

# 量子计算

## —编程篇

# Quantum Computer

网址: [www.qubits.top](http://www.qubits.top)

作者: Calvin Tang

邮箱: [179209347@qq.com](mailto:179209347@qq.com)

# 介绍

本开发教程基于本源量子的Qpanda框架的python版 – **PyQPanda** 编写。

- 一种功能齐全，运行高效的量子软件开发工具包
- QPanda 2是由本源量子开发的开源量子计算框架，它可以用于构建、运行和优化量子算法。
- QPanda 2作为本源量子计算系列软件的基础库，为OriginIR、Qurator、量子计算服务提供核心部件。

## 本教程包含：

1. 泡利算符类
2. 费米子算符类
3. 优化算法（直接搜索法）

## QPanda使用文档：

<https://pyqpanda-tutorial.readthedocs.io/zh/latest/index.html>

## Github & Gitee 代码地址：




[https://github.com/mymagicpower/quantum/tree/main/quantum\\_qpanda/components](https://github.com/mymagicpower/quantum/tree/main/quantum_qpanda/components)

[https://gitee.com/mymagicpower/quantum/tree/main/quantum\\_qpanda/components](https://gitee.com/mymagicpower/quantum/tree/main/quantum_qpanda/components)



# 泡利算符类

泡利算符是一组三个 $2 \times 2$ 的么正厄米复矩阵，又称酉矩阵。我们一般都以希腊字母  $\sigma$ （西格玛）来表示，记作  $\sigma_x$ ， $\sigma_y$ ， $\sigma_z$ 。  
 在 QPanda 中我们称它们为 X 门，Y 门，Z 门。它们对应的矩阵形式如下表所示。

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

1. 泡利算符与自身相乘得到是单位矩阵

$$\sigma_x \sigma_x = I$$

$$\sigma_y \sigma_y = I$$

$$\sigma_z \sigma_z = I$$

2. 泡利算符与单位矩阵相乘，无论是左乘还是右乘，其值不变

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

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

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

3. 顺序相乘的两个泡利算符跟未参与计算的泡利算符是  $i$  倍的关系

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

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

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

4. 逆序相乘的两个泡利算符跟未参与计算的泡利算符是  $-i$  倍的关系

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

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

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

# 泡利算符类

根据泡利算符的上述性质，我们在 pyQPanda 中实现了泡利算符类 PauliOperator。我们可以很容易的构造泡利算符类，例如：

```
from pyqpanda import *

if __name__=="__main__":
    # 构造一个空的泡利算符类
    p1 = PauliOperator()

    # 2倍的"泡利Z0"张乘"泡利Z1"
    p2 = PauliOperator("Z0 Z1", 2)

    # 2倍的"泡利Z0"张乘"泡利Z1" + 3倍的"泡利X1"张乘"泡利Y2"
    p3 = PauliOperator({"Z0 Z1": 2, "X1 Y2": 3})

    # 构造一个单位矩阵，其系数为2，等价于 p4 = PauliOperator("", 2)
    p4 = PauliOperator(2)
```

其中PauliOperator p2("Z0 Z1", 2)表示的是

$$2\sigma_0^z \otimes \sigma_1^z$$

0\_PauliOperator.py

```
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())
```

# 费米子算符类

我们用如下的记号标识来表示费米子的两个形态，湮没:  $X$  表示  $a_x$ ，创建:  $X+$  表示  $a_x^\dagger$ ，例如:  
 "1+ 3 5+ 1"则代表  $a_1^\dagger a_3 a_5^\dagger a_1$

整理规则如下

## 1. 不同数字

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

## 2. 相同数字

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

## 费米子算符类

跟 PauliOperator 类似，FermionOperator 类也提供了费米子算符之间加、减和乘的基础的运算操作。通过整理功能可以得到一份有序排列的结果。

### 1\_FermionOperator.py

```
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())
```

输出：

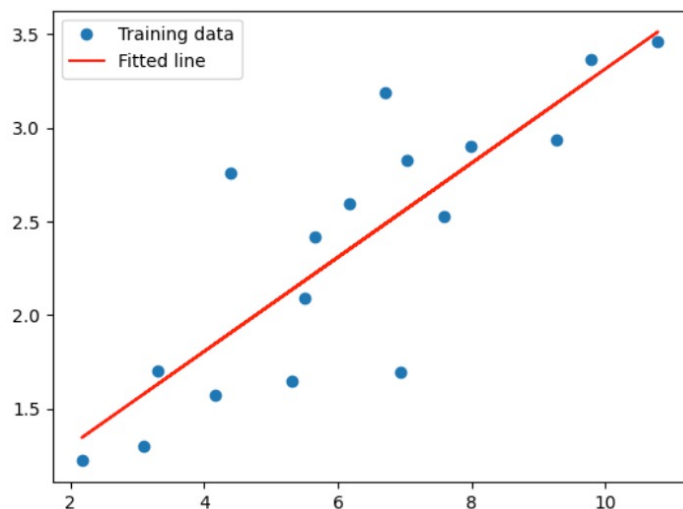
```
a + b = {
0 1+ : 2.000000
2+ 3 : 3.000000
}
a - b = {
0 1+ : 2.000000
2+ 3 : -3.000000
}
a * b = {
0 1+ 2+ 3 : 6.000000
}
normal_ordered(a + b) = {
1+ 0 : -2.000000
2+ 3 : 3.000000
}
normal_ordered(a - b) = {
1+ 0 : -2.000000
2+ 3 : -3.000000
}
normal_ordered(a * b) = {
2+ 1+ 3 0 : 6.000000
}
```

## 优化算法(直接搜索法)

本章节将讲解优化算法的使用，包括 Nelder-Mead 算法跟 Powell 算法，它们都是一种直接搜索算法。我们在 QPanda 中实现了这两个算法，OriginNelderMead 和 OriginPowell，这两个类都继承自 AbstractOptimizer。

我们可以通过优化器工厂生成指定类型的优化器，例如我们指定它的类型为 Nelder-Mead。

```
OptimizerFactory.makeOptimizer(OptimizerType.NELDER_MEAD)
```



<https://pyqpanda-tutorial.readthedocs.io/zh/latest/Optimizer.html>

## 2\_Optimizer.py

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



Thank

You