs.

We take  $N=15$  to verify the code of Shor's Algorithm as below:



```
#!/usr/bin/env python

import pyqpanda as pq

if __name__ == "__main__":

    N=15
    r = pq.Shor_factorization(N)
    print(r)
```

The prime factor decomposition result of 15 should be  $15=3*5$ , and therefore the algorithm success sign should be returned together with the two prime factors 3 and 5.

```
(True, (3, 5))
```

## 8.11 Quantum imaginary time evolution

Imaginary time evolution is a powerful tool for studying quantum systems. As a classical quantum hybrid algorithm, the imaginary time evolution algorithm can approximately get the ground state vector of any system where the Hamiltonian  $H$  is given which is the eigen vector corresponding to the minimum eigenvalue of  $H$ . This algorithm is provided with a quantum circuit easy to be implemented and characterized by a wide range of applications. It can solve some problems which are hard to be solved by the classical algorithm.



### 8.11.1 Overview of background

A system where the Hamiltonian  $H$  is given evolves according to the propagator  $e^{-iHt}$  over time  $t$ . The corresponding virtual time ( $\tau = it$ ) propagator is  $e^{-H\tau}$  which is a non-unitary operator.

With the Hamiltonian  $H$  and initial state  $|\psi\rangle$  given, the normalized imaginary time evolution is defined as below.

$$|\psi(\tau)\rangle = A(\tau)e^{-H\tau}|\psi(0)\rangle, A(\tau) = (\langle\psi(0)|e^{-2H\tau}|\psi(0)\rangle)^{\frac{1}{2}}$$

$A(\tau)$  is the normalized factor. Generally, the Hamiltonian  $H$  of a multibody system is  $H = \sum_i \lambda_i h_i$  where  $\lambda_i$  is the real coefficient and  $h_i$  is the observables and can be expressed as the direct product of Pauli matrices.

Thus, we obtain the following equivalent Schrodinger Equation:

$$\frac{\partial|\psi(\tau)\rangle}{\partial\tau} = -\left(H - \frac{A'(\tau)}{A(\tau)}\right)|\psi(\tau)\rangle = -(H - E_\tau)|\psi(\tau)\rangle$$

---

#### Note

In practical applications, the real difficulty of QITE lies in how to transform the original problem into the ground state problem of Hamiltonian system and how to give the Hamiltonian to the Hamiltonian system.

---

### 8.11.2 Principle of algorithm

The quantum imaginary time evolution algorithm consists of 2 parts:

Based on the given problem system Hamiltonian, we construct the corresponding Schrodinger Equation and transform the solution problem of Schrodinger Equation into that of a system of linear equations.

We solve the system of linear equations to obtain the time evolution function of key variables. Also, we get the ground state corresponding to the lowest energy of the system by taking advantage of the characteristics of imaginary time evolution, so as to solve the problem.

The quantum imaginary time evolution algorithm is applicable to solving the state at any time and the final steady state from the initial state in any Hamiltonian system with the Hamiltonian known.

#### 8.11.2.1 Approximate solution from Schrodinger Equation to differential equation

Consider the Wick rotation of the Schrodinger Equation satisfied by the given Hamiltonian  $H$

$$\left(\frac{\partial}{\partial\tau} - (H - E_\tau)\right)|\psi(\tau)\rangle = 0, E_\tau = \langle\psi(\tau)|H|\psi(\tau)\rangle$$

Apply the McLachlan variational principle to get

$$\delta\left\|\left(\frac{\partial}{\partial\tau} - (H - E_\tau)\right)|\psi(\tau)\rangle\right\| = 0$$

Take the test state  $|\phi(\vec{\theta}(\tau))\rangle$ ,  $\vec{\theta}(\tau) = (\theta_1(\tau), \theta_2(\tau), \dots, \theta_N(\tau))$  to approximate the solution  $|\psi(\tau)\rangle$

Write  $\dot{\theta}_j = \frac{\partial \theta_j}{\partial \tau}$ ,  $S = (\frac{\partial}{\partial \tau} - (H - E_\tau))$  and meanwhile consider the normalization condition  $\langle \phi | \phi \rangle = 1$  to get

$$\begin{aligned} & \frac{\partial \|S|\phi(\tau)\rangle\|}{\partial \dot{\theta}_i} \\ &= \sum_{i,j} \frac{\partial \langle \phi |}{\partial \theta_i} \frac{\partial |\phi\rangle}{\partial \theta_j} \dot{\theta}_j + \sum_i \left( \frac{\partial \langle \phi |}{\partial \theta_i} H |\phi\rangle + \langle \phi | H \frac{\partial |\phi\rangle}{\partial \theta_i} \right) \\ &= \sum_j A_{ij} \dot{\theta}_j - C_j = 0 \end{aligned}$$

Where

$$\begin{aligned} A_{ij} &= \text{Re} \left( \frac{\partial \langle \phi |}{\partial \theta_i} \frac{\partial |\phi\rangle}{\partial \theta_j} \right) \\ C_i &= -\text{Re} \left( \frac{\partial \langle \phi |}{\partial \theta_i} H |\phi\rangle \right) \end{aligned}$$

Thus, the original Schrodinger Equation is transformed into a system of linear equations of which the solution is  $\dot{\theta}_j$ .

### 8.11.2.2 8.11.2.2 Imaginary time evolution approaching ground state

$x^\dagger A x > 0$  shows that A is positive definite, so is its generalized inverse  $A^{-1}$ .

Therefore, the average energy  $E_\tau$  of the system is as below.

$$\begin{aligned} \frac{dE_\tau}{d\tau} &= \frac{d\langle \psi(\tau) | H | \psi(\tau) \rangle}{d\tau} \\ &= \sum_i \text{Re} \left( \frac{\partial \langle \phi |}{\partial \theta_i} H |\phi\rangle \dot{\theta}_i \right) = - \sum_i C_i \dot{\theta}_i = - \sum_{i,j} C_i A_{i,j}^{-1} C_j \leq 0 \end{aligned}$$

As shown above, the application of quantum imaginary time evolution algorithm will result in continuous decrease of the average energy of the whole system.

Take the test state  $|\phi(\vec{\theta})\rangle = V(\vec{\theta})|\bar{0}\rangle = U_N(\theta_N) \cdots U_2(\theta_2) U_1(\theta_1) |\bar{0}\rangle$ , where  $U_i$  is the unitary operator,  $\bar{0}$  is the initial state of the system (instead of the ground state  $|0\rangle$ ).

Without loss of generality, we can assume that each  $U_i$  depends on only one parameter,  $\theta_i$  is a rotating or controlled rotating gate, and its derivative can be expressed as  $\frac{\partial U_i(\theta_i)}{\partial \theta_i} = \sum_k f_{k,i} U_i(\theta_i) \sigma_{k,i}$ , where  $\sigma_{k,i}$  are unitary operators and  $f_{k,i}$  are scalar functions. Consequently, the derivative of the test state can be expressed as  $\frac{\partial \phi(\tau)}{\partial \theta_i} = \sum_k f_{k,i} \tilde{V}_{k,i} |\bar{0}\rangle$

where

$$\tilde{V}_{k,i} = U_N(\theta_N) \cdots U_{i+1}(\theta_{i+1}) U_i(\theta_i) \sigma_{k,i} \cdots U_2(\theta_2) U_1(\theta_1)$$

Then, the differential equation  $\sum_j A_{ij} \dot{\theta}_j = C_j$  satisfies

$$\begin{aligned} A_{ij} &= \text{Re} \left( \sum_{k,l} f_{k,i}^* f_{l,i} \langle \bar{0} | \tilde{V}_{k,i}^\dagger \tilde{V}_{l,i} | \bar{0} \rangle \right) \\ C_i &= -\text{Re} \left( \sum_{k,l} f_{k,i}^* \lambda_l \langle \bar{0} | \tilde{V}_{k,i}^\dagger h_l V | \bar{0} \rangle \right) \end{aligned}$$

The above two expressions are in line with the general form  $a \operatorname{Re} (e^{i\theta} \langle 0|U|\bar{0}\rangle)$ , and thus we can use a quantum circuit to construct  $A_{ij}$  as below:

$$\langle \bar{0} | \hat{V}_{k,i}^\dagger \tilde{V}_{l,j} | \bar{0} \rangle = \langle \bar{0} | U_1^\dagger \cdots U_{i-1}^\dagger \sigma_{k,i}^\dagger U_i^\dagger \cdots U_{j-1}^\dagger \sigma_{l,j} U_j \cdots U_1 | \bar{0} \rangle$$

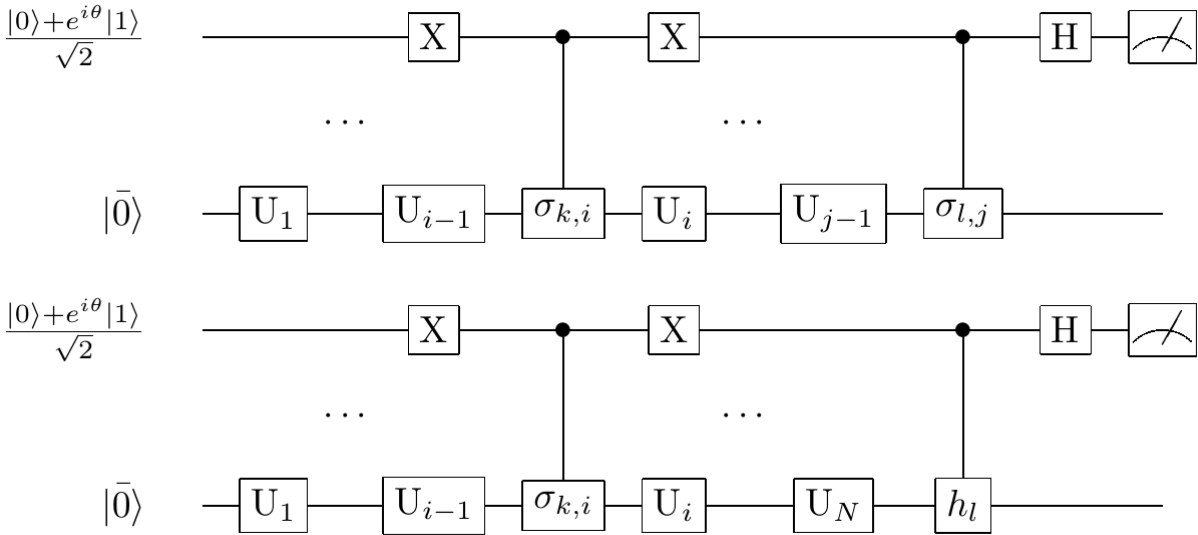
$C_{ij}$  is provided with the similar result. Thus, we can use a quantum circuit to construct  $A_{ij}$  and  $C_{ij}$ .

Therefore, we can introduce the quantum algorithm of system of linear equations, and obtain  $\dot{\theta}_j = \frac{\partial \theta_j}{\partial \tau}$  upon solving. Furthermore, imaginary time revolution can be performed to  $\phi(\vec{\theta})$  to obtain the ground state  $\theta$  under stable state of the system.

Thus, we complete the approximate solution of the ground state corresponding to any given Hamiltonian  $H$ .

### 8.11.3 Quantum circuit diagram and reference code

Below is the quantum circuit diagram of the left-hand item matrix and right one for constructing the system of linear equations in QITE algorithm.



The code implementation of QITE algorithm based on pyQPanda is shown in the QITE algorithm program source code under pyQPanda. The codes related to QITE algorithm in pyQPanda are included a category. Below is an introduction to all relevant input and output interface functions.

```
qite=QITE()
qite.set_Hamiltonian(Hamiltonian)
qite.set_ansatz_gate(ansatz)
qite.set_iter_num(int)
qite.set_delta_tau(float)
qite.set_upthrow_num(int)
qite.set_para_update_mode(GD_VALUE/GD_DIRECTION)
```

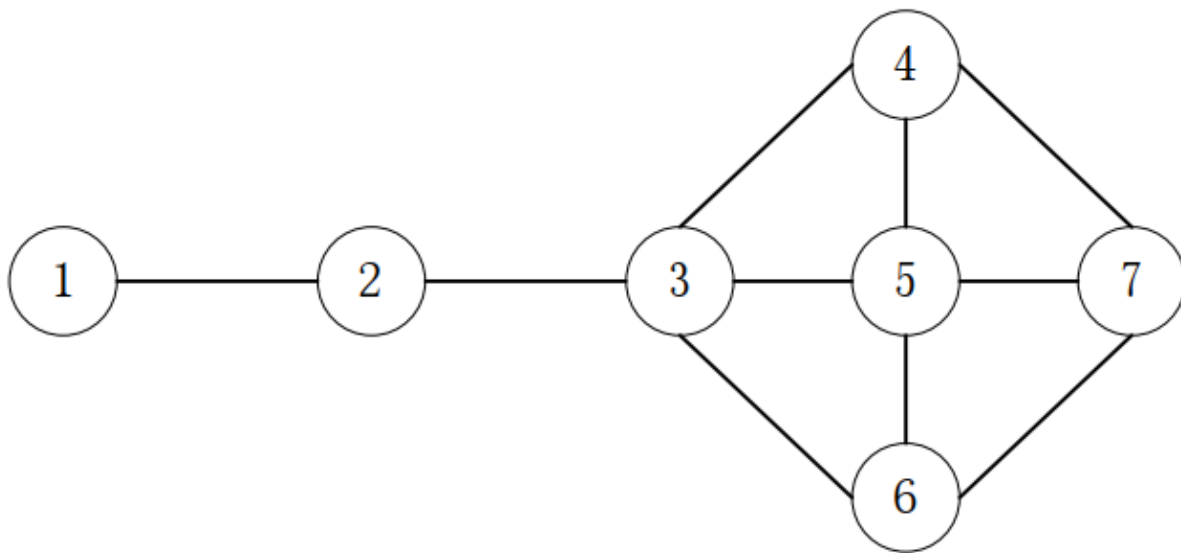
(continues on next page)

(continued from previous page)

```
qite.exec()
qite.get_result()
```

Among the above functions, the first function is the constructor of a class, and the following 6 ones serve to set the Hamiltonian, set the number of iteration(s) and the change rate of  $\tau$ , reset the number of iteration(s) and the reference gradient value or direction of convergence mode, perform imaginary time evolution and obtain the probability result of the list.

We can apply the quantum variational imaginary time evolution algorithm to the importance ranking of network nodes and quickly solve the importance weight of the nodes based on the existing conclusions. We select the importance ranking of network nodes as shown below for code implementation.



```
#!/usr/bin/env python

import pyqpanda as pq
import numpy as np

if __name__ == "__main__":
    node7graph = [[0, 1, 0, 0, 0, 0, 0],
                  [1, 0, 1, 0, 0, 0, 0],
                  [0, 1, 0, 1, 1, 1, 0],
                  [0, 0, 1, 0, 1, 0, 1],
                  [0, 0, 1, 1, 0, 1, 1],
                  [0, 0, 1, 0, 1, 0, 1],
                  [0, 0, 0, 1, 1, 1, 0]],

    problem = pq.NodeSortProblemGenerator()
```

(continues on next page)

(continued from previous page)

```

problem.set_problem_graph(node7graph)
problem.exec()
ansatz_vec = problem.get_ansatz()

cnt_num = 1
iter_num = 100
upthrow_num = 3
delta_tau = 2.6
update_mode = pq.UpdateMode.GD_DIRECTION

for cnt in range(cnt_num):
    qite = pq.QITE()
    qite.set_Hamiltonian(problem.get_Hamiltonian())
    qite.set_ansatz_gate(ansatz_vec)
    qite.set_iter_num(iter_num)
    qite.set_delta_tau(delta_tau)
    qite.set_upthrow_num(upthrow_num)
    qite.set_para_update_mode(update_mode)
    ret = qite.exec()
    if ret != 0:
        print(ret)
    qite.get_result()

```

Below is an example of the QITE solution code.

We can directly deduce that the node with the greatest importance in this 7-node network diagram shall be No. 3 node. Therefore, the result shall throw out No. 3 node, i.e., the most important node, written as 00000100:1.00. The output result as shown below satisfies the expectation.

```
4 0.999967
```



## INDEX

### V

- VariationalQuantumGate (C++ *class*), 129
- VariationalQuantumGate::feed (C++ *function*), 130
- VariationalQuantumGate::get\_constants (C++ *function*), 130
- VariationalQuantumGate::get\_vars (C++ *function*), 130
- VariationalQuantumGate::n\_var (C++ *function*), 129
- VariationalQuantumGate::var\_pos (C++ *function*), 130
- VariationalQuantumGate::VariationalQuantumGate (C++ *function*), 129
- VariationalQuantumGate\_CNOT (C++ *class*), 132
- VariationalQuantumGate\_CNOT::VariationalQuantumGate\_CNOT (C++ *function*), 132
- VariationalQuantumGate\_CZ (C++ *class*), 131
- VariationalQuantumGate\_CZ::VariationalQuantumGate\_CZ (C++ *function*), 132
- VariationalQuantumGate\_H (C++ *class*), 130
- VariationalQuantumGate\_H::VariationalQuantumGate\_H (C++ *function*), 130
- VariationalQuantumGate\_RX (C++ *class*), 131
- VariationalQuantumGate\_RX::VariationalQuantumGate\_RX (C++ *function*), 131
- VariationalQuantumGate\_RY (C++ *class*), 131
- VariationalQuantumGate\_RY::VariationalQuantumGate\_RX (C++ *function*), 131
- VariationalQuantumGate\_RY::VariationalQuantumGate\_RY (C++ *function*), 131
- VariationalQuantumGate\_RZ (C++ *class*), 131
- VariationalQuantumGate\_RZ::VariationalQuantumGate\_RZ (C++ *function*), 131